Summer 9-10-2019

Correct-by-Construction Typechecking with Scope Graphs

Katherine Imhoff Casamento
Portland State University

Recommended Citation
Casamento, Katherine Imhoff, "Correct-by-Construction Typechecking with Scope Graphs" (2019). *Dissertations and Theses.* Paper 5272.
https://doi.org/10.15760/etd.7145

Correct-by-Construction Typechecking with Scope Graphs

by

Katherine Imhoff Casamento

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Andrew Tolmach, Chair
Mark P. Jones
Bart Massey

Portland State University
2019

Abstract

Dependently-typed languages are well-known for the ability to enforce program invariants through type signatures, and previous work establishes the effectiveness of this style of program verification in the implementation of type-safe interpreters for a wide class of languages with a variety of interesting scoping semantics, offering an account of dynamic semantics. This thesis covers the complementary topic of static semantics, in the form of a pattern for constructing verified typechecking procedures in a dependently-typed setting. Implementations are given for simply-typed lambda calculus and a small procedural language as well as a module system with unrestricted cyclic module dependency semantics that are traditionally hard to formalize, parameterized over the choice of base language. A library of finite graphs and decision procedures for path search queries is presented and used in the construction of the example language implementations to resolve variable references. The resulting development is suitable as a static analysis phase ("middle end") in a hypothetical end-to-end verified interpreter developed in a dependently-typed setting.

Table of Contents

Chapter 1 Introduction

In the dependently-typed idiom of *correct-by-construction* programming, program specifications are encoded in data types with intrinsic invariants such that any well-typed program is guaranteed to respect the invariants. The best examples of correct-by-construction code elegantly combine the computational definition of a program with its own proof of correctness, eliminating the need for extrinsic program validation and incurring little or no syntactic proof overhead beyond the encoding of invariants in data types. In particular, some object languages admit very natural implementations of type-preserving-by-construction denotational semantics over *intrinsically-typed* abstract syntax: the type of an AST term in the host language is indexed by a representation of its object language type, ensuring that every valid AST represents a well-typed term in the object language. Unfortunately, it quickly becomes challenging to tame complexity and proof overhead in intrinsically-typed implementations of object languages featuring advanced scoping and typing rules, and especially to maintain readability; at worst, the dependently-typed encoding of the specification of a program can be too complicated for a human being to reliably audit against a less formal and more readable specification that it claims to encode.

Bach Poulsen et al. [3] give a formalization of the *scope graph calculus* [10] in Agda and show that their framework enables relatively clean implementations of intrinsically-typed syntax tree types and type-preserving interpreters for languages with sophisticated scoping semantics, giving a formalization of Middleweight Java [4] as an example. This thesis builds on their work with a similar focus on readability, covering the specification and implementation of typechecking procedures in this setting.

1.1 Background

A canonical example of the power and expressivity of dependently-typed languages is the implementation of a type-safe evaluator for simply-typed lambda calculus (STLC), given in Agda below.

```
data Type : Set where
  ⟨_⟩ : Set → Type
  _⇒_ : Type → Type → Type

data Expr Γ : Type → Set where
  con : A → Expr Γ ⟨ A ⟩
  var : t ∈ Γ → Expr Γ t
  ƛ : Expr (t₁ :: Γ) t₂ → Expr Γ (t₁ ⇒ t₂)
  _•_ : Expr Γ (t₁ ⇒ t₂) → Expr Γ t₁ → Expr Γ t₂

Val : Type → Set
Val ⟨ A ⟩ = A
Val (t₁ ⇒ t₂) = Val t₁ → Val t₂

eval : All Val Γ → Expr Γ t → Val t
eval ρ (con a) = a
eval ρ (var i) = lookup i ρ
eval ρ (ƛ e) = λ v → eval (v :: ρ) e
eval ρ (e₁ • e₂) = eval ρ e₁ (eval ρ e₂)
```

This definition can be read as a denotational semantics for STLC using Agda as the metalanguage: Val gives semantics to object language types (Type) in terms of

metalanguage types (Set), and eval gives semantics to object language expressions (Expr) in terms of metalanguage values (Val). The proof of type preservation for eval is *by construction*, in the sense that there is no explicit proof code and the property holds *definitionally* in each case according to the semantics of Agda, aided by the invariants encoded in the Expr type definition. For example, the λ expression form is defined to have an object-language type constructed with _⇒_, which Val interprets as an Agda-level function type. These definitions constrain the right-hand side of the corresponding case in eval to be an Agda function from a metalanguage value of the input type to a metalanguage value of the output type, and the Agda typechecker will report an error if the implementation is not type-correct in this way.

The STLC definition above also serves as a functioning interpreter for expressions of the Expr type. Constructing these expressions by hand is somewhat tedious but relatively straightforward, as demonstrated in the definition of id$\mathbb{N}$ below; the syntax here refl in this context is denoting the 0^{th} de Bruijn index. The Agda typechecker catches any type or scoping errors in the definition of id$\mathbb{N}$, for example ensuring that 0 is the only usable variable index in the body of a lambda with no free variables. The eval function is used in the definition of id$\mathbb{N}\prime$ to evaluate id$\mathbb{N}$ in an empty environment, yielding an Agda-level identity function over natural numbers.

```
idℕ : Expr [] (⟨ ℕ ⟩ ⇒ ⟨ ℕ ⟩)
idℕ = ƛ (var (here refl))

idℕ′ : ℕ → ℕ
idℕ′ = eval [] idℕ
```

The multifaceted nature of this style of denotational semantics offers distinct potential: the traditional processes of defining a formal semantics, implementing it in software, and verifying the implementation are unified into a single development process, avoiding the possibility of translation errors during implementation and possibly cutting down

significantly on code size and development effort. The resulting Agda program is roughly suitable as a “reference implementation” of the evaluation semantics of STLC with de Bruijn indices, for testing example programs with and testing other implementations against.

Of course, pure STLC with de Bruijn indices is not realistically representative of all interesting object languages. STLC is effectively a heavily restricted sublanguage of Agda and the construction of STLC interpreters in purely functional settings is well understood; it is not immediately clear whether this style of defining semantics scales to more complex languages that do not have these properties.

Bach Poulsen et al. [3] show that it is feasible to define intrinsically-typed syntax types and interpreters in this style for languages such as Middleweight Java [4] with complex scoping semantics much different from Agda’s, using a *scope graph* [10] framework to formalize variable binding. Their AST types and interpreter code are claimed to be readable as a formal semantics, but they leave for future work another desirable aspect of simpler correct-by-construction interpreters: a convenient syntax for reading and writing the object language terms that the interpreter operates over. Among other overhead, constructing variable terms in their intrinsically-typed syntax requires the manual construction of paths through a scope graph, and the scope graph itself for a program must also be constructed by hand to match the binding structure of the program. These manual processes are error-prone and often require some amount of debugging to get right.

This thesis covers the process of generating a scope graph to match the binding structure of a given “raw” term AST with no intrinsic invariants along with the scopechecking and typechecking of these terms. The target set of object languages is specifically the class of languages where these can be defined as orthogonal phases, i.e. where scope

graph construction does not depend on querying a partially constructed graph as it does in van Antwerpen et al. [13]. To demonstrate the pattern of development and illustrate that this class of languages includes nontrivial examples, implementations are given for STLC and a toy procedural language that includes array and pointer types, along with a small library for adding module system functionality to an arbitrary object language. The procedural language with the module system is sufficiently expressive that standard procedural pseudocode for an in-place quicksort algorithm can be directly translated to it, as demonstrated in the final example, and the module system fully supports cyclic module imports to demonstrate a natural feature of the scope graph theory that is often challenging to formally model in other settings.

The ability to write raw terms and typecheck them within Agda provides a solution to the issue of intrinsically-typed terms being challenging and tedious to construct by hand, which is useful when writing example and test programs, and also serves as a method of implementing a standard phase in a typical compiler or interpreter for a language. Taken along with the interpretation techniques described in Bach Poulsen et al. [3], the only remaining missing piece of an end-to-end interpreter is a frontend to parse source text into raw terms, which can be defined in a total parser combinator library like Agdarsec [1]. The potential benefits are clear, if readability is indeed maintained: the entire syntax and semantics of an object language can be defined in a style simultaneously suitable for both human and machine consumption, with the correct-by-construction approach guiding development and eliminating most of the need for manual verification.

This thesis does not cover the construction of a complete end-to-end interpreter, but contributes the missing part of the “middle end” in the form of a pattern for defining correct-by-construction typechecking procedures for intrinsically-typed object languages. While the focus is not on the interpretation of object language programs, the intrinsically-

typed terms that the typechecker outputs are claimed to be immediately suitable for interpretation in the style of Bach Poulsen et al. [3], which is justified in an Appendix detailing the evaluation of programs in the example procedural language with modules.

1.2 Specifying typechecker correctness

1.2.1 Theory

What is the specification of the correct behavior of a typechecker? It should be both *sound* and *complete* with respect to a defined typing semantics, in the sense that the user should be able to trust the veracity of both “yes” and “no” answers to questions about typechecking and type inference. It should also be able to report reliably on *ambiguity* in variable resolution, in cases where the (partial) specificity ordering does not determine a unique “best” resolution.

Typechecking and type inference procedures are often phrased as decision procedures that decide whether a typing derivation exists for a given input term. When working with intrinsically-typed syntax, these can be phrased as procedures to decide whether there exists an intrinsically-typed term that erases to a given raw input term. This does not account for the complete correctness specification of a typechecker for some object languages, however; specifically, some languages include the possibility of well-formed but semantically ambiguous terms in cases where there may be more than one semantically distinct typing derivation for a raw term according to the rules of the language. This arises in particular in the presence of scoping mechanisms that may introduce ambiguity into the process of variable resolution with the expectation that a language implementation will reject any program with ambiguous variable references.

For example, consider the pseudocode below in an imperative language with modules

where the `main` module executes when the program is run.

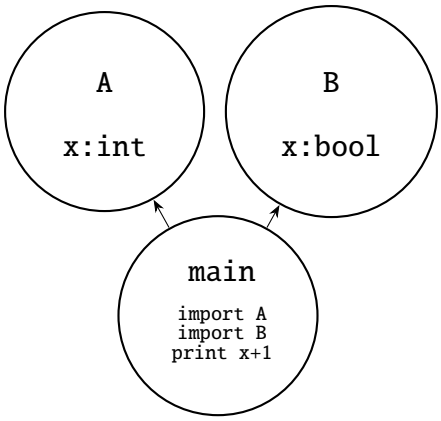
```
module A:
   var x:  int = 0

module B:
   var x:  bool = true
```

```
module main:
   import A
   import B
   print x+1
```

What are the semantics of this program? The answer is determined by the scoping semantics of the language in question: the ambiguity might be considered an error, or importing `x` from `B` may shadow the import of `x` from `A` so that the program has a type error when attempting to add `x` to `1`, or some form of type-directed resolution may recognize that the reference from `A` is uniquely well-typed, or the reference to `x` may be resolved by some other criteria entirely.

Defining the static semantics of the language in a scope graph framework yields a convenient visualization of the scoping structure of a program, which can be used to illustrate this variety of choices. The directed graph below represents one interpretation of the binding structure for the program given above; each node is a scope and the paths in the graph represent an accessibility relation between scopes. In this case, each edge in the graph represents a module import statement importing the contents of the destination scope into the source scope.

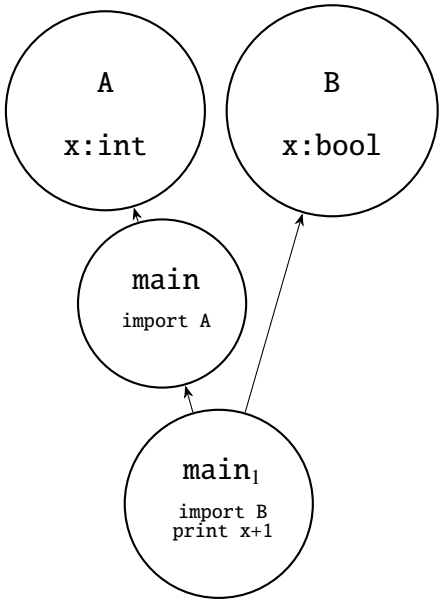


In a scope graph framework, the definition of an object language includes a *specificity ordering* and *well-formedness predicate* [10] over variable resolution paths that respectively order and filter the set of valid resolutions for the variables in a program. The process of variable resolution searches from the scope containing some variable reference for all *reachable* scopes that bind the same variable name; a scope in the scope graph is reachable from some origin scope when there exists a path to it that is maximal according to the specificity ordering and that meets the well-formedness predicate. When the reference to `x` in `main` is resolved, it resolves based on this set of reachable scopes and potentially a mechanism defined in the object language for dealing with ambiguity.

One notable feature of this style of scoping semantics is that module imports are *transitive* under a trivially satisfiable well-formedness predicate, in the sense that when some module `A` imports `B` and `B` imports `C`, the definitions in `C` are visible in `A`. This can be restricted by strengthening the well-formedness predicate, for example by requiring that all variable resolution paths contain at most one edge between two module scopes.

If the language is intended to consider ambiguity an error in all cases (i.e., the semantics forbid variable shadowing), the graph as given above is sufficient and no specificity ordering or well-formedness predicate is required; a path search finds that

there are two valid paths that resolve the variable `x` and there is no means to distinguish between them. If the intent is for resolution to be type-directed, the typechecker can inspect this set of paths and see that only one is well-typed. If the intent is for the import of `B` to shadow the import of `A`, an auxiliary scope can be added to represent the implicit sub-scope of `main` encapsulating the next two lines after `import A`, as below, and the specificity ordering defined to prefer shorter paths.



A technique for building correct-by-construction code to generate scope graphs and use them to typecheck terms should account for this variety, in that the mechanism of encoding the specification of a typechecker's correct behavior should be fine-grained enough to express the distinctions between these resolution semantics and many others. A typechecker expressed as a decision procedure for the existence of *some* typing derivation is not sufficient, then, since a typechecker should be able to decide whether a *unique* derivation exists, and this in general requires an inspection of the set of *all* derivations for the given input term.

1.2.2 Implementation

This section roughly traces the development process that led to the type signatures for the correct-by-construction typecheckers constructed in the rest of this thesis. A first attempt at type signatures for typechecking and type inference functions looked like the following, where RawTerm is a standard AST type with no intrinsic invariants and TypedTerm is an intrinsically-typed term:

```
check₁ : (Γ : Ctx) (t : Type) → RawTerm → Maybe (TypedTerm Γ t)
infer₁ : (Γ : Ctx) → RawTerm → Maybe (∃ λ t → TypedTerm Γ t)
```

It is possible to define functions with these signatures that have the correct observable behavior, but the type signatures guarantee neither completeness nor total soundness. They guarantee a partial soundness property, in the sense that they may not output ill-typed terms, but they may fail indiscriminately and may output intrinsically-typed terms with no relation to the input raw terms, as evidenced by the following two well-typed definitions of infer_1.

```
infer₁ _ = nothing
infer₁ _ = just (bool , true)
```

One critical insight is that the intrinsically-typed output term should bear some predictable relation to the raw input term, namely that the output term should *erase* to the input term (i.e., should yield the raw term when all intrinsic invariants are "forgotten"). The following signature is fully sound, using the name Erasure to represent a predicate type that witnesses that its first argument is the erasure of its second.

check_2 :

$(\Gamma : \text{Ctx})\ (t : \text{Type})\ (e : \text{RawTerm}) \rightarrow$

$\text{Maybe}\ (\exists\ \lambda\ (e^t : \text{TypedTerm}\ \Gamma\ t) \rightarrow \text{Erasure}\ e\ e^t)$

The check_2 signature still does not guarantee full completeness, however, as a definition that always returns nothing is still type-correct. A next attempt might involve the Dec type, as in check_3 below.

check_3 :

$(\Gamma : \text{Ctx})\ (t : \text{Type})\ (e : \text{RawTerm}) \rightarrow$

$\text{Dec}\ (\exists\ \lambda\ (e^t : \text{TypedTerm}\ \Gamma\ t) \rightarrow \text{Erasure}\ e\ e^t)$

The check_3 signature is complete and sound in the sense that an implementation is guaranteed by construction to give a yes answer exactly when the input term is well-typed and a no answer exactly when it is not, but the signature ignores *ambiguity* information that may be relevant: in general, the specificity ordering of an object language's definition may be only a partial ordering, so any individual variable reference in a program may have more than one "best" resolution. Object languages may be specified to handle these cases in language-specific ways, so a fully correct typechecker should return the *complete collection* of "best" resolutions.

The approach taken in this thesis is to have a typechecker output a value of the Listable type from Firsov and Uustalu [8], which contains a list of elements of the type along with a total function that assigns a distinct index in this list to each possible value of the type, establishing that the elements of the type are finitely enumerable.

```
check :
  (Γ : Ctx) (t : Type) (e : RawTerm) →
  Listable (∃ λ (e' : TypedTerm Γ t) → Erasure e e')
```

This pattern illustrated by the signature of check above is the one used for typechecker specifications throughout the rest of this thesis.

1.3 Contributions

This thesis makes the following contributions, which are all formalized in Agda.

- A pattern is presented for the implementation of intrinsically-verified typechecking functions in a dependently-typed setting, using scope graphs for variable resolution. The type signatures of these functions are designed such that the entire correctness specification of the typechecker is encoded intrinsically and verified by the Agda typechecker: typechecking is guaranteed to succeed over a uniquely well-typed input term, and guaranteed to fail over ill-typed terms as well as terms involving ambiguous variable references. Typechecking functions in this style are presented for simply-typed lambda calculus (STLC) and a toy procedural language, as concrete examples of the pattern.

- The design and implementation of a generic module system is presented, illustrating that the style of language definition covered in this thesis supports not only the implementation of intrinsically-verified typecheckers for term languages but also higher-order constructs that take the term languages themselves as input. The module system can be instantiated over a choice of arbitrary base language,

including the presentations of STLC and the toy procedural language given in this paper. The system fully supports cyclic dependencies between modules, and the details of scoping semantics, including variable shadowing rules, are left to the base language definition.

- To resolve variable references in a language defined in a scope graph framework, the design and implementation of an intrinsically-verified function for an all-paths finite graph search in a dependently-typed setting is presented, guaranteed by the Agda type system to return the complete set of acyclic paths that originate at a given source node and meet a given predicate.

1.4 Organization

Part I describes a general framework used in Part II to build the example language implementations.

Part I: Framework

Chapter 2 is a brief overview of the Agda language features and idioms used throughout this thesis along with some relevant standard library types, assuming a basic prerequisite knowledge of pure functional programming and the theory of dependent types. Readers experienced with Agda and its standard library can probably skim it without missing anything important. The techniques presented in this thesis are directly portable to similar dependently-typed languages like Idris [5], and should apply in principle in Coq [12] with potentially some differences in encoding.

Chapter 3 is a summary of the Listable type from Firsov and Uustalu [8] and some relevant operations that they define on it. The type is comparable to a list type and

has behavior similar to a container type in general, but the invariant it encodes is fairly restrictive, and the standard higher-order list operations like mapping and filtering require a little more care and proof effort to define over Listable values. In return, the guarantees provided by Listable are very strong: in particular every listable type has decidable cardinality, which is used to check whether the output of a typechecker for some given input term is unique, and a universally- or existentially-quantified predicate is decidable over a Listable type whenever the unquantified predicate is decidable over individual elements of the type.

Chapter 4 presents a type of finite graphs and decision procedures for graph search queries, which are used in the typecheckers in Part II to resolve variable references in scope graphs. The path search implementation is correct by construction, expressed as an all-paths breadth-first search returning a Listable type of all found paths.

Chapter 5 presents a minor generalization of the scope graph framework of Bach Poulsen et al. [3]. Where their implementation uses the Agda standard library Fin type of bounded natural numbers to represent scopes, the implementation in this thesis is parameterized over an arbitrary Listable type. This offers a benefit to code clarity and simplicity in some portions of the code at the cost of an additional requirement to prove inductively-defined scope types finite, and this tradeoff is discussed and illustrated with an example.

Part II: Scopechecking and Typechecking

Chapter 6 presents the abstract syntax and static semantics of a generic module system parameterized over an arbitrary term language, along with an introduction to the high-level pattern of term language definition used in the next two chapters. A module consists of a list of term definitions each given a name and a type annotation, and a

program consists of a list of named modules along with a distinguished “main” term that runs when the program is executed. A module may import another module, bringing the contents of the imported module into scope in the module containing the import. All choices about the set of valid variable resolution paths are deferred to the term language definition, offering some flexibility in the details of the semantics of module imports in the resulting language with modules. A procedure is given to construct a scope graph representing the binding structure of a program, depending on a term language procedure to construct a scope graph to represent the binding structure of an individual term.

Chapter 7 presents a scopechecker and typechecker for simply-typed lambda calculus (STLC) in the scope graph framework, outputting the same kind of intrinsically-typed terms that the STLC interpreter in Bach Poulsen et al. [3] accepts as input. The STLC implementation is suitable as a term language for the module system from Chapter 6. The resulting language of modules containing STLC terms makes conservative choices of specificity ordering and well-formedness predicate, encoding the standard STLC variable shadowing conventions but simply reporting an error for any other ambiguous variable reference (as may be introduced by module import statements).

Chapter 8 presents a toy procedural language with arrays and pointers. This implementation is also suitable as a term language for the module system from Chapter 6. The resulting language of modules containing procedures and module-scoped mutable variables makes a liberal choice of scoping semantics, enabling a flexible form of type-directed overloading and transitive module imports.

The Conclusion discusses some challenges encountered during development and suggests potential future work. An Appendix details the interpretation of programs in the language from Chapter 8, using the same style of intrinsically-typed interpreter definition

presented in Bach Poulsen et al. [3].

Part I

Framework

Chapter 2 Agda

Agda [11] is a purely functional programming language with comprehensive support for dependent types, allowing programs and proofs to be written in the same language. Its syntax is mostly a superset of Haskell's with some modifications, notably that the : and :: operators are swapped to represent "has type" and "list cons" respectively and that full Unicode support is available and used liberally in the Agda standard library [6]. The code in this thesis has been tested and compiled to LaTeX with Agda version 2.6.0.1 and standard library version 1.0.1, and the accompanying development can be found at `http://web.cecs.pdx.edu/~cas28/thesis`. Some definitions are abridged or omitted for brevity in the thesis, with only a type signature and description given.

The intent of this chapter is not to serve as a full-fledged Agda tutorial but to briefly familiarize the reader with the syntax and conventions of Agda code in order to facilitate reading the rest of the thesis. Some basic prerequisite knowledge of pure functional programming and the theory of dependent types are assumed, but the reader is not assumed to have used Agda previously. Most of the types and some of the functions presented are used later in this thesis, and some are just for the sake of an introduction to Agda as a language.

All of the top-level modules defined in the code accompanying this thesis are prefixed with the qualifier Code; all other modules referenced in the thesis are from the Agda standard library. The color and font conventions in use are the default for the Agda LaTeX backend:

keyword `string` number constructor data/record type
record field module function *variable* `comment`

Agda supports a variety of language extensions enabled by flags on the command

line or in source code directives, some of which significantly change some of the logical properties of the language. This thesis uses Agda with no flags except `--type-in-type`, explained below, and `--sized-types` in Chapter 8, explained there. Statements in this thesis about "base Agda" should be interpreted as about the language implemented by the Agda compiler with all flags set to their default values.

2.1 Universes

The type Set is Agda's built-in universe of types. Set is technically an alias for Set_0, the first in an infinite hierarchy of types $\mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : \ldots$, to prevent inconsistencies that arise from self-reference paradoxes if Set : Set. This becomes relevant in practice when defining constructors that take types as arguments, as in some of the record types used throughout this thesis: a record type $A : \mathsf{Set}_n$ may not have any fields of type Set_m for any $m \geqslant n$, so a record of type Set_0 may not have any fields that are types at all. Agda supports a *universe polymorphism* feature for programming generically over these universe levels, so for example an identity function over all types may have the type $(\alpha : \mathsf{Level}) \to (A : \mathsf{Set}_\alpha) \to A \to A$. Library code in Agda is usually written in this style in order to assume as little as possible about the types being used in client code.

Agda also supports a command line option `--type-in-type`, adding the axiom Set : Set so that universe levels can be ignored altogether. As mentioned, this makes the language inconsistent; in programming terms, this means the type system can be circumvented and well-typed programs can "go wrong". While this option is generally unsuitable for the final verified implementation of a program or proof, it offers an attractive tradeoff during development and presentation: the inconsistencies introduced by `--type-in-type` are unlikely to be accidentally exploited by a programmer acting in good faith, and the details of explicit universe polymorphism are almost always irrelevant

to a reader's understanding of a program. Accordingly, the code presented in this thesis has the `--type-in-type` option enabled, and the accompanying development includes a nearly identical version in the `Code/WithoutTypeInType` directory using explicit universe polymorphism that typechecks with it disabled.

2.2 Inductive data types

Inductive data types in Agda are defined in the style of generalized algebraic data types (GADTs) by giving a name and type for a data type itself and then a name and type for each of its constructors. For example, the standard library Bool type (from the standard library module Data.Bool) is defined with two nullary constructors true and false, and the natural number type $\mathbb{N}$ (from Data.Nat) has one nullary constructor zero and one unary constructor suc with an argument of type $\mathbb{N}$.

```
data Bool : Set where
  true false : Bool
```

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

2.3 Functions

The syntax for functions in Agda is almost identical to Haskell's, allowing pattern-matching against constructors on the left-hand side.

```
not : Bool → Bool
not true = false
not false = true
```

Anonymous functions are written with the λ symbol, which associates as far right as

possible - the right-hand side of quadruple parses the same as twice ($\lambda\ x \to$ double x).

```
twice : (ℕ → ℕ) → ℕ → ℕ
twice = λ f x → f (f x)

quadruple : ℕ → ℕ
quadruple = twice λ x → double x
```

All functions in Agda are required to pass a conservative syntactic totality checker, restricting recursion to "obviously" structural recursion; the definition of double below is total because its recursive call in the suc case receives an argument n which is a strict syntactic subpattern of the corresponding input argument pattern suc n.

```
double : ℕ → ℕ
double zero = zero
double (suc n) = suc (suc (double n))
```

Anonymous functions may also case over their arguments with *pattern-matching lambdas* (similar to the `LambdaCase` Haskell extension), denoted with the syntax λ where and delimited by indentation.

```
pred : ℕ → ℕ
pred =
  λ where
    zero → zero
    (suc n) → n
```

2.4 Implicit arguments

An argument can be surrounded in curly braces to denote it as an *implicit argument*, as in the definition of the identity function id below. By default implicit arguments are resolved automatically by unification and do not appear in the source code, as in id-ex_1; an implicit argument can be given explicitly at a call site by enclosing it in curly braces, either as a positional argument as in id-ex_2 or by name as in id-ex_3. A typechecking error is thrown if an implicit argument is not given explicitly and cannot be resolved uniquely by unification.

```
id : {A : Set} → A → A
id x = x

id-ex₁ id-ex₂ id-ex₃ : Bool
id-ex₁ = id true
id-ex₂ = id {Bool} true
id-ex₃ = id {A = Bool} true
```

Similarly, implicit arguments in definitions can be accessed positionally or by name, as in typeof_1 and typeof_2. The underscore pattern has the same meaning as in Haskell, leaving the argument unnamed.

```
typeof₁ : {A : Set} → A → Set
typeof₁ {X} _ = X

typeof₂ : {A : Set} → A → Set
typeof₂ {A = X} _ = X
```

The special underscore term explicitly directs Agda to attempt to produce a unique term by the same unification procedure that resolves implicit arguments.

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

id′-ex : Bool
id′-ex = id′ _ true
```

2.5 Mixfix operators

Agda offers flexible support for *mixfix* operator definitions: in general, Agda names may contain any number of non-consecutive underscores, which are treated as argument positions, and nearly any character is allowed to be part of a name. For example, the standard library defines the addition function over natural numbers as an infix binary operator.

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

The mixfix style can also be used to define operators with other arities, such as the traditional ternary if/then/else notation. The underscore at the end of the identifier if_then_else_ indicates that the application of this mixfix function to its rightmost argument is right-associative rather than left-associative, so that, for example, if *a* then *b* else *c d* parses as if *a* then *b* else (*c d*) assuming *c* is not a mixfix operator. Operators can be given integer precedence (with higher precedence binding tighter) and left/right/no associativity with the infixl/infixr/infix operators respectively, as in Haskell.

```
infix 0 if_then_else_
if_then_else_ : {A : Set} → Bool → A → A → A
```

```
if true then x else y = x
if false then x else y = y
```

Although Agda has no built-in syntax for casing directly over expressions on the right-hand side of a definition, the standard library Function module provides a mixfix operator with the identifier case_of_ that serves this purpose when combined with a pattern-matching lambda; the type of the pattern-matching lambda in the definition of plus below is $\mathbb{N} \to \mathbb{N}$, and it implements the same pattern-matching behavior as in _+_.

```
case_of_ : {A B : Set} → A → (A → B) → B
case x of f = f x

plus : ℕ → ℕ → ℕ
plus m n =
  case m of λ where
    zero → n
    (suc m′) → suc (plus m′ n)
```

Mixfix operators also support *operator sections* similar to the feature under the same name in Haskell, but with slightly different syntax: a mixfix operator can be applied with any number of its arguments left out, denoted by an underscore as in the definition of the operator (i.e., not surrounded by spaces). For example, the expression if_then 1 else 0 has type Bool → ℕ and returns 1 for true and 0 for false, and the expression _+ 1 has type $\mathbb{N} \to \mathbb{N}$ and adds its argument to 1.

2.6 Indexed types

An inductive type may have *indices* of any combination of types; in general, the data keyword defines an indexed family of types (as with GADTs). Indices that do not vary

across the return types of the constructors of a data type can be declared as *parameters*, given a name once in the signature of the type and referenced in each constructor, as in the List and _⊎_ definitions below (from Data.List and Data.Sum). If a parameter is declared without an explicit type, the type is inferred by unification.

```
infixr 5 _::_
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

Indices that vary across different constructors are expressed as part of the type signature of a data declaration, as with the List *A* index of the All type (from Data.List.Relation.Unary.All) defined below. A unary predicate over values of type *A* is encoded as a dependent type *A* → Set, so the All type can be seen as a type of proofs that some predicate *P* holds for every element of a list, in the form of a list of proofs with parallel structure to the list of elements.

```
data All {A} (P : A → Set) : List A → Set where
  [] : All P []
  _::_ : ∀ {a as} → P a → All P as → All P (a :: as)
```

The type Fin *n*, sometimes used as a type of intrinsically-bounded list indices, has a similar structure to ℕ, and can be seen as the type of natural numbers strictly less than *n*.

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc : ∀ {n} → Fin n → Fin (suc n)
```

Although this thesis generally uses a different type for list indices, the Fin type is still useful in some circumstances. The name Fin alludes to an interpretation of the set of values of this type as "the" finite set of size n, since all finite sets of the same size are isomorphic, though not all are equally suited for any particular use case in dependently-typed programming. For a mostly trivial example, Bool could be defined as Fin 2, but the data definition of Bool in Section 2.2 is much more immediately clear to a reader and a little more convenient in pattern-matching.

2.7 Propositional equality

Two terms of the same type are considered *definitionally equal* in a context if they are equal up to $\alpha\beta\eta$-equality in that context. The _≡_ type (from Relation.Binary.PropositionalEquality) encodes *propositional equality*, a weaker notion: two types can be shown to be propositionally equal when they are definitionally equal under any consistent instantiation of their free variables.

```
infix 2 _≡_
data _≡_ {A : Set} (a : A) : A → Set where
  refl : a ≡ a
```

In base Agda, there is always at most one element of the type $a \equiv b$ for any a and b, namely refl. A proof term of type $a \equiv b$ can be given by refl exactly when a and b are definitionally equal in the context of the proof case being defined. The semantics of pattern-matching on a term of type $a \equiv b$ can be subtle, and depend on an attempt to unify a with b; pattern-matching on refl is used in some of the same situations as mechanisms in Agda and other languages to "rewrite" with equality proofs, but is a distinctly different operation with some different usage patterns.

At a high level, the unification engine "knows" little other than the definitional equality rules and that datatype constructors are injective, and its behavior can sometimes be surprising. The higher-order unification algorithm that Agda depends on is only semi-decidable, so there are three possible scenarios when pattern-matching on a value of type $a \equiv b$: there is a consistent unifying substitution that makes a and b definitionally equal ("yes"), they are definitionally inequal ("no"), or the unification engine cannot decide ("maybe").

- In the "yes" case (a and b can be successfully unified), the pattern refl is valid and causes the unification engine to apply the unifying substitution in the context of the case containing the pattern. The expressions a and b are then seen by Agda as definitionally equal. For example, in the definition of cong below, pattern-matching on the second argument unifies x with y, making $f\ x$ definitionally equal to $f\ y$. (The syntax $\forall\ \{A\ B\} \to \ldots$ is syntactic sugar for $\{A : \_\} \to \{B : \_\} \to \ldots$, leaving the types of the declared arguments implicit.)

  ```
  cong : ∀ {A B} (f : A → B) {x y} → x ≡ y → f x ≡ f y
  cong f refl = refl
  ```

  The cong function is included in the same standard library module as the _≡_ type, along with several other derived combinators for reasoning with propositional equality. Its implementation works specifically because the x and y arguments to cong can be unified; the more explicit equivalent implementation below illustrates this with a *dotted pattern*, which asserts that the value of the third argument is constrained by unification to be identical to the value of the second argument.

  ```
  cong′ : ∀ {A B} (f : A → B) x y → x ≡ y → f x ≡ f y
  cong′ f x .x refl = refl
  ```

Since datatype constructors are definitionally injective in unification, pattern-matching successfully against refl can also be used to give a witness of this property phrased in terms of propositional equality. This generally has to be defined for each constructor manually, as in suc-injective below (defined in Data.Nat.Properties).

```
suc-injective : ∀ {x y} → suc x ≡ suc y → x ≡ y
suc-injective refl = refl
```

- In the "no" case (*a* and *b* are definitionally inequal terms), the pattern match is shown to be unsatisfiable and the special *empty pattern* () (with a different meaning than the same syntax in Haskell) is used to signify that there are no possible values of the type. The empty pattern indicates that the case being defined is impossible in a closed context and therefore doesn't need a right-hand side, since it will never occur at runtime. For example, this can be used to show that 0 ≡ 1 implies anything, since different constructors (in this case zero and suc) are definitionally disjoint.

```
0≢1 : {A : Set} → zero ≡ suc zero → A
0≢1 ()
```

- In the "maybe" case (*a* and *b* cannot be unified but also are not definitionally inequal), the Agda typechecker throws an error. For example, when giving a definition for plus-lem, attempting to pattern match immediately on the first explicit argument gives an error saying the two sides cannot be unified: the + operator is not a constructor and therefore not definitionally injective, so attempting to unify

$x + y$ with $y + x$ does not produce any constraints that could be used to uniquely resolve expressions for the x and y arguments.

```
plus-lem : ∀
  {x y z} →
  x + y ≡ y + x →
  (x + y) + z ≡ (y + x) + z
```

These cases require either pattern-matching on the terms that appear within the equality type or applying higher-level combinators to operate over the propositional equality proofs less directly. plus-lem can be defined as cong (_+ z).

In general, definitional equality is almost always more convenient in Agda than propositional equality, since it requires no manual proof effort to make use of. This often leads to a style of proof where propositional equality is used sparingly, preferring inductive data types that encode invariants more directly when feasible and matching against refl whenever possible to avoid explicitly reasoning with propositional equality proofs.

2.8 Record types

Record types in Agda may have parameters, but no other indices. A record type has an optional constructor name and zero or more named fields, whose types may depend on the values of fields above them. The derived constructor has arity equal to the number of fields and takes arguments left to right corresponding to the fields top to bottom; field names can be enclosed in curly braces to make them implicit in the constructor type. A record type definition may also optionally contain arbitrary definitions after the field declarations, which may reference the fields by name and are brought into scope when the record's module is opened (discussed in the section on modules below).

An important record type from the standard library is the *dependent pair* or "sigma" type (from Data.Product), which is used to encode existentially quantified propositions. $\exists$ and $\exists_2$ are type synonyms for Σ with the type inferred for respectively the first component and the first two components, and $\_\times\_$ is the type of ordinary non-dependent pairs where the second component's type does not depend on the first's value. (The definitions of $\exists_2$ and $\_\times\_$ depend on the right-associativity of the λ operator: the definitions are read as though a left parenthesis was present immediately to the left of each lambda with corresponding right parentheses at the end of the line.)

```
record Σ A (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁

∃ : ∀ {A} → (A → Set) → Set
∃ = Σ _

∃₂ : ∀ {A} {B : A → Set} → (∀ a → B a → Set) → Set
∃₂ C = ∃ λ a → ∃ λ b → C a b

_×_ : Set → Set → Set
A × B = Σ A λ _ → B
```

Record types enjoy an η-expansion rule similar to functions, so that, for example, (proj₁ x , proj₂ x) is definitionally equal to x for any x of a pair type.

Often, the first component of a term of a dependent pair type can be uniquely inferred by unification if the second component is known; this can be expressed as _ , x or

with the standard library operator -,_ as -, *x* with some suitable second component *x*. Since the comma character is not lexically distinguished from any other identifier characters, pair expressions in Agda generally require spaces both before *and* after a comma so that the comma isn't read as part of an identifier in the first or second component expression. The _,′_ operator constructs specifically non-dependent pairs; while the _,_ constructor already serves this purpose, indicating that an expression is a non-dependent pair can sometimes be an aid to type inference.

Another syntax for giving terms of record types is with the record keyword, giving each field a definition in an arbitrary order.

```
ex : ℕ × ℕ
ex = record { proj₂ = 1 ; proj₁ = 0 }
```

The unit type is defined in Data.Unit as a trivial record type, with no fields and exactly one distinct member of the type called tt.

```
record ⊤ : Set where
  constructor tt
```

There is an advantage in Agda to defining ⊤ as a trivial record instead of a trivial inductive data type: η-expansion makes every expression of type ⊤ *definitionally* equal to tt, whereas all expressions of a unit type declared inductively can be shown to be propositionally equal by pattern-matching but not all are definitionally equal.

2.9 Negation

From Relation.Nullary.Negation, the negation of a type *A* is the type of functions mapping every element of *A* to an element of the empty type ⊥ (from Data.Empty), which has no elements.

```
data ⊥ : Set where

infix 3 ¬_
¬_ : Set → Set
¬ A = A → ⊥
```

The negation of the propositional equality type is defined in Relation.Binary.PropositionalEquality.

```
_≢_ : ∀ {A} → A → A → Set
a ≢ b = ¬ (a ≡ b)
```

Proofs of negated types are usually created by pattern-matching against unsatisfiable patterns with the empty pattern; for example, a slightly different encoding of the proof of 0 ≢ 1 from before can be written as a negated type.

```
absurd′ : zero ≢ suc zero
absurd′ ()
```

Similarly, the eliminator for the empty type (*ex falso quodlibet* or the principle of explosion) is defined in Data.Empty as a function mapping values of the empty type to values of any other type, using the empty pattern to assert that the pattern match on the input argument is unsatisfiable. The empty pattern can be used for any definitionally empty type, not just propositional equalities. An indexed type is definitionally empty at some particular indices when the return types of all of its constructors fail to unify with the type at those indices, which is trivially true of a type with no constructors (and in this case incidentally no indices).

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

The contradiction combinator combines ⊥-elim and the principle of explosion to produce a value of any arbitrary type when both a type and its negation are inhabited in the same context. (The name ¬*a* is just an identifier with no special semantic significance - the ¬ character is a normal identifier character.)

```
contradiction : ∀ {A B} → A → ¬ A → B
contradiction a ¬a = ⊥-elim (¬a a)
```

2.10 List membership

The inductive type of membership proofs for some element in a list is particularly relevant to this thesis, serving as a type of list indices guaranteed to be within the bounds of a specified list. The here constructor says that the head of a nonempty list is a member of the list, and there says that any member of the tail of a nonempty list is a member of the whole list.

```
infix 4 _∈_
data _∈_ {A} (a : A) : List A → Set where
  here : ∀ {b as} → a ≡ b → a ∈ b :: as
  there : ∀ {b as} → a ∈ as → a ∈ b :: as
```

The standard library actually defines _∈_ in Data.List.Membership.Setoid as a specialization of the Any type (from Data.List.Relation.Unary.Any); the _∈_ type is also parameterized over an arbitrary equivalence relation, but this thesis only uses the variant specialized to propositional equality in Data.List.Membership.Propositional, which has the same name and constructors as the definition shown above. In this thesis here is always used with the argument refl, making it effectively a nullary constructor relying on definitional equality.

With this usage, the type $a \in as$ can be seen as a type of unary indices in the list as that contain the element a, and the type $\exists\ (\_\in as)$ as the type of all unary indices in as. The indexed function, provided in Code.Util in the development accompanying this thesis, produces an output list with each element of the input list paired with its own index, similar to the Haskell function $\lambda\ as \rightarrow$ zip as [0 .. length as]. (The List. and Σ. prefixes refer to the Data.List and Data.Product modules respectively, and map_2 is the function to map a function over the second component of a pair.)

```
indexed : ∀ {A} (as : List A) → List (∃ (_∈ as))
indexed [] = []
indexed (a :: as) = (-, here refl) :: List.map (Σ.map₂ there) (indexed as)
```

2.11 Decision procedures

Agda implements a constructive logic by default, without the law of the excluded middle as a fully general axiom or theorem; in the setting of type theory, this means that there is no general decision procedure for inhabitance of types. A type is defined to be *decidable* when it is known to be either inhabited or uninhabited, as witnessed by either an element of the type or a proof that there are no elements. This is represented by the Dec type from Relation.Nullary, which also includes the True data type and conversion functions defined below.

```
data Dec A : Set where
  yes : A → Dec A
  no : ¬ A → Dec A
```

A decidable proposition can be "squashed" into one with exactly zero or one members with the True type. For some A : Set and $A?$: Dec A, an element of type True $A?$

witnesses that A is inhabited and therefore that all elements of the type Dec A must be constructed with yes.

```
True : ∀ {A} → Dec A → Set
True (yes _) = ⊤
True (no _) = ⊥
```

A and True $A?$ are logically equivalent in the sense of bidirectional implication, but they are not isomorphic in Agda's proof-relevant logic; some element toWitness t : A can be extracted from a value t : True $A?$, but for any nontrivial A there are elements of Dec A that are not propositionally equal to yes (toWitness t), and therefore toWitness ∘ fromWitness is not an identity function in general.[1]

```
fromWitness : ∀ {A} {A? : Dec A} → A → True A?
fromWitness {A? = yes _} a = tt
fromWitness {A? = no ¬a} a = contradiction a ¬a

toWitness : ∀ {A} {A? : Dec A} → True A? → A
toWitness {A? = yes a} _ = a
toWitness {A? = no _} ()
```

Importantly, propositional equality is decidable over many types. The definition for natEq? below illustrates a common pattern in decidable equality proofs: a case for each possible pair of constructors is written out, and the cases with inequal pairs are given the result no λ (), asserting that the goal equality type in each case is definitionally uninhabited. The cases with matching constructor patterns match against the result of a recursive call over each parallel pair of their arguments; in the case that all of

[1] True $A?$ is properly isomorphic to the type ∃ λ (a : A) → $A?$ ≡ yes a.

them are equal, the two input arguments are equal, and in the case that any parallel pair of constructor arguments is not equal, the input arguments are not equal. The pattern argument $\neg\equiv$ has type $\neg\ (m \equiv n)$ as the negative result of the recursive equality test, and the term λ where refl $\rightarrow \neg\equiv$ refl here has type $\neg\ (\text{suc}\ m \equiv \text{suc}\ n)$, exploiting the definitional injectivity of constructors to reduce the suc $m \equiv$ suc n argument to a definitional equality between m and n so that refl is a valid argument to $\neg\equiv$.

```
natEq? : (m n : ℕ) → Dec (m ≡ n)
natEq? zero zero = yes refl
natEq? (suc m) (suc n) =
  case natEq? m n of λ where
    (yes refl) → yes refl
    (no ¬≡) → no λ where refl → ¬≡ refl
natEq? zero (suc n) = no λ ()
natEq? (suc m) zero = no λ ()
```

Unfortunately, this quadratic explosion of cases is sometimes unavoidable. Boolean equality tests over sum types traditionally rely on catch-all cases returning false to combine all cases with disjoint constructors into one case, but the term no λ () is only well-typed as a right-hand side for a case in a decidable equality proof when the case's argument patterns are definitionally disjoint, which isn't true in a catch-all clause in Agda even if previous cases in the definition already handle all non-disjoint pairs of constructors. (Agda provides a semantic guarantee of top-to-bottom evaluation of cases, but there's no straightforward way to obtain an internal proof of that property within a particular definition.)

2.12 Modules

Modules in Agda programs can be parameterized over an arbitrary sequence of typed identifiers, and then brought into scope with the open keyword with any (or none) of the module parameters instantiated. Any parameters left uninstantiated when a module is opened become arguments to all of the terms that the open brought into scope. For example, the AddMod module is parameterized over a natural number; when it's opened with the parameters uninstantiated in the Ex_1 module, the function add takes two arguments, and when it's opened with the parameter instantiated with 1 in the Ex_2 module, add takes one argument. An open expression can be accompanied by the keyword public to re-export all of the definitions being imported, and the using/hiding/renaming keywords can be used to avoid name clashes and avoid littering a local scope.

```
module AddMod (n : ℕ) where
  add : ℕ → ℕ
  add = n +_

module Ex₁ where
  open AddMod
  -- add : ℕ → ℕ → ℕ
  -- add n = n +_

module Ex₂ where
  open AddMod 1
  -- add : ℕ → ℕ
  -- add = 1 +_
```

Definitions in modules are also always accessible unparameterized by their fully qualified names as long as the module itself is in scope, so the expression AddMod.add

always takes two arguments. Like data type indices and function arguments, module parameters can be explicit or implicit, with implicit parameters surrounded by curly braces.

Modules can also be defined directly in terms of other modules, as in the examples Ex_3 and Ex_4. This syntax can be combined with the open keyword for significant flexibility in specializing modules, particularly with the form used in Ex_4 where some parameters of the module being referenced are instantiated with nontrivial terms constructed over parameters to the module being defined.

```
open module Ex₃ = AddMod 1 renaming (add to add1)
-- add1 :  ℕ → ℕ
-- add1 = 1 +_

open module Ex₄ (n : ℕ) = AddMod (suc n) renaming (add to addsuc)
-- addsuc :  ℕ → ℕ → ℕ
-- addsuc = λ n → suc n +_
```

Data types and record types have associated modules that are created automatically; the module associated with a data type has no parameters, contains only its constructors, and is automatically opened with all its definitions exported publicly, while the module associated with a record type is parameterized over the record type itself, contains the record's constructor and fields and other definitions, and only automatically exports the constructor. The standard library exports $proj_1$ and $proj_2$ explicitly, but otherwise they would only be accessible qualified or through an open Σ statement.

A special form of the module declaration syntax allows for the definition of an anonymous module, which gets opened into the public scope immediately. This can be used to abstract a block of code over some parameters in order to avoid repeating the parameter

types in each definition. The comment at the end of this anonymous example module definition is its effective definition in the scope outside the anonymous module.

```
module _ (n : ℕ) where
  flip-add : ℕ → ℕ
  flip-add m = m + n
-- flip-add :  ℕ → ℕ → ℕ
-- flip-add = λ n m → m + n
```

The variable keyword provides a different form of abstraction over common parameters: in the scope after a variable definition, whenever one of the defined names is referenced in a type where the name is otherwise unbound, the type is automatically defined to take an implicit argument with that name. For example, the dependent function composition operator can be defined as follows, taking *A*, *B*, and *C* as implicit arguments.

```
variable
  A : Set
  B : A → Set
  C : ∀ a → B a → Set

_∘_ : (∀ {a} (b : B a) → C a b) → (f : ∀ a → B a) → ∀ a → C a (f a)
(g ∘ f) x = g (f x)
```

Unlike with anonymous modules, only the names from a variable definition that are used in a particular type are included as implicit arguments in the type.

2.13 Instance arguments

In addition to explicit arguments given by the user and implicit arguments uniquely inferred by unification, Agda supports a third kind of arguments called *instance argu-*

ments [7], which the typechecker attempts to resolve uniquely through a constrained proof search. The *instance context* contains all instance arguments in scope along with all definitions in scope that have no explicit arguments and are declared within an instance block, and when the typechecker encounters a term with an unresolved instance argument, it searches through the instance context to try to construct a unique term for the type. Since this is not guaranteed to terminate in general, the search has a fixed cutoff depth that can be configured with a command-line option.

Instance arguments are declared and used with the same syntax as implicit arguments, but with double curly braces instead of single braces.

```
auto₁ : {A : Set} {{a : A}} → A
auto₁ {{a}} = a

auto₂ : {A : Set} {{a : A}} → A
auto₂ {{a = x}} = x
```

Instance arguments are most useful in conjunction with a special form of the open syntax that opens a module with all of its parameters resolved by instance search at each use site of each of its definitions, allowing record types and the modules they generate to serve some of the purposes of typeclasses in other languages. The types in this section are defined in `Util.lagda` in the development accompanying this thesis.

```
record Show (A : Set) : Set where
  field
    show : A → String

  print = putStrLn ∘ show

open Show {{...}}
```

In the scope after this open expression, show is in scope with the type ∀ {*A*} {{_ : Show *A*}} → *A* → String. With the instance definition below in scope, for example, this can be automatically specialized by instance search at a use site to show : Bool → String.

```
instance
  show-bool : Show Bool
  show {{show-bool}} = Data.Bool.Show.show
```

An anonymous definition in an instance block can be given by using a single underscore _ as the name; this does not add any new names into scope, but adds the instance into the *local* instance context anonymously. This can be useful for introducing instances into scope in a where or let clause, in order to seed the local instance context without affecting the global instance context. An instance defined in this way within a module can't be exported from the module, however, so library code defining instances intended to be used by client code needs to give names to the instances, even though the names themselves are often immaterial since they get resolved by instance search instead of being written explicitly in the source code.

The DecEq type classifies types with decidable equality. A function to decide list

membership over a list of elements of a type with decidable equality is exported into the DecEq module scope by opening a standard library module specialized with the equality decision operator, making the _∈?_ operator also available through open DecEq {{...}}.

```
record DecEq A : Set where
  constructor decEq
  infix 4 _≟_
  field
    _≟_ : (a b : A) → Dec (a ≡ b)

  open module ListMembership =
    Data.List.Membership.DecPropositional _≟_ using (_∈?_) public

open DecEq {{...}}
```

The Propositional type classifies *propositional* types in the sense of "mere propositions" or (-1)-truncated types in homotopy type theory; the type isPropositional *A* is defined in Relation.Binary.PropositionalEquality as ∀ (*x y* : *A*) → *x* ≡ *y*, the type of proofs that every element of *A* is propositionally equal, witnessing that *A* has exactly zero or one distinct elements up to propositional equality. This type can be particularly useful when working with finite types, as any propositional type whose inhabitance is decidable can be shown to be finite. The Propositional type can also be useful as a stronger variant of the DecEq type, for types where the result of an equality decision procedure is always yes.

```
record Propositional (A : Set) : Set where
  constructor propositional
  infix 4 _≟*_
```

```
    field
      _≟_ : isPropositional A

  open Propositional {{...}}
```

The empty and unit types are trivially propositional, and by virtue of these the True type is propositional. This thesis uses Agda with the K axiom enabled (as by default), so the equality type is also propositional. (The uip proof comes from Axiom.UniquenessOfIdentityProofs.WithK).

```
  instance
    ⊥-propositional : Propositional ⊥
    ⊥-propositional = propositional λ ()

    ⊤-propositional : Propositional ⊤
    ⊤-propositional = propositional λ _ _ → refl

    true-propositional : ∀ {A} {A? : Dec A} → Propositional (True A?)
    true-propositional {A? = yes _} = ⊤-propositional
    true-propositional {A? = no _} = ⊥-propositional

    ≡-propositional : ∀ {A} {a b : A} → Propositional (a ≡ b)
    ≡-propositional = propositional uip
```

The Singleton type is similar in character to the DecEq and Propositional, although it isn't used with the open {...} feature in this thesis. A type is defined to be a singleton if it has exactly one element up to propositional equality; a problem has a unique solution if the type of its valid solutions is a singleton type.

```
  record Singleton (A : Set) : Set where
    constructor singleton
```

```
    field
      point : A
      {{unique}} : Propositional A
```

Finally, Agda provides the built-in types Number, Negative, and IsString in the Agda.Builtin.FromNat, Agda.Builtin.FromNeg, and Agda.Builtin.FromString modules, allowing the constant literal syntax to be overloaded for positive integer literals, negative integer literals, and strings, respectively.

Chapter 3 Listable types

The code associated with this chapter is found in the Code.Listable module.

3.1 Representation

As the list membership type can be seen as a type of unary list indices, the membership field in the definition of Listable below can be read computationally as an inverse lookup function, mapping elements to their indices in the elements list. This function is total (as all Agda functions are), which establishes that *A* is finite. The function Any.index from the standard library converts a list membership proof to a bounded numeric index, so Listable.index identifies each element of the type *A* with a unique index in elements.

```
record Listable A : Set where
  constructor finite
  field
    elements : List A
    membership : ∀ a → a ∈ elements

  size : ℕ
  size = List.length elements

  index : A → Fin size
  index = Any.index ∘ membership
```

3.2 Properties

The definitions in this section are defined within the Listable record definition, so they have access to the elements and membership fields directly.[1] The Listable module defined by the record type is usually opened with open Listable, without giving a specific record value as a parameter, so that these definitions all take a Listable argument and the references to elements and membership are bound to the corresponding fields to that argument.

3.2.1 Decidability

The Listable type can be seen as a generalization of the Dec type, in the sense that a value of type Dec *A* witnesses whether there are *zero or nonzero* elements of *A* and a value of type Listable *A* witnesses *how many* elements of *A* there are. By this reasoning, a value of type Dec *A* is derivable from a value of type Listable *A* by checking whether the elements list is empty or nonempty.

```
dec : Dec A
dec with elements | membership
... | [] | p = no λ a → case p a of λ ()
... | a :: _ | _ = yes a
```

The syntax with elements | membership is a more general form of pattern matching than case_of_, used in this definition to match against elements and membership at the same time so that in the [] case the *p* argument has type $\forall\ x \to x \in []$ rather than $\forall\ x \to x \in$ elements. Typechecking definitions using the with keyword can require the

[1] Due to a peculiarity in the way the Agda compiler outputs LaTeX code, these definitions are not printed in this thesis with leading indentation, but in the actual Agda code they are all indented to the same level as the size and index definitions.

typechecker to do more work than typechecking functionally equivalent definitions using case_of_ and pattern-matching lambdas, and the with syntax can only be used on the left-hand side of a definition, so case_of_ is usually preferred when it suffices.

3.2.2 Decidable equality

Equality is decidable over the type $\exists\ (\_\in as)$ for any *as*, following from the view of the _∈_ type as a type of unary indices - the implementation is similar to the equality decision procedure over the $\mathbb{N}$ type.

```
∃∈-≟ : ∀ {A} {as : List A} (i j : ∃ (_∈ as)) → Dec (i ≡ j)
∃∈-≟ (_ , here refl) (_ , here refl) = yes refl
∃∈-≟ (_ , there i) (_ , there j) =
  case ∃∈-≟ (-, i) (-, j) of λ where
    (no ¬≡) → no λ where refl → ¬≡ refl
    (yes refl) → yes refl
∃∈-≟ (_ , here refl) (_ , there j) = no λ ()
∃∈-≟ (_ , there i) (_ , here refl) = no λ ()
```

A consequence of this is a decision procedure for equality over any listable type, following from the view of membership as an inverse lookup function.

```
instance
  finite-decEq : DecEq A
  finite-decEq =
    record
      { _≟_ =
          λ a b →
```

```
    case ∃∈-≟ (-, membership a) (-, membership b) of λ where
      (yes refl) → yes refl
      (no ¬≡) → no λ where refl → ¬≡ refl
}
```

3.2.3 Recursion over listable types

A general pattern of recursion over listable types is to maintain a list of "seen" elements and a list of "unseen" elements, along with a proof that some informative decidable procedure holds for every seen element. Functions defined in this style can be seen as recursively "discovering" proofs of the relevant predicate until they have enough information to prove some proposition.

```
Rec : (A → List A → Set) → Set → Set
Rec P B = ∀ xs ys → (∀ a → (a ∈ xs × P a xs) ⊎ (a ∈ ys)) → B

rec : ∀ {B : Set} {P : A → List A → Set} → Rec P B → B
rec r = r [] elements (inj₂ ∘ membership)
```

Most importantly, this style of recursive function definition can be used to implement decision procedures for the existential and universal quantifiers over predicates with listable domains. The implementations of ∃-rec and ∀-rec are respectively upgraded versions of the standard list functions any and all of type (A → Bool) → List A → Bool.

```
module _ {P : A → Set} (P? : ∀ a → Dec (P a)) where
  ∃-rec : Rec (λ a _ → ¬ P a) (Dec (∃ P))
  ∀-rec : Rec (λ a _ → P a) (Dec (∀ a → P a))
```

```
∃? : Dec (∃ P)
∃? = rec ∃-rec

∀? : Dec (∀ x → P x)
∀? = rec ∀-rec
```

These decision procedures along with finite-decEq can be used to build decision procedures for the Singleton and Propositional types. The singleton? function is used in Part II to check whether the typing derivation of a term is unique.

```
singleton? : Dec (Singleton A)
propositional? : Dec (Propositional A)
```

3.3 Functions and instances

The definitions in this section are outside the Listable record module, so they take explicit Listable arguments as needed.

3.3.1 Listable algebraic data types

The empty and unit types are trivially listable. A superscript f is used to distinguish the definitions that construct Listable values, as a mnemonic for the interpretation of Listable as a type of proofs of "**f**initeness". As seen in various definitions later in this section, it's often a benefit to readability to be able to express Listable values with a syntax that closely mirrors the types they construct proofs of listability for.

```
⊥ᶠ : Listable ⊥
⊥ᶠ = finite [] λ ()
```

```
⊤ᶠ : Listable ⊤
⊤ᶠ = finite [ tt ] λ where tt → here refl
```

The sum and dependent pair types are both listable whenever their constituent types are, and the non-dependent pair type is shown to be listable by a specialization of Σ^f.

3.3.2 Mapping

The Listable type is often naturally seen as a container type, but it's not a functor in the sense of Haskell's Functor typeclass or Agda's Category.Functor.RawFunctor type (an endofunctor over the category of types and unrestricted function arrows): only surjective functions can be mapped over a Listable value in general in order to produce another Listable value. The code in this thesis mostly uses an infix version of map^f with the arguments flipped, named after the _<&>_ operator in Category.Functor.RawFunctor.

```
mapᶠ : ∀ {A B} → (A ↠ B) → Listable A → Listable B

infix 0 _<&ᶠ>_
_<&ᶠ>_ = flip mapᶠ
```

The type _↠_ is defined in Function.Surjection and represents functions that are surjective up to propositional equality. The surjection function from the same module constructs a term of type $A \twoheadrightarrow B$ given a function from A to B and a function left inverse to it.

```
surjection : ∀ {A B} (f : A → B) (g : B → A) → (∀ x → g (f x) ≡ x) → A ↠ B
```

The first argument to surjection is the function that gets mapped over the elements list; when a value constructed with surjection is used as an argument to map^f, the

second and third arguments to surjection are only used as proof artifacts witnessing the surjectivity of the first argument, so notably the g function is never called during an evaluation of map^f at runtime.

The map^f function is most often useful when given a function between some inductively-defined type and another straightforwardly structurally isomorphic type already known to be listable. For example, the type Maybe A is isomorphic to $\top \uplus A$, which is known to be listable when A is listable. The first argument in the definition of Maybe^f below is of type $\top \uplus A \to \mathsf{Maybe}\ A$, and the second argument is its inverse; the third argument is the witness to the surjectivity of the first function, showing that both cases reduce to definitional equality.

```
Maybeᶠ : ∀ {A} → Listable A → Listable (Maybe A)
Maybeᶠ Aᶠ =
  ⊤ᶠ ⊎ᶠ Aᶠ <&ᶠ>
    surjection
      (λ where
        (inj₁ _) → nothing
        (inj₂ a) → just a)
      (λ where
        nothing → inj₁ tt
        (just a) → inj₂ a)
      (λ where
        nothing → refl
        (just _) → refl)
```

This illustrates a common tension in the tradeoff between bespoke inductive data types with specific declarative constructor names and more reusable polymorphic types,

also at play in the decision not to define Bool as Fin 2. Generic programming features like GHC's `Generic` typeclass enable library code to traverse and manipulate values of unknown user-defined types by offering a procedure to derive structurally isomorphic types constructed with a known set of primitives like the sum and sigma types; along with Agda's metaprogramming features, this might make it possible to automate away some or all of the proofs that use map^f, which would significantly simplify the source code of this development.

3.3.3 Filtering

Σ^f can be seen as a kind of filtering function: the set of all of the proj_1 components in the elements list of the output type is a subset of the set of elements in the elements list of the input. Depending on the predicate used as the second argument, however, there may in general be any number of values in the output elements list for any given element of the input elements list, making Σ^f unsuitable as a general counterpart to the traditional filtering function over lists.

Since True $A?$ is listable for any $A?$, an appropriate type for the filter^f function over listable types in general is Listable (∃ (True ∘ $P?$)), for some decidable unary predicate argument $P?$. In the special case where P is propositional at all indices, True ($P?$ a) is isomorphic to P a for any a, so this filtered output is guaranteed to contain all elements of P at all indices as witnessed by the surjection given in the definition of filterProp^f.

```
module _ {A} {P : A → Set} (P? : ∀ a → Dec (P a)) where
  filterᶠ : Listable A → Listable (∃ (True ∘ P?))
  filterᶠ Aᶠ = Σᶠ Aᶠ (Trueᶠ ∘ P?)

  filterPropᶠ : {{_ : ∀ {a} → Propositional (P a)}} → Listable A → Listable (∃ P)
```

```
filterProp^f A^f =
  filter^f A^f <&^f>
    surjection (Σ.map₂ toWitness) (Σ.map₂ fromWitness) λ where
      (a , p) → cong (a ,_) (_ ≟ p)
```

3.3.4 Maximal values

The definitions in this subsection appear in the Ordered module, which the languages defined later in this thesis open specialized to their respective specificity orderings. The type IsDecStrictPartialOrder from Relation.Binary witnesses that a given ordering relation is a decidable strict partial order over a given equivalence relation.

```
module Ordered
  {A} {_≈_ : A → A → Set} {_<_ : A → A → Set}
  (<-dpo : IsDecStrictPartialOrder _≈_ _<_)
  (A^f : Listable A)
  where
  open IsDecStrictPartialOrder <-dpo
  open Listable A^f
```

A listable set ordered by a decidable strict partial order has a listable subset of maximal values, in the sense of values that are not strictly smaller than any other values of the type.

```
  Maximal : A → Set
  Maximal a = ∀ b → ¬ (a < b)
```

```
maximal? : ∀ a → Dec (Maximal a)
maximal? a = ∀? λ b → ¬? (a <? b)

∃maximal^f : Listable (∃ (True ∘ maximal?))
∃maximal^f = filter^f maximal? A^f
```

3.3.5 Listable standard library types

All empty types are trivially listable.

```
¬^f_ : ∀ {A} → ¬ A → Listable A
¬^f ¬a = finite [] λ a → contradiction a ¬a
```

All singleton types are listable, with exactly one element.

```
Singleton^f : ∀ {A} → Singleton A → Listable A
Singleton^f s = finite [ point s ] λ x → here (x ≟ point s)
```

Propositional types are not necessarily listable in general, but any decidable propositional type is listable, with either zero or one elements.

```
decProp^f : ∀ {A} {{_ : Propositional A}} → Dec A → Listable A
decProp^f (yes a) = finite [ a ] λ b → here (b ≟ a)
decProp^f (no ¬a) = finite [] λ a → contradiction a ¬a
```

As a consequence, the propositional equality type for any type with decidable equality is listable.

```
_≡^f_ : ∀ {A} {{_ : DecEq A}} (a b : A) → Listable (a ≡ b)
a ≡^f b = decProp^f (a ≟ b)
```

For any unary predicate *P* that's listable at all indices and any list *as* of elements in its domain, the type All *P as* is listable.

```
All^f : ∀ {A} {P : A → Set} {as : List A} → (∀ a → Listable (P a)) → Listable (All P as)
```

Similarly, the Pointwise type from Data.List.Relation.Binary.Pointwise is the variant of All for binary instead of unary predicates, and is listable at all indices whenever the predicate type *P* is listable at all indices.

```
Pointwise^f : ∀
  {A B} {P : A → B → Set} {as : List A} {bs : List B} →
  (∀ a b → Listable (P a b)) →
  Listable (Pointwise P as bs)
```

Chapter 4 Finite graph search

The main challenges in formalizing a graph search decision procedure in Agda are establishing termination and completeness. Traditional graph search algorithms are usually implemented with a collection of vertices that grows and shrinks dynamically during the execution of the algorithm, and termination is shown by an argument that the collection will eventually reach an empty state. Agda enforces termination by allowing only structural recursion, where at least one argument to every recursive call must be a strict syntactic subpattern of the corresponding argument pattern in the case being defined; the termination arguments for the traditional algorithms are not easily phrased in these terms. A recognition procedure for the existence of a path between given vertices can be implemented structurally recursively over the set of all vertices in the graph, monotonically removing elements from the set until a path is found or the set is empty, but the fact that the failure case implies that no path at all exists is not proven by construction and is nontrivial to prove separately.

One conceptually simple technique for formalizing a complete search algorithm by construction in this setting is to show that the search space is finite, which allows the size of the space to serve as a structurally decreasing termination measure and yields a proof of completeness by construction. The set of all paths in a graph is infinite if the graph contains any cycles, but the search space for a finite graph search is always finite: it needs only to consider acyclic paths, since the ability to "cut loops" out of a path means the impossibility of an acyclic path between two nodes implies the impossibility of any path at all between them.

This can be formalized in Agda with the Finite type, representing a finite graph as a Listable type of vertices along with a binary edge relation indexed over vertices which is

listable at all indices. Paths in the graph correspond to elements of the reflexive transitive closure of the edge relation. A search procedure for paths out of a given source to a destination matching some given decidable predicate can be expressed as a decision procedure that iterates over the list of acyclic paths; if no acyclic path is found, a function that reduces arbitrary paths to acyclic paths serves as witness that no path of any length exists.

The code associated with this chapter is found in the Code.Graph, Code.Graph.Cut, Code.Graph.Finite, and Code.Graph.Search modules.

4.1 Finite graphs

```
module Code.Graph where
```

This variable block is declared private to the Code.Graph module so that it doesn't conflict with other definitions in later modules.

```
private
  variable
    A : Set
    a b c : A
    m n l : ℕ
```

Finite graphs are represented by the FiniteGraph type. The Edge^f field is an adjacency list representation of a graph, with each element in its elements list containing a pair of vertices and an edge between them.

```
record FiniteGraph : Set where
  constructor finiteGraph
```

```
  field
    {Vertex} : Set
    {Edge} : Vertex → Vertex → Set
    Vertex^f : Listable Vertex
    Edge^f : Listable (∃₂ Edge)
```

The edgeFrom^f function is defined by filtering the adjacency list down to a list of edges that have a given source; edgeTo^f is defined similarly.

```
  edgeFrom^f : ∀ a → Listable (∃ (Edge a))
  edgeFrom^f a =
    filterProp^f (λ where (a′ , _ , _) → a ≟ a′) Edge^f <&^f>
      surjection
        (λ where ((_ , _ , e) , refl) → -, e)
        (λ where (_ , e) → (-, -, e) , refl)
        (λ where (_ , e) → refl)

  edgeTo^f : ∀ b → Listable (∃ λ a → Edge a b)
```

4.2 Reflexive transitive closure types (Star)

The standard library defines the type of the reflexive transitive closure of an arbitrary binary relation in Relation.Binary.Construct.Closure.ReflexiveTransitive with the name Star, which can be seen as a type of paths.

```
  data Star {A} (R : A → A → Set) a : A → Set where
    ε : Star R a a
    _◅_ : ∀ {b c} → R a b → Star R b c → Star R a b
```

Computationally, Star is a list of edges of type R with the usual graph path invariant requiring the sources and destinations of edges at adjacent indices in the list to agree, with ϵ as the empty list and _◁_ as the "cons" operator.

4.3 Path types

The definitions in this section and the next are parameterized over an arbitrary vertex type A and edge type R.

```
module Path (R : A → A → Set) where
```

A type of length-indexed vectors is sometimes useful in intermediate steps when reasoning about List values, in cases where careful reasoning about the lengths of lists is required. Similarly, the Star type is the end goal of path search, but since the termination argument for graph search relies on careful reasoning about the lengths of paths, it's convenient to express some of the logic with the length-indexed Path type.

```
  data Path a : A → ℕ → Set where
    ε : Path a a zero
    _◁_ : R a b → Path b c n → Path a c (suc n)
```

The _◁◁_ operator acts as an append function, witnessing the transitivity of paths. The _▷_ operator appends a single edge to the end of a path, like the traditional "snoc" function on lists; unsnoc splits the last element off of a nonempty path, serving as an inverse to _▷_.

```
  _◁◁_ : Path a b m → Path b c n → Path a c (m + n)
  _▷_ : Path a b n → R b c → Path a c (suc n)
  unsnoc :
```

```
  (p : Path a c (suc n)) →
  ∃ λ b → ∃₂ λ (p′ : Path a b n) (e : R b c) → p ≡ (p′ ▷ e)
```

The types Path< and Path⩽ represent paths that are respectively less than and less than or equal to the length index, useful for tracking decreasing path lengths to establish termination. The convenience function boundedPath accesses the Path component of either of these types.

```
Path< Path⩽ : A → A → ℕ → Set
Path< a b n = ∃ λ m → m < n × Path a b m
Path⩽ a b n = ∃ λ m → m ⩽ n × Path a b m

boundedPath : {P : ℕ → Set} (p : ∃ λ n → P n × Path a b n) → Path a b (proj₁ p)
boundedPath = proj₂ ∘ proj₂
```

4.4 Isomorphism between path representations

The type Star *a b* is isomorphic to ∃ (Path *a b*); the conversion functions back and forth are analogous to the ones that convert between List and Vec (the type of length-indexed lists) in the standard library. The fold function here is defined in Relation.Binary.Construct.Closure.ReflexiveTransitive.

```
starLength : Star R a b → ℕ
starLength = fold _ (const suc) zero

toStar : Path a b n → Star R a b
toStar ε = ε
toStar (e ◁ p) = e ◁ toStar p
```

```
fromStar : (p : Star R a b) → Path a b (starLength p)
fromStar ε = ε
fromStar (e ◁ p) = e ◁ fromStar p
```

One direction of the isomorphism is easy to witness: `toStar ∘ fromStar : Star R a b → Star R a b` is extensionally (pointwise) equal to the identity function by straightforward structural induction on the input argument.

```
toStar-fromStar : (p : Star R a b) → p ≡ toStar (fromStar p)
```

The other direction is a little more awkward; since the return type of `fromStar ∘ toStar` has length index `starLength (toStar p)` instead of n, a witness to the equality of the input and output lengths is required in order to express the equality between the input and output values with the `_≡_` type. An alternative is to phrase the proposition with the heterogeneous equality type `_≅_`, effectively requiring the proof to witness both the equality between types and the equality between terms. Heterogeneous equality is often less convenient to work with than propositional equality: the constructor pattern `refl` is only applicable for heterogeneous equality when both the types and the terms of the two arguments to `_≅_` can be respectively unified, in which case propositional equality is often possible to use directly instead, and the heterogeneous equality combinators often require explicit type annotations to use. Thankfully, `fromStar-toStar` is not directly relevant to most of this development, and is only used once in the Agda source code.

```
fromStar-toStar : (p : Path a b n) → p ≅ fromStar (toStar p)
```

4.5 Cycles

The definitions in this section are parameterized over an arbitrary `FiniteGraph` named g.

```
module Code.Graph.Cut (g : FiniteGraph) where
open FiniteGraph g
open Path Edge
```

4.5.1 Path membership

The type _∈P_ is an inductive predicate similar to _∈_, witnessing the index of the source of an edge in a path. It could be equivalently defined as _∈_ composed with a function that extracts a list of edge sources from a path, but being able to pattern-match against a specialized inductive predicate is convenient in many cases.

```
data _∈P_ x : Path a b n → Set where
  here : {e : Edge x b} {p : Path b c n} → x ∈P e ◁ p
  there : {e : Edge a b} {p : Path b c n} → x ∈P p → x ∈P e ◁ p
```

Edge sources can also be accessed by numeric index; ∈P-lookup can be seen as converting a numeric index to a more strongly-typed index in the type _∈P_.

```
lookup : Path a b n → Fin n → Vertex
∈P-lookup : {p : Path a b n} (i : Fin n) → lookup p i ∈P p
```

4.5.2 Cyclic paths

A path is defined to be cyclic if it includes at least two distinct edges with the same source or if it includes an edge with the last vertex in the path as a source. In the here case, the source of the head edge is found as the source of an edge somewhere in the tail; in the there case, the tail of the path contains a cycle; and in the end case, the last vertex in the path is found as the source of an edge somewhere in the path.

```
data Cyclic : Path a b n → Set where
  here : {e : Edge a b} {p : Path b c n} → a ∈P p → Cyclic (e ◁ p)
  there : {e : Edge a b} {p : Path b c n} → Cyclic p → Cyclic (e ◁ p)
  end : {p : Path a b n} → b ∈P p → Cyclic p
```

A proof of Cyclic *p* encodes an ordered pair of vertex indices in a path, delimiting a cycle in the path. Both _∈P_ and Cyclic are decidable by structural induction, assuming equality between vertices is decidable.

```
cyclic? : (p : Path a b n) → Dec (Cyclic p)
_∈P?_ : ∀ x (p : Path a b n) → Dec (x ∈P p)
```

A path is defined to be acyclic if it is not cyclic. The function ¬? from Relation.Nullary.Negation decides the negation of a decidable predicate.

```
Acyclic : Path a b n → Set
Acyclic p = ¬ (Cyclic p)

acyclic? : (p : Path a b n) → Dec (Acyclic p)
acyclic? p = ¬? (cyclic? p)
```

4.5.3 Locating cycles

The type Segmented *a b n* represents a path of length *n*, broken into a prefix, a cycle of nonzero length, and a suffix. Even though it only has one constructor, it's defined as an inductive data type instead of a record type because the third index of the return type of the constructor is not a variable and therefore can't be a parameter to the type.

```
data Segmented (a b : Vertex) : ℕ → Set where
  _◄_◄_ : ∀ {x} →
```

```
    Path a x m → Path x x (suc n) → Path x b l →
    Segmented a b (m + suc n + l)
```

Given an index in a path, the path can be broken into a prefix strictly before the index and a suffix including the index and everything after it.

```
prefixLength : ∀ {x} {p : Path a b n} → x ∈P p → ℕ
prefix : ∀ {x} {p : Path a b n} (i : x ∈P p) → Path a x (prefixLength i)

suffixLength : ∀ {x} {p : Path a b n} → x ∈P p → ℕ
suffix : ∀ {x} {p : Path a b n} (i : x ∈P p) → Path x b (suffixLength i)

lengthsAddUp : ∀
  {x} {p : Path a b n} (i : x ∈P p) →
  n ≡ prefixLength i + suffixLength i
```

4.5.4 Cutting cycles

The segment function recursively identifies the cycle represented by a proof of Cyclic. The base case, here, indicates a path that starts with a cycle; the lemma lengthsAddUp witnesses that splitting the path *p* at index *i* gives back two paths whose lengths add up to the length of the original path. The rewrite keyword rewrites *n* to prefixLength *i* + suffixLength *i* in the type of the goal on the right-hand side of the case, so that the _◂_◂_ constructor can be used to build a return expression (with zero implicitly given for the *m* argument). (The +-identity^r lemma is from Data.Nat.Properties, witnessing that 0 is a right identity for the _+_ function.)

```
segment : {p : Path a b n} → Cyclic p → Segmented a b n
segment {p = e ◃ p} (here i) rewrite lengthsAddUp i =
```

```
    ε ◀ (e ◁ prefix i) ◀ suffix i
  segment {p = e ◁ p} (there r) =
    case segment r of λ where (p₁ ◀ p₂ ◀ p₃) → (e ◁ p₁) ◀ p₂ ◀ p₃
  segment {n = n} {p = p} (end here) rewrite sym (+-identityʳ n) =
    ε ◀ p ◀ ε
  segment {p = e ◁ p} (end (there i)) =
    case segment (end i) of λ where (p₁ ◀ p₂ ◀ p₃) → (e ◁ p₁) ◀ p₂ ◀ p₃
```

Cutting a cycle out of a cyclic path then reduces to segmenting the path and appending the prefix and suffix together without the cycle. The lemma lengthLem $m : m + l < m + \text{suc } n + l$ shows that the output path is strictly shorter than the input path, where the lengths of the prefix, cycle, and suffix are respectively m, suc n, and l.

```
  cutCycle< : {p : Path a b n} → Cyclic p → Path< a b n
  cutCycle< r =
    case segment r of λ where
      (_◀_◀_ {m = m} p₁ p₂ p₃) → -, lengthLem m , (p₁ ◁◁ p₃)
```

The maximum length of an acyclic path in a graph is the number of distinct nodes in the graph; by the finite pigeonhole theorem, any path longer than this maximum length must contain at least one cycle. The standard library includes a form of the finite pigeonhole theorem in Data.Fin.Properties with the name pigeonhole and the type $\forall \{m\ n\} \to m < n \to (f : \text{Fin } n \to \text{Fin } m) \to \exists_2\ \lambda\ i\ j \to i \not\equiv j \times f\ i \equiv f\ j$, witnessing that a mapping from some finite prefix of the natural numbers to a strictly smaller one must map at least two distinct inputs to the same output.

The lemma indicesCycle constructs a proof that a path is cyclic if it includes two

distinct indices that lookup to the same value. In the base cases, where one of the indices is zero, the path begins with a cycle that ends at the other index.

```
indicesCycle : {p : Path a b n} → i ≢ j → lookup p i ≡ lookup p j → Cyclic p
indicesCycle {i = zero} {zero} {e ◁ p} z≢z eq = contradiction refl z≢z
indicesCycle {i = zero} {suc j} {e ◁ p} _ refl = here (∈P-lookup j)
indicesCycle {i = suc i} {zero} {e ◁ p} _ refl = here (∈P-lookup i)
indicesCycle {i = suc i} {suc j} {e ◁ p} si≢sj eq =
  there (indicesCycle (si≢sj ∘ cong suc) eq)
```

With this, pigeonhole can be applied to find a cycle in any path longer than the longest possible acyclic path. The composition Listable.index vertexFinite ∘ lookup *p* maps indices of edge sources in *p* to indices in the list of all vertices; since the list of all vertices is smaller than the length of the path, there must be at least two distinct indices in the path that map to the same index in the list of all vertices. The lemma index-injective transforms a proof of equality between two indices in the list of all vertices into a proof of equality between the vertices at those indices.

```
findCycle : (p : Path a b n) → n > size Vertex^f → Cyclic p
findCycle p gt =
  let _ , _ , i≢j , eq = pigeonhole gt (Listable.index Vertex^f ∘ lookup p) in
    indicesCycle i≢j (index-injective Vertex^f eq)
```

An acyclic path in a graph must contain at most as many edges as there are distinct vertices in the graph. This can be proved by contradiction: in any path with more edges than vertices in the graph, findCycle will be able to find a cycle, showing that the path is not acyclic. While proof by contradiction is not admissible in general in Agda, a form of proof by contradiction over an assumption of a decidable type can be given

as a theorem, provided in Relation.Nullary.Negation as decidable-stable with the type $\forall \{A\} \to \text{Dec } A \to \neg (\neg A) \to A$.

```
acyclic-length-⩽ : (p : Path a b n) → Acyclic p → n ⩽ size Vertex^f
acyclic-length-⩽ {n = n} p ¬r =
  decidable-stable (n ⩽? size Vertex^f)
    λ n⩤v → contradiction (findCycle p (⩤⇒> n⩤v)) ¬r
```

4.5.5 Acyclic paths

As mentioned, the Acyclic predicate restricts the set of all paths to a finite (listable) set. It's possible to construct the Listable set of all Acyclic paths by filtering over the set of all paths of length up to size Vertex^f, but this has performance repercussions: generating every path of up to the maximum acyclic path length and then filtering that list down to only acyclic paths takes time at least proportional to the number of paths of length up to size Vertex^f, which may be significantly greater than the number of distinct acyclic paths in some particular language's set of valid resolution paths.

A more efficient method of construction is to define the listable type of all acyclic paths as a family of listable types of acyclic paths of particular fixed lengths in a style similar to dynamic programming, with the type at each length defined as an extension of the type at the previous index that preserves the Acyclic invariant. By then showing with acyclic-length-⩽ that every type in the family at indices strictly greater than size Vertex^f is empty, all acyclic paths are shown to be contained in the union of the types indexed by values between 0 and size Vertex^f.

Representation

Type synonyms are defined for acyclic paths of fixed lengths and bounded lengths.

```
AcyclicPath : Vertex → Vertex → ℕ → Set _
AcyclicPath a b n = ∃ λ (p : Path a b n) → True (acyclic? p)

AcyclicPath⩽ : Vertex → Vertex → ℕ → Set _
AcyclicPath⩽ a b n = ∃ λ (p : Path⩽ a b n) → True (acyclic? (boundedPath p))
```

Listable proof

```
module Code.Graph.Finite (g : FiniteGraph) where
```

The nexts function generates a list of all paths that can be obtained by extending a given input path by one edge at the end, including any cyclic paths obtained this way.

```
nexts : ∀ {a b n} → Path a b n → List (∃ λ b → Path a b (suc n))
nexts {b = b} p = List.map (λ where (_ , e) → -, p ▷ e) (elements (edgeFrom^f b))
```

The acyclic-nexts function is defined by filtering the output of nexts to only acyclic paths.

```
acyclic-nexts : ∀
  {a b n} (p : Path a b n) →
  List (∃ λ b → AcyclicPath a b (suc n))
```

The ∈-nexts function witnesses that mapping nexts over a list containing all elements of the relevant fixed-length path type generates a list that contains every path one edge

longer, and ∈-acyclic-nexts witnesses the same about acyclic-nexts for fixed-length acyclic paths.

```
∈-nexts : ∀ {a c n} →
  (path^f : Listable (∃ λ b → Path a b n)) →
  (p : Path a c (suc n)) →
  (c , p) ∈ (concatMap (nexts ∘ proj₂) (elements path^f))

∈-acyclic-nexts : ∀ {a c n} →
  (path^f : Listable (∃ λ b → AcyclicPath a b n))
  (p : AcyclicPath a c (suc n)) →
  (c , p) ∈ (concatMap (acyclic-nexts ∘ proj₁ ∘ proj₂) (elements path^f))
```

The fixedAcyclicPathFrom^f and boundedAcyclicPathFrom^f functions generate the listable types of all fixed-length and bounded-length acyclic paths, respectively. The construction of these functions is such that acyclic-nexts is always applied to an acyclic input path, so the filtering operation it carries out is over a list of elements with length equal to the number of acyclic paths of length n from a to b multiplied by the number of edges out of b.

```
fixedAcyclicPathFrom^f : ∀ n a → Listable (∃ λ b → AcyclicPath a b n)
boundedAcyclicPathFrom^f : ∀ n a → Listable (∃ λ b → AcyclicPath⩽ a b n)
```

The acyclicPathFrom^f function uses boundedAcyclicPathFrom^f along with acyclic-length-⩽ to define the type of all acyclic paths out of a given vertex. Since the only use of acyclic-length-⩽ in this development is in the second argument to surjection here, which is never evaluated at runtime, and the definition of acyclic-length-⩽ is the only use of any of the functions relating to identifying and cutting cycles in paths, those

functions are also never evaluated with an argument at runtime during the evaluation of acyclicPathFrom^f.

```
acyclicPathFrom^f : ∀ a → Listable (∃₂ (AcyclicPath a))
acyclicPathFrom^f a =
  boundedAcyclicPathFrom^f (size Vertex^f) a <&^f>
    surjection
      (λ where (b , (n , l , p) , t) → -, -, p , t)
      (λ where (b , n , p , t) → -, (-, acyclic-length-⩽ p (toWitness t) , p) , t)
      λ _ → refl
```

Along with the Listable type used in this thesis, Firsov & Uustalu[8] also give an account of listable types whose elements list contains no duplicates; the elements list output for any input to acyclicPathFrom^f should qualify, and a proof of this is a witness that the union of all the elements lists in the image of acyclicPathFrom^f is minimal as a search space. In general, the code in this development is focused on the goal of verifying the logical specifications of typecheckers by construction rather than on verifying properties of their efficiency, so the definitions above are only shown by construction to produce sufficient search spaces and the claim that they are also non-repetitive search spaces (assuming Vertex^f and Edge^f are non-repetitive) is left formally unverified. Informally, the claim is justified by induction on the lengths of paths: the set of zero-length paths is trivial to define as listable with no duplicates, and the use of nexts in the types of ∈-nexts and ∈-acyclic-nexts produces an output list with no duplicates assuming Edge^f and the *path*^{f} arguments contain no duplicates.

4.5.6 "Best" paths

The Code.Graph.Search module defines the interface in this section, used by the language implementations in Part II to resolve variable references. A language defined in a scope graph framework may include as part of its definition a decidable strict partial ordering over paths and a unary predicate over paths; variable resolution paths must be acyclic, be maximal by the partial ordering, and meet the unary predicate in order to be valid.

```
module Code.Graph.Search
  (g : FiniteGraph)
  {P : ∀ {a} → ∃ (Star (FiniteGraph.Edge g) a) → Set}
  (P^f : ∀ {a} (p : ∃ (Star (FiniteGraph.Edge g) a)) → Listable (P p))
  {_≈_ : ∀ {a} → Rel (∃ (Star (FiniteGraph.Edge g) a)) _}
  {_<_ : ∀ {a} → Rel (∃ (Star (FiniteGraph.Edge g) a)) _}
  (<-dpo : ∀ {a} → IsDecStrictPartialOrder (_≈_ {a}) _<_)
  where
```

The MaximalPath type represents valid variable resolution paths out of a given node.

```
record MaximalPath (a : Vertex) : Set where
  constructor maximalPath
```

The field types (specifically maximal) are defined in a context where the Code.Listable.Ordered module has been opened into scope, specialized to an ordering derived from *<-dpo* for acyclic paths that meet *P*.

```
  field
    destination : Vertex
```

```
    path : Star Edge a destination
    predicate : P (destination , path)
    acyclic : True (acyclic? (fromStar path))
    maximal : True (maximal? ((destination , path , acyclic) , predicate))

open MaximalPath public
```

The size of the set of all variable resolution paths should not depend on the number of distinct values of the types of the acyclic and maximal fields, since these represent logical invariants, so they are encoded with the True type. The unary predicate type, however, should factor into the total number of valid variable resolution paths: among other things, this is where a language definition includes the requirement that a path resolving some variable must end at a declaration with the same name as the variable it's resolving, so there should be more than one MaximalPath value with the same path and different values for predicate in cases where a single scope binds the same name to multiple declarations, to represent ambiguity in the resolution.

The work to show MaximalPath listable has all already been done in the Code.Listable.Ordered module. The definition of maximalPathsFrom^f simply converts a MaximalPath value to the nested product type used in the type of maxes, and unification can even infer the inverse function in the other direction. The occurrence of maxes here is specialized to a derived ordering in a similar way as in the types of the MaximalPath fields, this time specialized to *maximal* paths that meet the given predicate.

```
maximalPathsFrom^f : (a : Vertex) → Listable (MaximalPath a)
maximalPathsFrom^f a =
  ∃maximal^f <&^f>
    surjection
```

```
_
(λ where (maximalPath b p pr ac m) → ((b , p , ac) , pr) , m)
(λ _ → refl)
```

The first argument to surjection in the maximalPathsFrom^f definition can be uniquely inferred by unification, thanks to the definitional η-equality rules for function abstractions and record constructors. This is generally only true when the destination type is a simple type, but is very convenient when it works: the surjectivity of the function is effectively proven for free, and the only cost to readability over an unverified implementation aside from a bit of syntactic boilerplate is that the function written in the source code is actually the inverse of the function that is mapped over the elements list of ∃maximal^f at runtime.

Chapter 5 Scope graphs

The *scope graph calculus* [10] is a flexible theoretical framework for describing the scoping and binding semantics of programming languages, in which some interesting language features are more natural to model than with traditional contexts of bindings. The object languages in Part II of this thesis are defined within an Agda framework encoding the theory of scope graphs from Bach Poulsen et al. [3]. This chapter summarizes Sections 4.2 and 4.3 in their work and explains a minor generalization of their framework in which any arbitrary type can be used as the type of scopes in a scope graph; variable resolution in a scope graph becomes a special case of the graph resolution procedure from Chapter 4 of this thesis in any graph with a Listable type of scopes.

5.1 Motivation

This section does not attempt to fully motivate the theory of scope graphs in general but focuses mainly on the benefit it brings to the module system implementation in Chapter 6, specifically the ability to model *cyclic dependencies* between scopes in a natural way, enabling a straightforward definition of a module system with no restrictions on cyclic imports. Cyclic module imports are not the most common example of cyclic scope dependencies in real-world languages, but are a relatively minimal representative example of the feature. Many languages do support some kind of local cyclic scope dependency, especially in the OOP paradigm in the form of mutually recursive class definitions; this is the form of cyclic dependency covered in the formalization of Middleweight Java in Bach Poulsen et al. [3].

Consider the small program below, in pseudocode in a language including modules

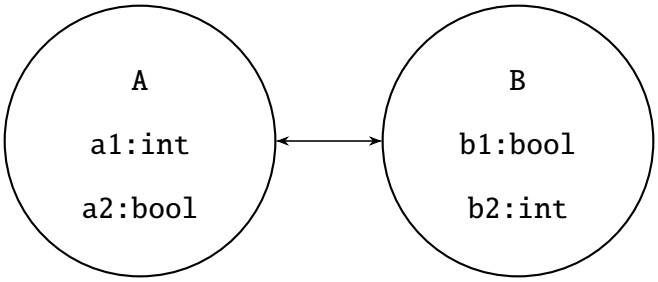
and module imports.

```
module A:
    import B
    var a1:  int = b2
    var a2:  bool = true
```

```
module B:
    import A
    var b1:  bool = a2
    var b2:  int = 0
```

`A` and `B` are in a *dependency cycle*, since each imports the other. The intended semantics of this particular program are informally clear: when all definitions are evaluated, the result should be `a1 = b2 = 0` and `b1 = a2 = true`. The general semantics of cyclic module dependencies, however, are more complicated, and require significant effort to formalize in a traditional setting.

The scoping semantics of the program above can be given straightforwardly as a scope graph, as illustrated below. The two-headed arrow between the scopes for `A` and `B` represents the two edges corresponding to the two import statements in the program, and each scope contains a name and type for each declaration that it binds.



The evaluation semantics of this program can be described in the "scopes-and-frames" model of Bach Poulsen et al. [2], and an Agda implementation of the similar feature in Middleweight Java is described in Bach Poulsen et al. [3]. Briefly, the idea is that the runtime heap contains "frames" in a many-to-one relation with the set of scopes in the program being evaluated, where each frame contains a value for each declaration

bound in the associated scope and a pointer to a corresponding destination frame for each edge leading out of the associated scope. The frames for the module-scoped values declared in `A` and `B` each depend on data in the other, but in such a way that it is possible to construct them programatically without diverging. The Appendix of this thesis covers an interpreter in this style for the language defined in Chapter 8.

5.2 Basic definitions

The code in this section is found in the Code.Scope module in the development associated with this thesis.

The data contained within a scope is represented by a pair of lists, one representing an adjacency list encoding of the set of edges out of that scope and one representing the declarations that the scope binds.

```
ScopeData : Set → Set → Set
ScopeData S D = List S × List D

edges = proj₁ ; decls = proj₂
```

In constructing scope graphs programmatically, it will sometimes be necessary to map a function over the edges list in a ScopeData value; this corresponds to applying a graph homomorphism to the represented edges, which is used when extending a scope graph with new scopes while preserving the meaning of the data at each of the existing scopes. (Σ is a local alias for Data.Product, as before, and $\Sigma.\mathrm{map}_1$ is the function to map over the first element of a pair.)

```
mapEdges : ∀ {S S′ D} → (S → S′) → ScopeData S D → ScopeData S′ D
mapEdges f = Σ.map₁ (List.map f)
```

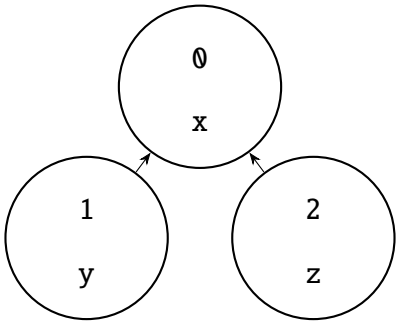
A ScopeGraph is a function mapping scopes to associated ScopeData.

```
ScopeGraph : Set → Set → Set
ScopeGraph S D = S → ScopeData S D
```

It is illustrative at this point to consider an encoded scope graph for an example program. The raw abstract syntax of a small example language of Boolean values is given below; this language is only used to demonstrate some of the basic principles of scope graph generation and will not be given a full semantics. (The let′ constructor is named to avoid clashing with the Agda let keyword.)

```
data Expr : Set where
  true false : Expr
  var : String → Expr
  let′ : String → Expr → Expr → Expr
  if : Expr → Expr → Expr → Expr
```

The meaning of let′ x e_1 e_2 in standard notation is let $x = e_1$ in e_2. Each let′ node in an Expr AST corresponds to a single scope in the graph of the expression, since each let′ declares a new identifier and no other Expr forms do. For example, the scope graph for the term let′ "x" true (if (var "x") (let′ "y" true true) (let′ "z" true true)) is given below, with the scopes numbered in an arbitrary order.



As this language has only one type, scopes bind only names, and the type of every binding is assumed to be the type of Boolean values. When expressing this graph as a ScopeGraph, the *D* argument will simply be String. One natural choice for the *S* argument when creating graphs by hand is the Fin type as used in Bach Poulsen et al. [3], in this case Fin 3 to represent the three scopes. The definition of g below is an encoding of the above graph in this manner.

```
g : ScopeGraph (Fin 3) String
g zero = [] , [ "x" ]
g (suc zero) = [ zero ] , [ "y" ]
g (suc (suc zero)) = [ zero ] , [ "z" ]
```

5.3 Motivating the generalization

While Fin is a convenient type of scopes in manually-constructed scope graphs, it is a somewhat inconvenient type of scopes to use during scope graph generation. To demonstrate the issues that arise, this section details the construction of scope graphs representing the binding structures of terms in this Expr language both with Fin scopes and with an inductively-defined datatype. This is not intended as an in-depth explanation of the mechanics of scope graph generation, which is covered in Part II, but rather as a motivation for the generalization of the scope graph library from Bach Poulsen et al. [3] to support arbitrary types of scopes (where theirs only supports scopes of type Fin).

Generating scope graphs with Fin scopes

A function to generate a scope graph with Fin scopes for any arbitrary input Expr must first calculate how many scopes there are in the term, in order to know the index to use

with Fin. The scopes function does this recursively, where the suc constructor in the let′ represents adding a new scope. In this language, the number of scopes in a term is exactly the number of let′ AST nodes in the term, because let′ is the only expression form that binds a new name.

```
scopes : Expr → ℕ
scopes (let′ x e₁ e₂) = suc (scopes e₁ + scopes e₂)
scopes (if e₁ e₂ e₃) = scopes e₁ + scopes e₂ + scopes e₃
scopes _ = zero
```

Given this definition, the type of the scope graph representing the binding structure of a given term e is a ScopeGraph with scopes of type Fin (scopes e).

```
graph : (e : Expr) → ScopeGraph (Fin (scopes e)) String
```

This type normalizes to $(e : \text{Expr}) \to \text{Fin}\ (\text{scopes}\ e) \to \text{List}\ (\text{Fin}\ e) \times \text{List String}$, so it is defined as taking two arguments and returning a pair: the graph of e is represented as a lookup function that takes a scope value as an argument and returns the contents of the scope. In the let′ case, the scope zero is designated as the scope corresponding to the outermost scope in the program, the one that the let′ AST node under consideration binds.

```
graph (let′ x e₁ e₂) zero = [] , [ x ]
```

An impediment arises in the suc case for let′, when the pattern is graph (let′ x e_1 e_2) (suc i). The type of i is Fin (scopes e_1 + scopes e_2), and the types of graph e_1 and graph e_2 are ScopeGraph (Fin (scopes e_1)) String and ScopeGraph (Fin (scopes e_2)) String respectively. A value of type Fin (scopes e_1 + scopes e_2) can be decomposed into a value of type

Fin (scopes e_1) ⊎ Fin (scopes e_2) in order to obtain an index suitable for recursing into one of the two subgraphs with, but this is not a definitional property of the Fin type, and must be given by a lemma, as in split+ below.

```
split+ : ∀ {m n} → Fin (m + n) → Fin m ⊎ Fin n
split+ {zero} i = inj₂ i
split+ {suc m} zero = inj₁ zero
split+ {suc m} (suc i) =
  case split+ i of λ where
    (inj₁ i′) → inj₁ (suc i′)
    (inj₂ i′) → inj₂ i′
```

The suc case for let′ can be defined using this lemma. The expression suc ∘ inject+ _ here has type Fin (scopes e_1) → Fin (suc (scopes e_1 + scopes e_2)), and raise is used with the type Fin (scopes e_2) → Fin (suc (scopes e_1 + scopes e_2)); inject+ and raise are from the standard library Data.Fin module. In the case where i is an index for a scope in the subgraph of e_2, an edge to the zero scope is added to its adjacency list with Σ.map₁ (zero ::_) to represent the "parent scope" relationship between the scope associated with the let′ AST node and the subgraph of scopes associated with the bindings in e_2. (This is a somewhat careless design, made to keep this example simple for the purpose of demonstration, with the result that an edge is added to zero for *each* scope in the graph of e_2, not just the root scope as would likely be expected.)

```
graph (let′ x e₁ e₂) (suc i) =
  case split+ i of λ where
    (inj₁ i′) → mapEdges (suc ∘ inject+ _) (graph e₁ i′)
    (inj₂ i′) → Σ.map₁ (zero ::_) (mapEdges (raise _) (graph e₂ i′))
```

The case for if does not have to deal with adding a new scope to the graph, but does still have to split its Fin argument in a similar way as the code above, this time with an input of type Fin (scopes e_1 + scopes e_2 + scopes e_3). This can be done with a new bespoke lemma or by applying split+ twice, as in the code below. The function arguments to mapEdges convert scope indices from each respective subgraph into indices in the graph of the if term, in the same way as in the let′ (suc i) case above.

```
graph (if e₁ e₂ e₃) i =
  case split+ i of λ where
    (inj₁ i′) →
      case split+ i′ of λ where
        (inj₁ i″) → mapEdges (inject+ _ ∘ inject+ _) (graph e₁ i″)
        (inj₂ i″) → mapEdges (inject+ _ ∘ raise (scopes e₁)) (graph e₂ i″)
    (inj₂ i′) → mapEdges (raise _) (graph e₃ i′)
```

The split+ lemma is used in the definition of graph above to select which subexpression of a term a given scope index is bound within, but it is not particularly well-suited for this purpose, running in O(m) time in the implicit m argument to the function, and it only applies directly to expression AST node types with exactly two subexpressions. While it may be possible to define a more general library for casing over Fin values in this way, there is a simpler solution requiring no additional code: inductive data types.

The only semantically important property of Fin for representing scopes is that it is a *finite* type, so that the graph search procedure from Chapter 4 can be used to resolve variable references when typechecking with a scope graph. This property is not depended on by any code within the scope graph library, and it is satisfied by any Listable type, so the approach taken in this thesis is to generalize the scope graph library to any arbitrary type of scopes and use the Listable type in typechecking code to

constrain scope types to be finite types.

Generating scope graphs with inductively-defined scope types

The type of scopes in an object language term is encoded by an inductively-defined data type specific to the object language, defined to have exactly as many members as there are scopes in the graph corresponding to the binding structure of the term. The Scope data type is not meant to represent the entire binding structure of a term, but just the *set of vertices* in the scope graph representing the binding structure.

```
data Scope : Expr → Set where
  let′-here : ∀ {x e₁ e₂} → Scope (let′ x e₁ e₂)
  let′-def : ∀ {x e₁ e₂} → Scope e₁ → Scope (let′ x e₁ e₂)
  let′-body : ∀ {x e₁ e₂} → Scope e₂ → Scope (let′ x e₁ e₂)
  if-cond : ∀ {e₁ e₂ e₃} → Scope e₁ → Scope (if e₁ e₂ e₃)
  if-true : ∀ {e₁ e₂ e₃} → Scope e₂ → Scope (if e₁ e₂ e₃)
  if-false : ∀ {e₁ e₂ e₃} → Scope e₃ → Scope (if e₁ e₂ e₃)
```

This pattern is explained in more detail in Chapter 7, but at a high level the Scope type encodes the type of all paths from the root of an Expr AST to some let′ node within the AST, with the let′-here constructor representing the scope of a let′ AST node at the root of the AST. The type is Listable at all indices, so it is suitable for use as the vertex type in the FiniteGraph library from Chapter 4, and for any *e*, Scope *e* is isomorphic to Fin at some index (specifically the cardinality of Scope *e*). The definition of graph below with this type of scopes is significantly simpler and more efficient than the definition above with Fin scopes, because Agda's built-in pattern-matching functionality is used to decide which subgraph to recurse into, as opposed to the split+ lemma.

```
graph : (e : Expr) → ScopeGraph (Scope e) String
graph (let′ x e₁ e₂) let′-here = [] , [ x ]
graph (let′ x e₁ e₂) (let′-def i) = mapEdges let′-def (graph e₁ i)
graph (let′ x e₁ e₂) (let′-body i) =
  Σ.map₁ (let′-here ::_) (mapEdges let′-body (graph e₂ i))
graph (if e₁ e₂ e₃) (if-cond i) = mapEdges if-cond (graph e₁ i)
graph (if e₁ e₂ e₃) (if-true i) = mapEdges if-true (graph e₂ i)
graph (if e₁ e₂ e₃) (if-false i) = mapEdges if-false (graph e₃ i)
```

5.4 Resolution in scope graphs

The definitions in this section are parameterized over an arbitrary scope graph.

```
module ScopeGraph {S D} (g : ScopeGraph S D) where
```

As mentioned, the scope graph library described in this chapter is intended to be used in conjunction with the graph search library from Chapter 4 for path resolution. In order to set this up, it is helpful to make the types of edges and paths in a scope graph explicit. The Edge type defined below represents the type of edges between two scopes in a scope graph: a scope *s* has an edge to another scope *s′* when *s′* is in the edges list of *s*. The _►_ operator is defined as a binary synonym for Edge, and _►*_ as a synonym for the type of paths over scope graph edges, using the Star type from Relation.Binary.Construct.Closure.ReflexiveTransitive as a type of paths as in Chapter 4.

```
  Edge : S → S → Set
  Edge s s′ = s′ ∈ edges (g s)
```

_►_ = Edge

_►*_ = Star Edge

This type of edges forms a FiniteGraph whenever S is Listable, as witnessed by finiteScopeGraph below.

```
finiteScopeGraph : Listable S → FiniteGraph
finiteScopeGraph S^f =
  finiteGraph S^f (Σ^f S^f λ s → Σ^f S^f λ s′ → s′ ∈^f edges (g s))
```

The object language implementations in Part II use this graph to specialize the Code.Graph.Search module from Chapter 4 in order to obtain a version of the maximalPathsFrom^f function suitable for resolving variable reference paths.

Part II

Scopechecking and Typechecking

Part II presents a module system parameterized over an arbitrary base language, defined using the constructs from the previous chapter, and then presents two suitable example base languages: simply-typed lambda calculus with boolean and natural number types, and a toy procedural language with booleans, integers, arrays, and pointers (without pointer arithmetic). In contrast to the generally bottom-up style of presentation in most of this thesis, the module system is presented before the languages it applies to; the purpose of this presentation structure is to introduce the general pattern of base language definition that the module system works with, in order to motivate some otherwise nonobviously-relevant features of the example language definitions.

Chapter 6 Module system

The module system presented in this chapter can be described as a small framework for implementing the semantics of languages that feature a particular kind of module system. The idea is similar to that of the “modular module system” presented in Leroy [9], but works within a scope graph framework and fully supports cyclic dependencies between modules. A module in this system is defined as a collection of named and explicitly-typed terms, and a program is a collection of modules along with an explicitly-typed “main” expression that describes the evaluation of the program when executed.

The clients of this framework import the definitions in this module, with a sort of double inversion of control: the client provides a scope graph construction procedure for terms in a term language and obtains a scope graph construction procedure for programs, and then the client provides a typechecking procedure for terms within those constructed program scope graphs and obtains a typechecking procedure for programs. The module system code does not provide any code to evaluate a program, but does include code to construct an initial heap for a program’s evaluation, so term languages can easily define their own evaluators over intrinsically-typed programs. (Evaluation is described in the Appendix; most details about it are left out of this part of the thesis.)

The code associated with this chapter is found in the Code.Module module in the Agda development.

6.1 Names

Identifiers are encoded with the primitive String type imported from Data.String.

```
Name = String
variable x y z : Name
```

This is an arbitrary choice for convenience; the only algorithmically relevant property of strings is that equality over them is decidable.

6.2 PreScope and PreGraph

When a term language procedure constructs a scope graph for some term in a program, it does not have access to the complete program scope graph for inspection, since it hasn't been fully built yet; still, the procedure should have some way in general to produce a graph that includes edges that point outside the graph of that particular term to other scopes in the program. A limited form of this capability is achieved by having the term language scope graph construction procedure for terms output a PreGraph, in which the adjacency list at any given scope may contain references to a PreScope called free; when the program graph is built, all edges to free in the term graphs are replaced with edges to the scopes that each respective term is scopechecked within.

```
data PreScope S : Set where
  free : PreScope S
  bound : S → PreScope S

PreGraph : Set → Set → Set
PreGraph S T = S → ScopeData (PreScope S) (Name × T)
```

The bind function is the eliminator for PreScope, from which a mapping function can be derived. (PreScope is isomorphic to Maybe, and bind corresponds to the maybe eliminator with the arguments flipped.)

```
bind : {S S′ : Set} → S′ → (S → S′) → PreScope S → S′
bind x f free = x
bind x f (bound s) = f s

mapPreScope : ∀ {S S′} → (S → S′) → PreScope S → PreScope S′
mapPreScope f = bind free (bound ∘ f)
```

6.3 Raw programs

The RawModuleSystem module takes parameters from a term language defining the Agda types of object term language types and raw terms, along with a finite type of scopes and a function constructing a PreGraph over those scopes for any given term. The rest of the code in this chapter is defined within this module, including the TypedModuleSystem module in Section 5.4. (The variable declarations are private to this module to avoid clashing with similar declarations in the base language modules.)

```
module RawModuleSystem
  (Type Term : Set)
  {TermScope : Term → Set}
  (TermScope^f : ∀ e → Listable (TermScope e))
  (termGraph : ∀ e → PreGraph (TermScope e) Type)
  where
  private
    variable
      t : Type
      tm : Term
```

6.3.1 Modules

A raw Module contains a list of imports and a list of named and explicitly-typed raw declarations.

```
record Module : Set where
  constructor mod
  field
    imports : List Name
    declarations : List (Name × Type × Term)

  termSignatures : List (Name × Type)
  termSignatures = List.map (λ where (x , t , _) → x , t) declarations

open Module

variable md : Module
```

6.3.2 Programs

A raw Program contains a list of named modules and an explicitly-typed main declaration depending on a specified list of imports. The rest of the code within this subsection is defined within the Program record module.

```
record Program : Set where
  constructor prog
  field
    modules : List (Name × Module)
    mainImports : List Name
```

```
  mainType : Type
  mainTerm : Term
```

Scopes

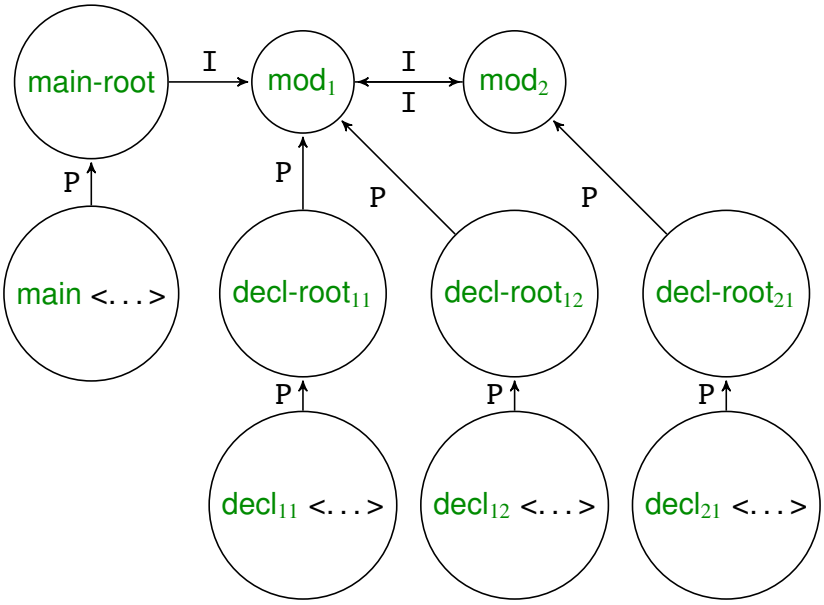
The Scope type describes the set of scopes in a program.

```
data Scope : Set where
  main-root : Scope
  main : TermScope mainTerm → Scope
  mod : (x , md) ∈ modules → Scope
  term-root : (i : (x , md) ∈ modules) (j : (y , t , tm) ∈ declarations md) → Scope
  term :
    (i : (x , md) ∈ modules) (j : (y , t , tm) ∈ declarations md) →
    TermScope tm → Scope

Scope^f : Listable Scope
instance scope-decEq : DecEq Scope
```

The example graph below illustrates the general shape of a program graph in this language: the main declaration is contained within a distinct root scope, and each other module in the program has a scope associated with the module itself and a root scope for each of the module's declarations, each containing the graph of the respective declaration. Subscript numbers represent values of list membership types seen as numeric indices, and the metasyntax <...> after a constructor name within a node indicates that the node actually represents a subgraph for some declaration, with all scopes in the subgraph constructed with that construtor. Import edges are annotated

with an `I` and lexical parent edges with a `P` - a term language might in general choose to restrict the set of valid variable resolution paths to e.g. only those that include at most one import edge. In this example graph, mod_1 has two declarations, mod_2 has one, main imports mod_1, and mod_1 and mod_2 mutually import each other. The -root scopes are auxiliary scopes used in the process of scopechecking raw terms, explained in detail by example in Subsection 7.2.2.



Scope graph construction

A program Graph is a ScopeGraph mapping program Scope values to declaration signatures.

```
Graph : Set
Graph = ScopeGraph Scope (Name × Type)
```

The importScopes function takes in a list of names and returns the list of all module scopes whose names are in the input list, used to generate the list of scopes that

represent the outgoing edges of a module scope (the module's import edges). One consequence of this definition is that attempting to import a nonexistent module is not an error in general, instead simply silently adding zero import edges to the resulting graph; this could be rectified by returning a Maybe result from importScopes to represent failure in the case that an import would add zero edges, but it is left as-is here for simplicity. (Similarly, importing the same module multiple times will only trigger ambiguity errors if definitions from the module are referenced.)

```
importScopes : List Name → List Scope
```

The graph function defined below constructs a scope graph representing the binding structure of any given program. The uses of bind in graph map occurrences of the free constructor in the graphs of declarations generated by the term language to the appropriate -root scopes; main-root has edges to all of the imports in mainImports and binds the main expression with the name `"main"`, mod scopes have edges to their imports and bind their declarations, and decl-root scopes have edges to their parent modules.

```
graph : Graph
graph main-root = importScopes mainImports , [ "main" , mainType ]
graph (main i) = mapEdges (bind main-root main) (termGraph mainTerm i)
graph (mod {md = md} i) = importScopes (imports md) , termSignatures md
graph (term-root i j) = [ mod i ] , []
graph (term {tm = tm} i j k) =
  mapEdges (bind (term-root i j) (term i j)) (termGraph tm k)
```

6.4 Intrinsically-typed programs

The code in this section (still within the RawModuleSystem module) defines the TypedModuleSystem module as an inner module with a parameter list representing a language with intrinsically-typed values (_⊨_) and terms (_⊢_⦂_), along with a function that produces every intrinsically-typed term with a given type that corresponds to a given declaration in a given program graph and scope. (The Weakenable type is the same as the one defined in [3], and the _⊨_ and ⊨-*weakenable* parameters are only relevant in evaluation, covered in the appendix to this thesis.) The code in this module is also parameterized over a raw Program, which is the program to be typechecked. Notably, the term type is parameterized over specifically the type of scopes in program *p*, so a term language definition has the ability to reason about program scopes and not just term scopes (e.g., to limit the set of valid variable resolution paths so that module imports are intransitive).

```
module TypedModuleSystem
  (p : Program)
  (_⊨_ : ScopeGraph.HeapType (Program.graph p) → Type → Set)
  {{⊨-weakenable : ∀ {t} → Weakenable (_⊨ t)}}
  {_⊢_⦂_ : Program.Scope p → Term → Type → Set}
  (_⊢_⦂ᶠ_ : ∀ s e t → Listable (s ⊢ e ⦂ t))
  where
  open Program p
  open ScopeGraph graph
```

6.4.1 Modules

The ModuleTyping type represents intrinsically-typed modules, indexed by some element of the modules list of the raw program *p*. An intrinsically-typed module is defined as an All list containing a term for each declaration in the given module, where the terms are each intrinsically-typed in their corresponding root scope.

```
ModuleTyping : (x , md) ∈ modules → Set
ModuleTyping {md = md} i =
  All
    (λ where ((_ , t , e) , j) → term-root i j ⊢ e ⦂ t)
    (indexed (declarations md))
```

6.4.2 Programs

A ProgramTyping is a term for the main declaration in *p* scopechecked in the main-root scope, along with a ModuleTyping for each module in the modules list of *p*.

```
record ProgramTyping : Set where
  constructor prog
  field
    mainTyping : main-root ⊢ mainTerm ⦂ mainType
    moduleTypings : All (λ where (_ , i) → ModuleTyping i) (indexed modules)
```

6.4.3 Typechecking

The type of intrinsically-typed raw modules corresponding to any given module in the modules list of *p* is a finite type, since the type of terms is finite.

```
ModuleTyping^f : (i : (x , md) ∈ modules) → Listable (ModuleTyping i)
ModuleTyping^f i = All^f λ where ((_ , t , e) , j) → term-root i j ⊢ e ◃^f t
```

This implies that the type of intrinsically-typed programs corresponding to any given raw program p is finite, since both fields of the ProgramTyping field are of finite types.

```
ProgramTyping^f : Listable ProgramTyping
```

It's then straightforward to decide whether p has a unique typing.

```
uniqueTyping? : Dec (Singleton ProgramTyping)
uniqueTyping? = singleton? ProgramTyping^f
```

The uniqueTyping function is the main top-level interface to typechecking used by clients of the language implementations in the next two sections.

Chapter 7 Simply-typed Lambda Calculus

The definition of the intrinsically-typed syntax type for simply-typed lambda calculus (STLC) in this chapter is almost identical to the one in Bach Poulsen et al. [3], with one important difference: the path that resolves a variable reference is (intrinsically) required to be a *maximal* and *well-formed* path according to the specificity ordering and well-formedness predicate that define the scoping rules of the language. The standard lexical shadowing rules of arguments in STLC are encoded as an ordering on the lengths of paths, and a well-formedness predicate is chosen to allow only paths with at most one import edge, in order to avoid transitive imports.

An implementation of STLC as a base language for the module system in the previous section must provide a procedure to construct a scope graph for an STLC term and a procedure to generate all intrinsically-typed terms under the scope graph of some program that erase to some given raw term. Encoding the restrictions on variable resolution paths intrinsically guarantees that a typechecker outputting a finite type of intrinsically-typed terms only outputs terms that respect the lexical shadowing rules of STLC - for example, the raw term $\lambda\ (x : \mathsf{nat}).\ \lambda\ (x : \mathsf{nat}).\ x$ has two corresponding members in a type of intrinsically-typed terms with unrestricted variable paths, but only one corresponding member in a type of intrinsically-typed terms with maximal paths in which the x reference resolves to the inner binding.

7.1 Raw expressions

7.1.1 Types

Natural number and boolean types are included for the sake of building non-trivial example programs.

```
infixr 7 _⇒_
data Type : Set where
  bool nat : Type
  _⇒_ : Type → Type → Type

variable t t₁ t₂ : Type
```

In typechecking function applications it will be especially relevant to decide whether some given type t is a function type accepting an argument of another type t_1, which is implemented by the acceptsArg? decision procedure; the output of this function is propositional, making it suitable for use with the filterProp^f function. Propositional equality is also decidable in general over STLC types.

```
acceptsArg? : ∀ t t₁ → Dec (∃ λ t₂ → t ≡ (t₁ ⇒ t₂))

instance
  acceptsArg-prop : Propositional (∃ λ t₂ → t ≡ (t₁ ⇒ t₂))
  type-decEq : DecEq Type
```

7.1.2 Expressions

This representation of STLC syntax is explicitly-typed: lambda parameter types are always given explicitly in the source representation of a term.

```
infixl 8 _•_
data Expr : Set where
  bool : Bool → Expr
  nat : ℕ → Expr
  suc pred iszero : Expr → Expr
  if : Expr → Expr → Expr → Expr
  var : Name → Expr
  ƛ : Name → Type → Expr → Expr
  _•_ : Expr → Expr → Expr

variable
  b : Bool
  n : ℕ
  e e₁ e₂ e₃ : Expr
```

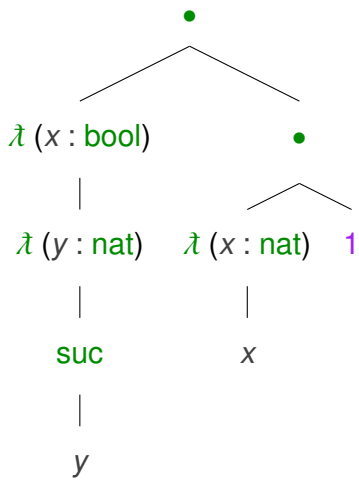
7.1.3 Scopes

The set of lexical scopes local to a particular STLC expression can be identified with the set of all lambda nodes in the expression's AST. This set is encoded as an inductive type of paths from the root of an expression AST to some lambda node; the structure of the type is similar to that of _∈_, with ƛ-here corresponding to the here constructor and each other constructor corresponding to a variant of there for some specified subtree of the root node of the given expression AST. (Note that the Scope type defined in the Program record module is only in scope here as Program.Scope since Program hasn't been opened, so this is not a name conflict.) The set of scopes in some given expression AST is finite, by straightforward structural induction on the AST.

```
data Scope : Expr → Set where
  ƛ-here : Scope (ƛ x t e)
  ƛ-there : Scope e → Scope (ƛ x t e)
  •-left : Scope e₁ → Scope (e₁ • e₂)
  •-right : Scope e₂ → Scope (e₁ • e₂)
  suc : Scope e → Scope (suc e)
  pred : Scope e → Scope (pred e)
  iszero : Scope e → Scope (iszero e)
  if-cond : Scope e₁ → Scope (if e₁ e₂ e₃)
  if-yes : Scope e₂ → Scope (if e₁ e₂ e₃)
  if-no : Scope e₃ → Scope (if e₁ e₂ e₃)

Scope^f : ∀ e → Listable (Scope e)
```

For example, the AST is given below for the term $(\lambda\ (x : \mathsf{bool}).\ \lambda\ (y : \mathsf{nat}).\ \mathsf{suc}\ y)\ ((\lambda\ (x : \mathsf{nat}).\ x)\ 1)$; the three members of Scope indexed over this term are •-left ƛ-here, •-left (ƛ-there ƛ-here), and •-right ƛ-here, representing the ƛ nodes in this AST in the order of an inorder traversal.

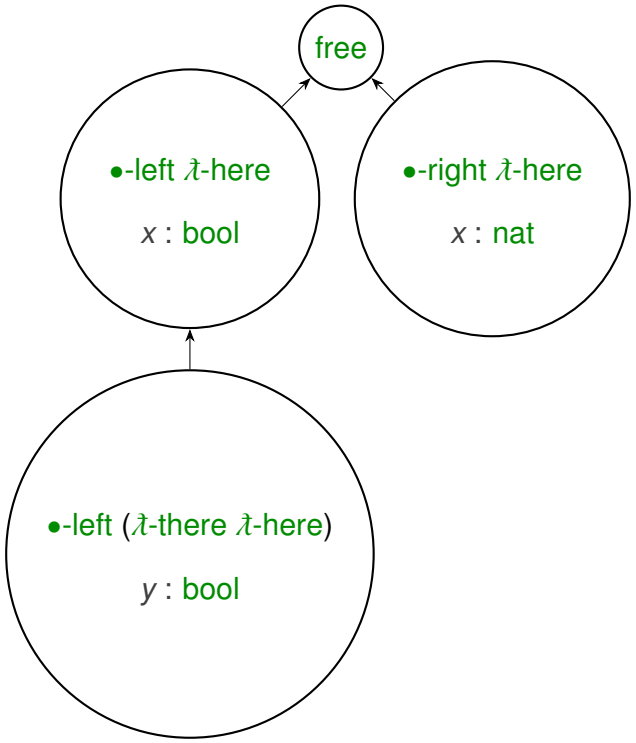


7.1.4 Scope graph construction

The PreGraph for an STLC expression encodes the lexical binding structure of the term.

```
graph : ∀ e → PreGraph (Scope e) Type
```

For an example of the goal, the term given an AST in the previous subsection should generate this scope graph below; the scopes apart from free are to be interpreted as under the bound constructor, which is omitted in the diagrams to save space. (This is just the subgraph for a particular term's local scopes, so the diagram does not include module scopes or import edges.)



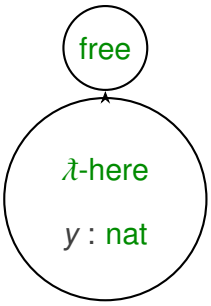
As this function is specifically returning a PreGraph, the edges lists of the output ScopeData values may reference the free pre-scope. The λ-there case notably leverages this to replace any references to the free scope in the graph of the lambda body with a

reference to the scope of the lambda itself (ƛ-here), representing the conversion of a free variable to a bound variable that can occur when grafting raw ASTs together. The ƛ-here case includes an edge to the free scope, so that the definitions in the body of the lambda will have access to all declarations accessible from the scope containing the lambda once everything has been all connected up.

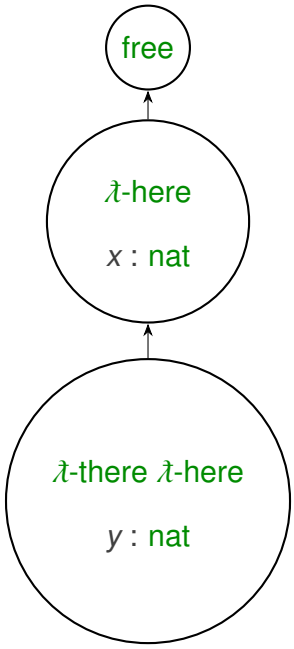
```
graph (ƛ x t e) ƛ-here = [ free ] , [ (x , t) ]
graph (ƛ x t e) (ƛ-there i) = mapEdges (bound ∘ bind ƛ-here ƛ-there) (graph e i)
```

As an example of the behavior of these potentially nonobvious ƛ cases of graph, the construction of the scope graph for the term λ (x : nat). λ (y : nat). y proceeds as follows.

- graph is called on the outer lambda and recurses into the body subterm, λ (y : nat). y.
  - graph is called on the inner lambda and recurses into the body subterm, y.
    - An empty graph is returned for y, because there are no cases for the var constructor in graph - the type Scope (var p) is definitionally empty for any p, so there can be no value for the second argument to graph if the first unifies with the pattern var p.
  - The graph of λ (y : nat). y is constructed by adding one ƛ-here scope and a ƛ-there scope for each scope in the graph of y, which is empty, so the graph of λ (y : nat). y only has one actual scope, which has an edge to the free pre-scope.



- The graph of $\lambda\ (x : \mathsf{nat}).\ \lambda\ (y : \mathsf{nat}).\ y$ is constructed by adding one λ-here scope and a λ-there scope for each scope in the graph of $\lambda\ (y : \mathsf{nat}).\ y$, which only has the one scope, so the graph of $\lambda\ (x : \mathsf{nat}).\ \lambda\ (y : \mathsf{nat}).\ y$ has two actual scopes apart from free. Each scope from the graph of the inner lambda is copied over under a λ-there, and the edges that used to point to free are redirected to point to λ-here while the λ-here is given a new edge pointing to free.

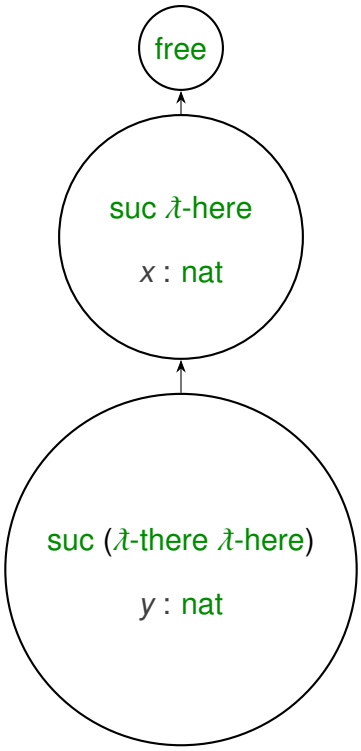


All other cases of the graph function for STLC are straightforward instances of a simpler pattern, recursing into the appropriate subterm and mapping the appropriate

Scope constructor over all of the edges in the resulting graph. None of these affect any existing edges to free, since none of the expression forms in these cases bind any new identifiers.

```
graph (suc e) (suc i) = mapEdges (mapPreScope suc) (graph e i)
graph (pred e) (pred i) = mapEdges (mapPreScope pred) (graph e i)
graph (iszero e) (iszero i) = mapEdges (mapPreScope iszero) (graph e i)
graph (if e₁ e₂ e₃) (if-cond i) = mapEdges (mapPreScope if-cond) (graph e₁ i)
graph (if e₁ e₂ e₃) (if-yes i) = mapEdges (mapPreScope if-yes) (graph e₂ i)
graph (if e₁ e₂ e₃) (if-no i) = mapEdges (mapPreScope if-no) (graph e₃ i)
graph (e₁ • e₂) (•-left i) = mapEdges (mapPreScope •-left) (graph e₁ i)
graph (e₁ • e₂) (•-right i) = mapEdges (mapPreScope •-right) (graph e₂ i)
```

For a simple example, consider the (nonsense) term suc (λ (x : nat). λ (y : nat). y), obtained by adding a suc AST node at the root of the term considered in the example for the λ cases above. The example graph with three scopes above is the graph of the subterm of suc, so the graph of the whole term including suc is the same but with the suc Scope constructor added to the identifier of each scope.



7.1.5 Specificity ordering and well-formedness predicate

The definitions in this section so far are sufficient to fill in all of the parameters to the RawModuleSystem module, so the parts of the module system concerning scope graph construction can be opened and used to construct the specificity ordering that defines the lexical shadowing rules of STLC and the well-formedness predicate that will prevent transitive module imports. This module is opened publicly so that later stages of the validation pipeline can import its contents just by importing Code.STLC.Raw; the Code.STLC.Raw module can be thought of as a definition of the whole language of programs that include modules of raw STLC terms, with the definitions in previous subsections providing the definition of the term language and the opening of RawModuleSystem providing the definition of the program language.

```
open RawModuleSystem Type Expr Scope^f graph public
```

The rest of the definitions in this subsection are parameterized over an arbitrary program, an arbitrary scope graph for that program, and the name of a variable being resolved. This module is opened later with the *p* parameter supplied and the *x* argument left unfilled, so that all the definitions take an explicit Name argument.

```
module VarPath (p : Program) (x : Name) where
  open module Prog = Program p renaming (graph to programGraph)
  open ScopeGraph programGraph
```

The declIndex field of the well-formedness predicate type represents an index of the variable name in question in the list of declarations at the destination scope of the given variable resolution path *p*, and the ⩽1-import-edge field witnesses that *p* contains at most one import edge by requiring that the type of indices of module scopes in the list of all scopes in the path is a propositional type. (In contrast to the behavior of the Cyclic and Acyclic types from earlier, the trail function returns *all* scopes in a path, including both the first and last; in this situation it's just as convenient either way, so the more precise choice is made.)

The True type is used to "squash" the type of all edges between two mod scopes into a type with zero or one values up to propositional equality, which is not a property that can be proven of the Propositional type itself in base Agda; this ensures that the WellFormedVar type can be shown to be finite. (The ⩽-import-edge field exists in this development only to encode part of a correctness specification and isn't mentioned anywhere else except in the code that constructs WellFormedVar expressions.) The edgeList function returns a list of type List ($\exists_2$ Edge), representing all of the edges in a Star path, and the _,?_ operator is defined in Code.Util with the type

```
∀ {A B} → Dec A → Dec B → Dec (A × B).

  record WellFormedVar (p : Star Edge s s′) : Set where
    constructor var
    field
      declIndex : x ∈ List.map proj₁ (decls (programGraph s′))
      ⩽1-import-edge :
        True
          (propositional?
            (filterProp^f
              (λ where ((a , b , _) , _) → modScope? a ,? modScope? b)
              (∃∈^f (edgeList p))))

  open WellFormedVar public
```

WellFormedVar is finite, since both of its fields are finite.

```
  WellFormedVar^f : (p : Star Edge s s′) → Listable (WellFormedVar p)
```

The specificity ordering over paths is defined directly in terms of the lengths of paths, as in the standard STLC variable shadowing rule: shorter variable resolution paths shadow longer paths.

```
  _<_ : ∃ (Star Edge s) → ∃ (Star Edge s) → Set
  (_ , p) < (_ , p′) = starLength p ℕ.< starLength p′
```

With these definitions, Code.Graph.Search can be opened with all of its parameters filled in. The first argument, the call to finiteScopeGraph, is the proof that the graph of program p is finite; the second argument, WellFormedVar^f ∘ proj$_2$, is the proof that

the predicate used in the predicate field of the MaximalPath type is finite at all indices; and <-isDecStrictPartialOrder is a lemma showing that the specificity ordering is a decidable strict partial ordering on paths. The MaximalPath and maximalPathsFrom^f definitions are renamed to represent that they denote variable resolution paths. The varPathType function is used in the definition of intrinsically-typed terms; the varType function referenced in its definition takes an index in the list of names bound in a scope and returns the type paired with the name at that index.

```
open import
  Code.Graph.Search
    (finiteScopeGraph (Program.Scope^f p))
    (WellFormedVar^f ∘ proj₂)
    <-isDecStrictPartialOrder
    renaming (MaximalPath to VarPath; maximalPathsFrom^f to varPathsFrom^f)
    public

varPathType : VarPath s → Type
varPathType = varType ∘ declIndex ∘ predicate
```

7.2 Scoped expressions

A question that arises when defining this style of intrinsically-typed typechecker in a scope graph framework is whether to opt into a kind of automatic implementation of ad-hoc type-directed overloading: if a typechecker is defined as going from raw terms to intrinsically-typed terms in a single pass, the type of all well-typed terms that erase to a raw input variable term will be a singleton type if there exists a *uniquely well-typed* resolution path for that variable, not just a unique resolution path in general. For example,

the term λ (x : nat). λ (x : bool). if x x (suc x) has uniquely well-typed resolution paths for each of the occurrences of the variable name x, but each occurrence has two distinct resolution paths before type invariants are enforced.

The definition of the Proc language in the next chapter opts into this overloading feature. The definition of STLC in this chapter makes the more conservative choice for the sake of demonstration: scopechecking and typechecking are implemented in separate passes, and a term is only valid if all of its variables are uniquely well-scoped *before* any type-related analysis, enforced by requiring the type of intrinsically-scoped terms output by the scopechecking procedure to be a singleton type.

The Code.STLC.Scoped module imports Code.STLC.Raw to bring the VarPath-related definitions into scope, hiding the Expr type in order to reuse that name for a type of intrinsically-scoped expressions and hiding some graph-related definitions for expressions in favor of using Scope for program scopes, brought into scope by opening Raw.Program below.

```
open import Code.STLC.Raw as Raw hiding (Expr; Scope; Scope^f; graph)
```

The module is then parameterized over an arbitrary raw program. The code in this module scopechecks terms against the graph generated for this program, the scopes in which are of type Raw.Program.Scope p.

```
module Code.STLC.Scoped (p : Raw.Program) where
open Raw.Program p
open ScopeGraph graph
open VarPath p
```

7.2.1 Expressions

The type of intrinsically-scoped expressions has the same structure as the type of intrinsically-typed STLC expressions from Bach Poulsen et al. [3], but does not enforce type invariants. The *shape* argument to the λt constructor witnesses that the scope $s\prime$ being associated with the lambda actually has the shape of a lambda scope, with a single parent scope and a single binding; an expression in the scope $s\prime$ paired with a proof of this shape invariant can be seen as a lambda expression in s, with the *shape* argument witnessing the binding structure in the graph linking the body scope to the lambda scope. (This encoding with the *shape* argument is from Bach Poulsen et al. [3] and is motivated by the demands of an intrinsically type-preserving interpreter, explained briefly in the Appendix to this thesis.)

```
data Expr s : Set where
  bool : Bool → Expr s
  nat : ℕ → Expr s
  suc pred iszero : Expr s → Expr s
  if : Expr s → Expr s → Expr s → Expr s
  var : VarPath x s → Expr s
  λt : {{shape : graph s′ ≡ ([ s ] , [ x , t ])}} → Expr s′ → Expr s
  _•_ : Expr s → Expr s → Expr s

variable e^s e^s₁ e^s₂ e^s₃ : Expr s
```

As mentioned earlier, the var constructor takes not just any arbitrary path but a VarPath guaranteed to be maximal and well-formed according to the defined specificity ordering and well-formedness predicate.

7.2.2 Erasures

The Erasure type is the type of proofs witnessing that some raw term is the erasure of some intrinsically-scoped term. All of the cases are fairly straightforward; the predicate could be defined in terms of the output of an erasure function from intrinsically-scoped terms to raw terms, but defining it as an inductive data type enables some convenient pattern-matching in the definition of the scopechecker.

```
data Erasure {s} : Raw.Expr → Expr s → Set where
  bool : Erasure (bool b) (bool b)
  nat : Erasure (nat n) (nat n)
  suc : Erasure e eˢ → Erasure (suc e) (suc eˢ)
  pred : Erasure e eˢ → Erasure (pred e) (pred eˢ)
  iszero : Erasure e eˢ → Erasure (iszero e) (iszero eˢ)
  if :
    Erasure e₁ eˢ₁ → Erasure e₂ eˢ₂ → Erasure e₃ eˢ₃ →
    Erasure (if e₁ e₂ e₃) (if eˢ₁ eˢ₂ eˢ₃)
  var : {p : VarPath x s} → Erasure (var x) (var p)
  ƛ :
    {{shape : graph s′ ≡ ([ s ] , [ x , t ])}} → Erasure e eˢ →
    Erasure (ƛ x t e) (ƛ {{shape}} eˢ)
  _•_ : Erasure e₁ eˢ₁ → Erasure e₂ eˢ₂ → Erasure (e₁ • e₂) (eˢ₁ • eˢ₂)
```

A verified erasure function is fairly trivial to implement, along with a proof that any two raw terms that are the erasure of the same typed term must be propositionally equal.

```
erase : (eˢ : Expr s) → ∃ λ e → Erasure e eˢ
erased-unique : Erasure e₁ eˢ → Erasure e₂ eˢ → e₁ ≡ e₂
```

7.2.3 Scopechecking

The ∃Erasure^f function serves as a scopechecker: it takes in a scope *s* in the program graph and a raw expression *e*, and returns all of the intrinsically-scoped expressions that scopecheck in *s* and erase to *e*. (As a reminder, the <&^{f}> function is from Subsection 3.3.2, and surjection is a standard library function described in the same section.)

```
∃Erasure^f : ∀ s e → Listable (∃ λ (e^s : Expr s) → Erasure e e^s)
```

The cases for constants and suc are relatively straightforward: there is exactly one intrinsically-scoped term that erases to any given constant, and the set of intrinsically-scoped terms that erase to suc *e* are in one-to-one correspondence with the set of intrinsically-scoped terms that erase to *e*.

```
∃Erasure^f s (bool b) = finite [ -, bool ] λ where (_ , bool) → here refl
∃Erasure^f s (nat n) = finite [ -, nat ] λ where (_ , nat) → here refl
∃Erasure^f s (suc e) =
  ∃Erasure^f s e <&^f>
    surjection
      _
      (λ where (_ , suc er) → -, er)
      (λ where (_ , suc er) → refl)
```

There are two interesting applications of arguments resolved by unification in the code above: the specific value of a constant bool or nat input term (*b* or *n*) is never referenced on the right-hand side of the definition, and one direction of the surjection in the suc case is inferred by unification, namely the function that is actually mapped over the elements list of erasure^f *s e* at runtime when this scopechecking procedure

is executed. Intuitively, in the bool and nat cases the type of the right-hand side is "intrinsically-scoped terms that erase to the given constant value", so the Erasure.bool and Erasure.nat constructors used in the elements lists and the lambda argument patterns have their implicit arguments uniquely resolved by unification to the constants *b* and *n* without explicit specification; in the suc case, the third argument to surjection tells the unification engine that the first argument function must return an expression definitionally equal to the expression returned by the second argument function, and η-expansion of functions and the _,_ constructor along with the definitional injectivity of the suc constructor enable unification to fill in the entire function definition.

The effect of this style of implicit programming is that all of the explicit code in the three cases above is (hopefully somewhat readable) *proof* code, and all of the *program* code that executes at runtime is filled in by unification; the definition of $\exists\text{Erasure}^f$ reads as a proof that the output type is finite by virtue of surjections to other types known to be finite and computes as a procedure producing lists of scoped terms by mapping and filtering over other lists. The consistency of the pattern also indicates a possibility for future work to fill in most or all of this definition automatically with generic programming given the definitions for the expression and erasure types as input.

This same pattern used in the suc case is used in the pred, iszero, if, and _•_ cases. To save space, only the if case is listed here.

```
∃Erasure^f s (if e₁ e₂ e₃) =
  ∃Erasure^f s e₁ ×^f ∃Erasure^f s e₂ ×^f ∃Erasure^f s e₃ <&^f>
    surjection
      _
      (λ where (_ , if er₁ er₂ er₃) → (-, er₁) , (-, er₂) , (-, er₃))
      (λ where (_ , if er₁ er₂ er₃) → refl)
```

The variable case is similar: the type of intrinsically-scoped variable terms that erase to a given raw variable is isomorphic to the type of VarPath values that resolve the name of the raw variable.

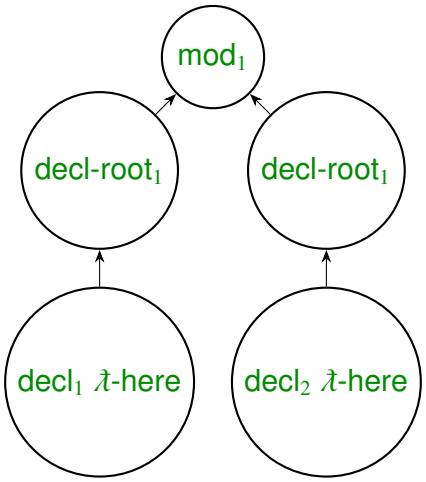
```
∃Erasure^f s (var x) =
  varPathsFrom^f x s <&^f>
    surjection
      _
      (λ where (var p , var) → p)
      (λ where (var p , var) → refl)
```

The lambda case has to generate only terms with a valid *shape* argument, so it filters the type of all scopes by a predicate matching only scopes with the correct shape of ScopeData and then generates the type of all suitable intrinsically-typed body terms that scopecheck in any of those filtered scopes.

```
∃Erasure^f s (ƛ x t e) =
  Σ^f
    (filterProp^f (λ s′ → graph s′ ≟ ([ s ] , [ x , t ])) Scope^f)
    (λ where (s′ , _) → ∃Erasure^f s′ e) <&^f>
      surjection
        _
        (λ where (_ , ƛ {{shape}} er) → (-, shape) , (-, er))
        (λ where (_ , ƛ er) → refl)
```

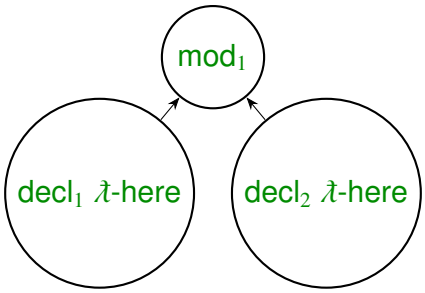
The lambda case illustrates the purpose of the -root nodes in the scope graphs generated by Program.graph: since the static semantics of this language require that each valid term is uniquely well-scoped, the root node of each term's scope graph needs

to be unique so that there will be only one scope that matches the predicate in the λ case for the outermost lambda of a term. For example, consider the scope graph below for a module that contains the terms $\lambda\ (x : \mathsf{nat}).\ x$ and $\lambda\ (x : \mathsf{nat}).\ \mathsf{suc}\ x$.



The first term is scopechecked in the root scope decl-root$_1$ (i.e. the scope index in the type of the intrinsically-scoped output term is decl-root$_1$), and the second is scopechecked in decl-root$_2$. When scopechecking either term, there is a unique scope in the graph that has the correct parent and binds the correct name and type: decl-root$_1$ and decl-root$_2$ both bind an argument named x with type nat, but they cannot occur together in the output of the call to filterProp^f in the λ case of $\exists$Erasure^f, since the s argument to $\exists$Erasure^f will be different in each case.

In contrast, consider the scopechecking of this module in a scope graph without decl-root nodes, where the outermost lambda scopes of each term point directly to the module scope and the overall expression is scopechecked in the module scope.



Under this scope graph, the λ cases in $\exists\text{Erasure}^f$ for each term both find that there are two scopes with the correct contents (decl$_1$ λ-here and decl$_2$ λ-here), and therefore both return ambiguous output with two distinct intrinsically-scoped terms that erase to each corresponding raw term. The issue is that the scopechecker is designed to scopecheck any arbitrary term in any arbitrary graph, so it does not have enough information to know that a given term under consideration is e.g. "the first term in the first module in the program".

This scheme with the -root nodes is admittedly a bit of a concession of computational efficiency for the sake of proof convenience; the execution of a realistic scopechecker should ideally not depend on the linear-time operation of filtering a list of all scopes in order to find a scope suitable to represent a particular lambda subterm. A potentially more principled solution would be to introduce an annotation phase before the scopechecker, annotating the lambda nodes in a raw expression with their scopes in the overall program graph so that the scopechecker knows which scopes are associated with each of them. Since the module in question does scopecheck uniquely in a graph with decl$_1$ λ-here and decl$_1$ λ-here unified into the same node with a single edge to mod$_1$, another potential solution might be to unify the subgraphs in the scope graph that have identical shapes and bindings, creating a "minimal" graph in some sense for the given module before scopechecking the module.

7.3 Typed expressions

The module defining intrinsically-typed expressions is also parameterized over a raw program; all the code in this section is defined within this module.

```
open import Code.STLC.Raw as Raw hiding (Expr; Scope; graph)

module Code.STLC.Typed (p : Raw.Program) where
open Raw.Program p
open ScopeGraph graph
open VarPath p
```

7.3.1 Expressions

The type of expressions is almost identical to the intrinsically-typed STLC syntax type in Bach Poulsen et al. [3], with the exception again of the VarPath argument in the var case.

```
data Expr s : Type → Set where
  bool : Bool → Expr s bool
  nat : ℕ → Expr s nat
  suc pred : Expr s nat → Expr s nat
  iszero : Expr s nat → Expr s bool
  if : Expr s bool → Expr s t → Expr s t → Expr s t
  var : (p : VarPath x s) → Expr s (varPathType x p)
  ƛ : {{shape : graph s′ ≡ ([ s ] , [ x , t₁ ])}} → Expr s′ t₂ → Expr s (t₁ ⇒ t₂)
  _•_ : Expr s (t₁ ⇒ t₂) → Expr s t₁ → Expr s t₂

variable e^t e^t₁ e^t₂ e^t₃ : Expr s t
```

7.3.2 Erasures

The erasure type is almost identical to the one in the Scoped module, with the exception of the slightly modified var and λ cases and the type indices that must be specified for the intrinsically-typed terms.

```
data Erasure {s} : Scoped.Expr s → Expr s t → Set where
  bool : Erasure (bool b) (bool b)
  nat : Erasure (nat n) (nat n)
  suc : Erasure e^s e^t → Erasure (suc e^s) (suc e^t)
  pred : Erasure e^s e^t → Erasure (pred e^s) (pred e^t)
  iszero : Erasure e^s e^t → Erasure (iszero e^s) (iszero e^t)
  if :
    Erasure e^s_1 e^t_1 → Erasure e^s_2 e^t_2 → Erasure e^s_3 e^t_3 →
    Erasure (if e^s_1 e^s_2 e^s_3) (if e^t_1 e^t_2 e^t_3)
  var : {p : VarPath x s} → Erasure (var p) (var p)
  λ :
    {{shape : graph s′ ≡ ([ s ] , [ x , t_1 ])}} → Erasure e^s e^t →
    Erasure (λ {{shape}} e^s) (λ e^t)
  _•_ : Erasure e^s_1 e^t_1 → Erasure e^s_2 e^t_2 → Erasure (e^s_1 • e^s_2) (e^t_1 • e^t_2)
```

The presence of the STLC type index in the erasure type makes it a little more challenging to specify the uniqueness properties of this erasure relation than those of the scoped one. erased-unique witnesses that any pair of scoped terms that are the erasure of the same typed term must be propositionally equal, pre-erased-type-unique witnesses that any two typed terms that erase to the same scoped term must have the same type, pre-erased-unique witnesses that two typed terms with the same type that

erase to the same scoped term must be propositionally equal, and erasure-unique-≅ combines the previous two into a proof that any two typed terms of any types that erase to the same scoped term must be heterogeneously equal. Finally, the Erasure type itself is shown to be propositional by an application of some of these lemmas.

```
erase : (e^t : Expr s t) → ∃ λ e^s → Erasure e^s e^t
erased-unique : Erasure e^s₁ e^t → Erasure e^s₂ e^t → e^s₁ ≡ e^s₂
pre-erased-type-unique :
  {e^t₁ : Expr s t₁} {e^t₂ : Expr s t₂} →
  Erasure e^s e^t₁ → Erasure e^s e^t₂ → t₁ ≡ t₂
pre-erased-unique : Erasure e^s e^t₁ → Erasure e^s e^t₂ → e^t₁ ≡ e^t₂
erasure-unique-≅ :
  {e^t₁ : Expr s t₁} {e^t₂ : Expr s t₂}
  (er : Erasure e^s e^t₁) (er′ : Erasure e^s e^t₂) → er ≅ er′
instance erasure-propositional : Propositional (Erasure e^s e^t)
```

7.3.3 Typechecking

Since there is at most one intrinsically-typed term that erases to a given intrinsically-scoped term (by pre-erased-type-unique and pre-erased-unique), a typechecker may be appropriately defined as a decision procedure; any ambiguity in the meaning of explicitly-typed STLC terms arises from scoping ambiguities, not typing ambiguities. The typechecker could be defined as a function returning a Dec value along with a proof that the type it returns is Propositional, in order to use it with filterProp^f to filter the scopechecker output.

Instead, it turns out to be more convenient for the recursive procedure defining the

typechecker to return a Finite type of intrinsically-typed terms that erase to a given intrinsically-scoped input term. This is partly because it's not possible to give an equivalent of the Σ^f function for Dec without requiring additional proof arguments: knowing that some element $a : A$ exists and that the type $B\ a$ is empty is not enough to know whether $\Sigma\ A\ B$ is inhabited. There are types with decidable inhabitance that are not finite (trivially including $\mathbb{N}$), so this is not a fully general technique for building decision procedures, but in situations where the output of a decision procedure is a type that can be given a Finite proof, it can be simpler to build the procedure using the somewhat more expressive Finite combinators. The values of the type $\exists_2\ \lambda\ t\ (e^t : \text{Expr}\ s\ t) \to \text{Erasure}\ e^s\ e^t$ can be seen as witnesses that the intrinsically-scoped expression e^s has a valid typing derivation, in the sense that they witness the existence of an intrinsically-typed term that erases to e^s.

```
∃Erasure^f : (e^s : Scoped.Expr s) → Listable (∃₂ λ t (e^t : Expr s t) → Erasure e^s e^t)
```

As in scopechecking, typechecking a constant expression is immediate.

```
∃Erasure^f (bool b) = finite [ -, -, bool ] λ where (_ , _ , bool) → here refl
∃Erasure^f (nat b) = finite [ -, -, nat ] λ where (_ , _ , nat) → here refl
```

The suc case filters the type of all terms that erase to e with a predicate requiring the inferred type to be nat.

```
∃Erasure^f (suc e) =
  filterProp^f (λ where (t , _ , _) → t ≟ nat) (∃Erasure^f e) <&^f>
    surjection
      (λ where ((_ , _ , er) , refl) → -, -, suc er)
      (λ where (_ , _ , suc er) → (-, -, er) , refl)
      (λ where (_ , _ , suc er) → refl)
```

The pred and iszero cases are omitted to save space, since they're nearly identical to the suc case.

The if case is similar to the suc case, although it has to check more invariants: e_1 must be of type bool, and e_2 and e_3 must both be well-typed and have the same type.

```
∃Erasure^f (if e₁ e₂ e₃) =
  filterProp^f (λ where (t₁ , _ , _) → t₁ ≟ bool) (∃Erasure^f e₁) ×^f
    filterProp^f
      (λ where ((t₂ , _ , _) , (t₃ , _ , _)) → t₂ ≟ t₃)
      (∃Erasure^f e₂ ×^f ∃Erasure^f e₃) <&^f>
        surjection
          (λ where
            (((_ , _ , er₁) , refl) , (((_ , _ , er₂) , (_ , _ , er₃)) , refl)) →
              -, -, if er₁ er₂ er₃)
          (λ where
            (_ , _ , if er₁ er₂ er₃) →
              ((-, -, er₁) , refl) , (((-, -, er₂) , (-, -, er₃)) , refl))
          (λ where (_ , _ , if er₁ er₂ er₃) → refl)
```

The var case is trivial: all of the work of variable resolution has been done during scopechecking, and all of the work of validating the types of variables occurs in the cases for the constructs that require typechecking subterms like suc and _•_.

```
∃Erasure^f (var p) = finite [ -, -, var ] λ where (_ , _ , var) → here refl
```

The λ case is straightforward: a lambda expression is well-typed iff its body is well-typed. The *shape* argument is passed silently from the input to the output by instance resolution.

```
∃Erasure^f (ƛ e) =
  ∃Erasure^f e <&^f>
    surjection
      _
      (λ where (_ , _ , ƛ er) → -, -, er)
      (λ where (_ , _ , ƛ er) → refl)
```

The application case checks that the type of the right subexpression matches up with the type of the input to the left subexpression.

```
∃Erasure^f (e₁ • e₂) =
  filterProp^f
    (λ where ((t₁ , _ , _) , (t₂ , _ , _)) → acceptsArg? t₁ t₂)
    (∃Erasure^f e₁ ×^f ∃Erasure^f e₂) <&^f>
      surjection
        (λ where (((_ , _ , er₁) , (_ , _ , er₂)) , (_ , refl)) → -, -, er₁ • er₂)
        (λ where (_ , _ , er₁ • er₂) → ((-, -, er₁) , (-, -, er₂)) , (-, refl))
        (λ where (_ , _ , er₁ • er₂) → refl)
```

The typecheck decision procedure is defined with Finite.dec, which looks only to see if the elements list has a head, so in runtime execution the typecheck function lazily returns after the first typing derivation is found. This is justified by ∃erasure-propositional, which witnesses that the output of typecheck in the positive case is unique.

```
typecheck : (e^s : Scoped.Expr s) → Dec (∃₂ λ t (e^t : Expr s t) → Erasure e^s e^t)
typecheck = Listable.dec ∘ ∃Erasure^f

instance
```

```
    ∃erasure-propositional :
      Propositional (∃₂ λ t (e[t] : Expr s t) → Erasure e[s] e[t])
```

The _⊢_⦂_ predicate witnesses that a raw expression *e* is uniquely well-scoped and well-typed with type *t* in scope *s*. The type of the scopedErasure field ensures that *e* has no variables that resolve ambiguously; as mentioned in the introduction to section 6.2, this disables type-directed overloading in cases where a raw term has more than one distinct corresponding intrinsically-scoped term but only one is well-typed.

```
  record _⊢_⦂_ s e t : Set where
    constructor typing
    field
      scopedErasure : True (singleton? (Scoped.∃Erasure[f] s e))

    scopedExpr = proj₁ (point (toWitness scopedErasure))

    field
      {typedExpr} : Expr s t
      typedErasure : Erasure scopedExpr typedExpr
```

This type is propositional and finite: since it requires the intrinsically-scoped term that it contains to be unique and an intrinsically-typed term that erases to a given intrinsically-scoped term is always unique, there is either zero or one member for at given indices.

```
  _⊢[f]_⦂_ : ∀ s e t → Listable (s ⊢ e ⦂ t)
  instance typing-propositional : ∀ {s e t} → Propositional (s ⊢ e ⦂ t)
```

These definitions are used to specialize the TypedModuleSystem module, completing the definition of the static semantics of this language.

```
open TypedModuleSystem p _⊨_ _⊢ᶠ_∶_ public
```

7.4 Program example

The test program below is defined in Code.STLC.Test, and the code accompanying this thesis includes a main function to test the static analysis and evaluation procedures over this test program (and a `README` file with compilation and execution instructions). A couple of features of the language definition are on display: the "`two`" and "`three`" modules import each other in a cycle, the definitions of "`even`" and "`odd`" are mutually recursive across different modules, and both "`two`" and "`three`" must import "`one`" to use the "¬" function because module imports are not transitive (if they were then only one of them would need to). The code takes advantage of Raw.Expr instances of the Number and IsString typeclasses to convert natural number literals and string literals to raw expressions with the nat and var constructors respectively. The program passes scopechecking and typechecking with a unique output intrinsically-typed term that evaluates to the value bool true, which prints as `true` in the Agda main function's execution.

```
test : Program
test =
  record
    { modules =
        ( "one"
        , record
            { imports = []
            ; declarations =
```

```
        [ ("¬" , bool ⇒ bool ,
              ƛ "x" bool (if "x" (bool false) (bool true)))
        ]
    }
) ::
( "two"
, record
    { imports = "one" :: "three" :: []
    ; declarations =
        [ ("even" , nat ⇒ bool ,
            ƛ "x" nat
              (if
                (iszero "x")
                (bool true)
                ("¬" • ("odd" • pred "x"))))
        ]
    }
) ::
( "three"
, record
    { imports = "one" :: "two" :: []
    ; declarations =
        [ ("odd" , nat ⇒ bool ,
            ƛ "x" nat
              (if
```

```
                (iszero "x")
                (bool false)
                ("¬" • ("even" • pred "x"))))
            ]
        }
    ) ::
    []
; mainImports = [ "two" ]
; mainType = bool
; mainTerm = "even" • 2
}
```

Chapter 8 Toy Procedural Language

The toy procedural language defined in this chapter includes imperative assignments and for loops along with pointer, array, boolean, and integer types. As mentioned in the introduction to Section 6.2, a flexible form of type-directed overloading is enabled, allowing some of the language definition to be streamlined. The implementation is described here in less detail than that of STLC in the previous chapter, referencing patterns from the STLC definition to avoid repeating their explanations except where they differ notably in the application to a different language.

The procedures in this language are not first-class, and can only be referenced by name. The definitions used as "terms" when instantiating the module system are a sum type of procedures and expressions; an expression defined at the top-level in a module acts as a module-scoped variable with a default value given by the expression.

8.1 Raw terms

8.1.1 Types

The syntax for types is taken roughly from C: the prefix asterisk represents a pointer type and the postfix square brackets represent an array type, defined as two distinct types. Arrays can be indexed with integers but arithmetic over pointers is not supported; pointers are obtained through a variable de-referencing operator comparable to the unary & operator in C.

```
infix 17 *_
infix 18 _[[]]
data Type : Set where
```

```
unit bool int : Type
_⟦⟧ *_ : Type → Type

variable t t₁ t₂ : Type
```

8.1.2 Expressions

The implicit parameter of type Size (from the standard library Size module) in the definition of Expr below is only to aid termination checking: specifically, the function application operator _⟨_⟩ accepts a List argument of Expr values, and a recursive call in a structurally recursive function cannot, in general, be used as the argument to a higher-order function like List.map over this list of subexpressions without some kind of additional termination evidence, even though the expressions in the list are actually structurally smaller than the argument pattern they come from. The ↑ function returns a Size value definitionally larger than its argument, so the implicit Size arguments effectively act as an index representing the depth of an AST value. The `--sized-types` compiler option fills in implicit Size arguments with a special value ∞ rather than filling them in by unification, allowing implicit Size arguments to be effectively ignored in uses of the type that don't require them for termination reasoning. The variable syntax is not quite as flexible with terms that involve Size arguments, so most implicit arguments are written out in the code in this section.

```
infix 13 _⟨_⟩
infix 14 _<=_
infix 15 -_
infixl 15 _+_
infix 16 _⟦_⟧
```

```
infix 17 &_
data Expr : {l : Size} → Set where
  unit : ∀ {l} → Expr {l}
  null : ∀ {l} → Type → Expr {l}
  _[[_]] : ∀ {l} → Expr {l} → Expr {l} → Expr {l}
  &_ : ∀ {l} → Name → Expr {l}
  bool : ∀ {l} → Bool → Expr {l}
  int : ∀ {l} → ℤ → Expr {l}
  new_[[_]] : ∀ {l} → Type → Expr {l} → Expr {l}
  -_ *_ : ∀ {l} → Expr {l} → Expr {l}
  length : ∀ {l} → Expr {l} → Expr {l}
  _+_ _<=_ : ∀ {l} → Expr {l} → Expr {l} → Expr {l}
  _⟨_⟩ : ∀ {l} → Name → List (Expr {l}) → Expr {↑ l}

variable
  e e₁ e₂ : Expr
```

The &_ constructor turns a name into a pointer to the data cell for the variable with that name; the *_ constructor accesses the value at the data cell specified by a pointer-typed expression; the _[[_]] constructor creates a pointer to a specific index in an array, given a pointer to the array and an expression for the index; the new_[[_]] constructor creates a new array of the given type with length specified by the given Expr, with all cells initialized with a type-specific default value; and the _⟨_⟩ constructor represents a procedure call expression, taking the name of a procedure and a list of argument expressions.

8.1.3 Statements

The Stmt type represents statements. The void_ statement evaluates an expression and ignores the resulting value, to allow calling procedures for only their side effects; the assignment statement $x := y$ is like $*x = y$ in C, assigning the value of y to the data cell pointed to by x; the struck-through semicolon _⨾_ is the sequencing operator (underlined instead in the Agda source code due to typesetting differences); the let operator binds the given expression value in the given substatement with the given name and type; and the for and if operators work as in C-like languages, with the second Expr argument and both Stmt arguments to for typechecked in a scope that includes the variable named by the first argument.

```
infixr 10 _⨾_
infixr 11 void_ for⟨_:=_⨾_⨾_⟩_ if⟨_⟩_ let⟨_∘_:=_⟩_
infix 12 _:=_
data Stmt : Set where
  void_ : Expr → Stmt
  _:=_ : Expr → Expr → Stmt
  _⨾_ : Stmt → Stmt → Stmt
  let⟨_∘_:=_⟩_ : Name → Type → Expr → Stmt → Stmt
  for⟨_:=_⨾_⨾_⟩_ : Name → Expr → Expr → Stmt → Stmt → Stmt
  if⟨_⟩_ : Expr → Stmt → Stmt

variable st st₁ st₂ : Stmt
```

8.1.4 Procedures

The Proc type represents procedures, each containing an argument list, a body statement, and a return expression.

```
record Proc : Set where
  constructor proc
  field
    params : List (Name × Type)
    body : Stmt
    ret : Expr

open Proc

variable pr : Proc
```

8.1.5 Declarations

The Decl type is used as the instantiation of *Term* in the module system. A declaration is either a procedure or a variable with a given initial expression.

```
data DeclType : Set where
  proc : List Type → Type → DeclType
  var : Type → DeclType

variable dt : DeclType

data Decl : Set where
  proc : Proc → Decl
```

```
    var : Expr → Decl

  variable d : Decl
```

8.1.6 Scope graph construction

There are no expression forms in the Proc language that introduce new scopes. The type of scopes in a statement are defined similarly to the Scope type in the STLC definition in the previous chapter, with the let and for constructors introducing new scopes; the proc-outer scope is used to bind the arguments of a procedure.

```
  data StmtScope : Stmt → Set where
    ;-left : ∀ {st₁ st₂} → StmtScope st₁ → StmtScope (st₁ ; st₂)
    ;-right : ∀ {st₁ st₂} → StmtScope st₂ → StmtScope (st₁ ; st₂)
    let-here : ∀ {x t e st} → StmtScope (let⟨ x ∶ t := e ⟩ st)
    let-there : ∀ {x t e st} → StmtScope st → StmtScope (let⟨ x ∶ t := e ⟩ st)
    for-here : ∀ {x e₁ e₂ st₁ st₂} → StmtScope (for⟨ x := e₁ ; e₂ ; st₁ ⟩ st₂)
    for-step : ∀
      {x e₁ e₂ st₁ st₂} →
      StmtScope st₁ → StmtScope (for⟨ x := e₁ ; e₂ ; st₁ ⟩ st₂)
    for-there : ∀
      {x e₁ e₂ st₁ st₂} →
      StmtScope st₂ → StmtScope (for⟨ x := e₁ ; e₂ ; st₁ ⟩ st₂)
    if-there : ∀ {e st} → StmtScope st → StmtScope (if⟨ e ⟩ st)

  data ProcScope (p : Proc) : Set where
    proc-outer : ProcScope p
```

```
    proc-body : StmtScope (body p) → ProcScope p

  data DeclScope : Decl → Set where

    proc : ∀ {p} → ProcScope p → DeclScope (proc p)
```

The graph of a statement or procedure is also generated with a similar pattern as in the STLC definition; the free pre-scope is referenced as a parent in the let-here, for-here, and proc-outer scopes, since these are the forms that introduce new scopes.

```
  stmtGraph : ∀ {s} → PreGraph (StmtScope s) Type
  stmtGraph (;-left i) = mapEdges (mapPreScope ;-left) (stmtGraph i)
  stmtGraph (;-right i) = mapEdges (mapPreScope ;-right) (stmtGraph i)
  stmtGraph {let⟨ x ⦂ t := e ⟩ s} let-here = [ free ] , [ x , t ]
  stmtGraph (let-there i) =
    mapEdges (bound ∘ bind let-here let-there) (stmtGraph i)
  stmtGraph {for⟨ x := e₁ ; e₂ ; s₁ ⟩ s₂} for-here =
    [ free ] , [ x , int ]
  stmtGraph (for-step i) =
    mapEdges (bound ∘ bind for-here for-step) (stmtGraph i)
  stmtGraph (for-there i) =
    mapEdges (bound ∘ bind for-here for-there) (stmtGraph i)
  stmtGraph (if-there i) =
    mapEdges (mapPreScope if-there) (stmtGraph i)

  procGraph : ∀ {p} → PreGraph (ProcScope p) Type
  procGraph {proc xts s e} proc-outer = [ free ] , xts
  procGraph (proc-body i) =
    mapEdges (bound ∘ bind proc-outer proc-body) (stmtGraph i)
```

There are no expression forms that introduce new scopes, so all scopes in the graph of a declaration must be procedure scopes.

```
declGraph : ∀ d → PreGraph (DeclScope d) DeclType
declGraph d (proc i) =
  let es , ds = procGraph i in
    List.map (mapPreScope proc) es , List.map (Σ.map₂ var) ds
```

All of the scope types involved in the graph are finite, so this raw language definition is sufficient to specialize RawModuleSystem with.

```
StmtScope^f : ∀ st → Listable (StmtScope st)
ProcScope^f : ∀ pr → Listable (ProcScope pr)
DeclScope^f : ∀ d → Listable (DeclScope d)

open RawModuleSystem DeclType Decl DeclScope^f declGraph public
```

8.1.7 Specificity ordering and well-formedness predicates

The well-formedness predicates are defined similar to the STLC one, but in this case there are two relevant predicates: one for resolving variable names that refer to data cells in memory and one for resolving procedure names in procedure call expressions.

```
record WellFormedVar x i : Set where
  constructor wellFormedVar
  field
    {type} : Type
    declIndex : (x , var type) ∈ decls (g i)
```

```
    record WellFormedProc x i : Set where
      constructor wellFormedProc
      field
        {paramTypes} : List Type
        {retType} : Type
        declIndex : (x , proc paramTypes retType) ∈ decls (g i)
```

Neither predicate imposes any constraints on the paths used to resolve names, leaving module imports implicitly transitive. The specificity ordering prioritizes the resolution paths with the fewest import edges and then the shortest paths in general, using a type Lex from Code.Util that encodes a lexical ordering relation over pairs of natural numbers.

```
    importEdges : ∀ {i j} → Star Edge i j → ℕ

    _<_ : ∀ {i} → ∃ (Star Edge i) → ∃ (Star Edge i) → Set
    (_ , p) < (_ , p′) =
      Lex (importEdges p , starLength p) (importEdges p′ , starLength p′)
```

The types VarPath and ProcPath are derived from these predicates and this ordering, in the same way as VarPath in the STLC definition (but one for each predicate).

8.2 Typed terms

As in the STLC definition, the module defining typed terms in this language is parameterized over a raw program. The Scoped step from STLC is skipped altogether, since in this language terms are defined to be unambiguous whenever they have a uniquely well-typed interpretation, not necessarily a uniquely well-scoped interpretation; typechecking goes directly from raw terms to intrinsically-typed terms.

```
module Code.Proc.Typed (p : Raw.Program) where
open Raw.Program p
open VarPath graph
open ScopeGraph graph
```

8.2.1 Expressions and statements

The intrinsically-typed expression and statement types follow effectively the same pattern as in the STLC definition. Of note is that the _⟨_⟩ constructor takes an All argument, requiring its argument sublist to have the correct length and types for the procedure being called. The VarPath and ProcPath types are the path types defined with the specificity ordering and the respective well-formedness predicates from the previous section, and the varType, paramTypes, and retType functions extract the corresponding component types from the destination of a variable or procedure name resolution path. The var constructor of DeclType, used earlier to represent module-scope variables, here also represents variables bound by statement forms and procedure parameters.

```
data Expr (s : Scope) : {l : Size} → Type → Set where
  unit : ∀ {l} → Expr s {l} unit
  null : ∀ {l t} → Expr s {l} (* t)
  &_ : ∀ {l x} (p : VarPath x s) → Expr s {l} (* varType p)
  _[[_]] : ∀ {l t} → Expr s {l} (* (t [[]])) → Expr s {l} int → Expr s {l} (* t)
  bool : ∀ {l} → Bool → Expr s {l} bool
  int : ∀ {l} → ℤ → Expr s {l} int
  new_[[_]] : ∀ {l} (t : Type) → Expr s {l} int → Expr s {l} (t [[]])
  *_ : ∀ {l t} → Expr s {l} (* t) → Expr s {l} t
```

```
    -_ : ∀ {l} → Expr s {l} int → Expr s {l} int
    _+_ : ∀ {l} → Expr s {l} int → Expr s {l} int → Expr s {l} int
    _<=_ : ∀ {l} → Expr s {l} int → Expr s {l} int → Expr s {l} bool
    length : ∀ {l t} → Expr s {l} (t [[]]) → Expr s {l} int
    _⟨_⟩ : ∀
      {l x} (p : ProcPath x s) →
      All (Expr s {l}) (paramTypes p) → Expr s {↑ l} (retType p)

  data Stmt (s : Scope) : Set where
    void_ : ∀ {t} → Expr s t → Stmt s
    _:=_ : ∀ {t} → Expr s (* t) → Expr s t → Stmt s
    _;_ : Stmt s → Stmt s → Stmt s
    let⟨_∘_:=_⟩_ : ∀
      {i i′} x t {{_ : graph i′ ≡ ([ s ] , [ x , var t ])}} →
      Expr s t → Stmt i′ → Stmt s
    for⟨_:=_;_;_⟩_ : ∀
      {i i′} x {{_ : graph i′ ≡ ([ s ] , [ x , var int ])}} →
      Expr s int → Expr i′ bool → Stmt i′ → Stmt i′ → Stmt s
    if⟨_⟩_ : Expr s bool → Stmt s → Stmt s
```

8.2.2 Procedures

Intrinsically-typed procedures carry a scope and a witness to the shape of the scope; this is the proc-outer scope in the graphs generated in the previous section.

```
  record Proc (i : Scope) (xts : List (Name × Type)) (t : Type) : Set where
    constructor proc
```

```
    field
      {scope} : Scope
      {{shape}} : graph scope ≡ ([ i ] , List.map (Σ.map₂ var) xts)
      body : Stmt scope
      ret : Expr scope t
```

The Erasure types and Finite proofs for the Ptr, Expr, Stmt, Proc, and Decl types are defined along the same pattern as those for the Expr type in the STLC definition. One somewhat nontrivial constructor case is that of the procedure call expression, listed below. The Pointwise type is used to relate the parameter type list of the procedure to the list of argument expressions by requiring that there exists an intrinsically-typed expression for each pair of type and raw expression (and implicitly that the lists are the same length).

```
    -⟨_⟩ : ∀
      {l x} {p : ProcPath x s} {es : List (Raw.Expr {l})} →
      (es′ :
        Pointwise
          (λ t e → ∃ λ (e′ : Expr s t) → ExprErasure e e′)
          (paramTypes p) es) →
      ExprErasure (x ⟨ es ⟩) (p ⟨ pointwise⇒all₁ (Pointwise.map proj₁ es′) ⟩)
```

The pointwise⇒all$_1$ function (from Code.Util) is used to convert a Pointwise value to an All value when the binary predicate used ignores its second argument. The use in the type of -⟨_⟩ extracts the list of intrinsically-typed expression proj$_1$ components from the *es′* argument to get the All list of intrinsically-typed arguments for the intrinsically-typed procedure call expression.

```
pointwise⇒all₁ : ∀
  {A B} {P : A → Set} {as : List A} {bs : List B} →
  Pointwise (λ a _ → P a) as bs →
  All P as
```

8.2.3 Typechecking

The Finite proofs for the erasure types establish a typechecking procedure for the various syntactic sorts in this language; since no special uniqueness invariants are being enforced, the Typing type for a raw declaration *d* just contains an intrinsically-typed declaration of type *t* that erases to *d*.

```
record Typing s d t : Set where
  constructor typing
  field
    typedDecl : Decl s t
    erasure : DeclErasure d typedDecl

Typing^f : ∀ s d t → Listable (Typing s d t)
```

With these definitions the TypedModuleSystem module can be opened and specialized, completing the definition of the static semantics of this language.

```
open TypedModuleSystem p DeclVal Typing^f public
```

8.3 Program example

A complete example program in the language of this chapter is presented on the next page. To simplify comprehension, mixfix operators and Agda's built-in features for

overloading certain syntax are abused below to define a Pascal-like syntax with begin and end delimiters for the module language. In a sense, the definitions of these operators represent the definition of a syntax analysis phase for the Proc language, delegating all of the work of lexing and parsing to Agda. In exchange, of course, the object language syntax can only be defined in terms of mixfix operators and overloadable Agda syntax, rather than any kind of direct definition in terms of a formal grammar or a bespoke parser, and Agda syntax errors are the only form of error reporting

```
_»_ = _⨾_
program xms main-imports xs main⦂ t := e end = prog xms xs (var t) (var e)
mod x imports xs begin es end = x , mod xs es
procedure x ⟨ xts ⟩ t begin s return e end =
  x , proc (List.map proj₂ xts) t , proc (proc xts s e)
procedure_⟨_⟩begin_end = procedure_⟨_⟩ unit begin_return unit end
var x ⦂ t := e = x , var t , var e
_of_ = _,′_
```

The do syntax in Agda desugars to whatever operators with the relevant names are in scope, similar to the `RebindableSyntax` Haskell extension. In this case the _»_ operator is defined to be the statement sequencing operator, allowing do syntax to be used to sequence statements in raw Stmt definitions, as in several of the definitions in test below.

The program test creates a buffer array, initializes it with 25 integers in descending order, and then calls an in-place quicksort procedure to sort the buffer before returning its contents; an unused secondary declaration with the same name as the buffer is given with a different type to demonstrate type-directed overloading. The program is translated from pseudocode on the Wikipedia page for quicksort [14], specifically the

pseudocode in the “Lomuto partition scheme” section. The test code is a more or less direct translation of the pseudocode in the style of a standard C implementation. While this style of implementation of object language syntax is probably not suitable for the user-facing frontend of a real-world language implementation, it does make it relatively straightforward to verify by eye that the implementation of test matches the pseudocode, which is a useful quality for a test to have.

```
test : Program
test =
  program
    mod "code" imports [] begin
      procedure "swapint" ⟨ "a" of * int :: "b" of * int :: [] ⟩begin
        let⟨ "temp" ◦ int := * "b" ⟩ (do
          "b" := * "a"
          "a" := "temp"
        )
      end ::
      procedure "quicksort"
        ⟨ "A" of * (int ⟦⟧) :: "low" of int :: "high" of int :: [] ⟩begin
        if⟨ "low" + 1 <= "high" ⟩
          let⟨ "p" ◦ int := "partition" ⟨ "A" :: "low" :: "high" :: [] ⟩ ⟩ (do
            void "quicksort" ⟨ "A" :: "low" :: "p" + -1 :: [] ⟩
            void "quicksort" ⟨ "A" :: "p" + 1 :: "high" :: [] ⟩
          )
      end ::
      procedure "partition"
```

```
    〈 "A" of * (int ⟦⟧) :: "low" of int :: "high" of int :: [] 〉 int begin
    let〈 "pivot" ◦ int := * ("A" ⟦ "high" ⟧) 〉 (do
      for〈 "j" := "low" ⨾ "j" + 1 <= "high" ⨾ & "j" := "j" + 1 〉
        if〈 * ("A" ⟦ "j" ⟧) + 1 <= "pivot" 〉 (do
          void "swapint" 〈 "A" ⟦ "low" ⟧ :: "A" ⟦ "j" ⟧ :: [] 〉
          & "low" := "low" + 1
        )
      void "swapint" 〈 "A" ⟦ "low" ⟧ :: "A" ⟦ "high" ⟧ :: [] 〉
    )
    return "low"
  end ::
  []
end ::

mod "buffer" imports "code" :: [] begin
  var "buf" ◦ int ⟦⟧ := new int ⟦ 25 ⟧ ::
  var "buf" ◦ bool := bool true ::
  procedure "go" 〈 [] 〉 int ⟦⟧ begin (do
    for〈 "i" := 0 ⨾ "i" <= 24 ⨾ & "i" := "i" + 1 〉
      & "buf" ⟦ "i" ⟧ := 24 + (- "i")
    void "quicksort" 〈 & "buf" :: 0 :: length "buf" + -1 :: [] 〉
  )
  return "buf"
  end ::
  []
end ::
```

```
  []

  main-imports "buffer" :: []

  main⁘ int ⟦⟧ := "go" ⟨ [] ⟩
end
```

Chapter 9 Conclusion

9.1 Results

This thesis demonstrates that the scope graph calculus framework constructed in Bach Poulsen et al. [3] admits implementations of correct-by-construction typecheckers in a relatively direct style, at least for languages whose programs' scope graphs can be constructed in a single pass over the program. The module language presented in Chapter 6 is an example of a kind of higher-order construct in the style of Leroy [9], taking a base language definition as input and generating a new language with a module system.

The code presented in Chapter 4 is only used a couple of times in the language implementations in Part II, but it constitutes a library in its own right that is suitable for reasoning about finite graphs in general. A version of the library is available independently of the rest of the code in this thesis at `https://www.github.com/kcsmnt0/graph`.

The generalization of the framework from Bach Poulsen et al. [3] presented in Chapter 5 is sufficient for defining other languages similar to the ones presented here, as well as languages with a wide range of scoping semantics that can be given in the scope graph calculus. It may not immediately extend to languages with more complicated typing semantics, as discussed in Section 9.3 below.

9.2 Readability

The set of potential target audiences for correct-by-construction code is somewhat different than in more traditional settings, at least at present, and accordingly the analysis of readability should also be somewhat different. A distinct characteristic of

fully correct-by-construction code is that the reader needs only to understand the type signatures in a program in order to understand the meaning of the program and verify its correctness; a reader of this thesis with the intent to understand the ideas presented and not necessarily the implementation details is thus mostly unconcerned with the term-level code, and will not be hindered by proofs that are informally clear but formally unpleasant. On the other hand, while a nontrivial formal proof is very rarely readable enough to stand entirely on its own without informal presentation, more readable proof code requires less explanation in general, sometimes significantly so.

In the specific case of this thesis, the specifications of the typecheckers in Part II are given by the definitions of the intrinsically-typed syntax types and the Erasure types; these are expected to be reasonably readable and informative to a reader with a general background in type theory, as with the definitions in Bach Poulsen et al. [3]. The term-level implementations of these functions are not expected to be immediately comprehensible to a reader without a strong familiarity with Agda and its standard library, but with the aid of the informal explanation given in the thesis text, the reader is expected to be able to understand and recognize the pattern of typechecker definition using surjection proofs as a variant of the more traditional pattern of mapping functions over lists.

For an example of where readability in proof code could potentially be improved, consider the following excerpt from the definition of the STLC scopechecking function ∃Erasure^f in Subsection 7.2.3.

```
∃Erasure^f s (bool b) = finite [ -, bool ] λ where (_ , bool) → here refl
∃Erasure^f s (nat n) = finite [ -, nat ] λ where (_ , nat) → here refl
∃Erasure^f s (suc e) =
  ∃Erasure^f s e <&^f>
```

```
  surjection

    _
    (λ where (_ , suc er) → -, er)
    (λ where (_ , suc er) → refl)
```

A more traditional Haskell-style definition capturing the computational behavior of this excerpt might read as follows.

```
∃Erasure^f s (bool b) = return (bool b)
∃Erasure^f s (nat n) = return (nat n)
∃Erasure^f s (suc e) = map suc (∃Erasure^f s e)  -- or "∃Erasure^f s e <&> suc"
```

This highlights the general pattern that the correct-by-construction code shares in common with more traditional code; each of the elements of the traditional definition is recognizable in the correct-by-construction definition, with the caveat that the second argument to surjection is actually the *inverse* of the function that is passed to map as its first argument and the actual first argument to map is inferred. The comparison also illustrates some syntactic overhead in the correct-by-construction code, as each line includes proof information that is most likely irrelevant to a human reader, namely the lambda subexpressions involving the refl constructor.

9.3 Future work

Most or all of the manual proof overhead covered in Section 9.2 can be eliminated by proof automation, at least in principle: all of the surjection proofs presented throughout

this thesis are trivial enough that the interactive Agda Emacs mode is able to fully guide their development with the help of some button presses but no actual ingenuity from the user, so it is expected that this process could be fully automated, requiring the user to specify only the codomain of each surjection and deriving the surjective function itself.

Extending the approach covered by this thesis to more complex languages that may require multiple passes and queries over incompletely-constructed scope graphs, as in van Antwerpen et al. [13], is left for potential future work. This includes languages that require the interleaving of scope graph construction and type inference in order to construct a scope graph for a program, including languages with traditional field accessor syntax ("dot notation") for record-like types.

The auxiliary -root scopes and linear-time scope search explained in Subsection 7.2.2 should be avoidable. It might be sufficient to introduce an annotation stage between raw terms and intrinsically-scoped terms, operating with knowledge of the relation between the structure of a raw term and the structure of the scope graph generated for it by the object language's static semanantics.

Another interesting avenue of further development would be to build parsers for these languages with a library like Agdarsec [1], to achieve an interpreter implemented end-to-end in Agda with important properties proven by construction. The evaluators defined in the appendix exist in this development primarily to justify the existence of the languages they evaluate, and are fairly inefficient; it may also be possible to construct interpreters over intrinsically-typed languages in this scope graph framework that verify more semantic properties by construction than just type preservation, such as guarantees about computational complexity, or to build a compiler backend verified to faithfully implement the semantics of an interpreter defined in this style.

9.4 Retrospective

Apart from initially learning Agda and becoming familiar with the standard library, the most challenging part of this project in terms of development time and effort was probably the graph search code, which went through several iterations before reaching a workably concise implementation (which is still fairly involved compared to a standard non-dependently-typed implementation). This speaks to a frequent general challenge in the space of dependently-typed programming, which is the lack of a cohesive established ecosystem of libraries, partly due to the relative youth and obscurity of languages like Agda and partly due to it still being unclear how to best organize a communal system of libraries of formalized mathematics at the scale of an ecosystem like Hackage for Haskell or NPM for JavaScript.

The scopechecking and typechecking code came together in light of the organizing principle of returning Listable values, which was motivated by the need to track ambiguity in terms in order to encode a complete correctness specification for typechecking. This also led to the need for the VarPath and ProcPath types in the example languages, to account for shadowing rules that uniquely resolve otherwise ambiguous references in some cases.

Ultimately, while this method of typechecker construction is certainly nontrivial to learn and requires a prerequisite familiarity with the constructs presented in Part I, it yields usable and explicable results for the set of languages it works over. The pattern, though not immediately applicable to practical interpreter construction, is definitely promising as an avenue of further development of verified language implementations.

References

[1] Guillaume Allais. Agdarsec total parser combinators. https://gallais.github.io/pdf/agdarsec18.pdf, 2017.

[2] Casper Bach Poulsen, Pierre Néron, Andrew P. Tolmach, and Eelco Visser. Scopes describe frames: A uniform model for memory layout in dynamic semantics. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 20:1--20:26, 2016. URL https://doi.org/10.4230/LIPIcs.ECOOP.2016.20.

[3] Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *PACMPL*, 2(POPL):16:1--16:34, 2018. URL https://doi.org/10.1145/3158104.

[4] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-563.pdf.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552--593, 2013. URL https://doi.org/10.1017/S095679681300018X.

[6] Nils Anders Danielsson, Ulf Norell, SC Mu, S Bronson, D Doel, P Jansson, and LT Chen. The Agda standard library. 2011. URL http://www.cs.nott.ac.uk/nad/repos/lib.

[7] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143--155, 2011. URL https://doi.org/10.1145/2034773.2034796.

[8] Denis Firsov and Tarmo Uustalu. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pages 33--44, 2015. URL https://doi.org/10.1145/2808098.2808102.

[9] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269--303, 2000. URL http://journals.cambridge.org/action/displayAbstract?aid=54525.

[10] Pierre Neron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 205--231, 2015. URL https://doi.org/10.1007/978-3-662-46669-8_9.

[11] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[12] The Coq Development Team. The coq proof assistant, version 8.9.0, January 2019. URL https://doi.org/10.5281/zenodo.2554024.

[13] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *PACMPL*, 2(OOPSLA):114:1--114:30, 2018. URL https://doi.org/10.1145/3276484.

[14] Wikipedia contributors. Quicksort --- Wikipedia, the free encyclopedia, 2019. URL https://en.wikipedia.org/w/index.php?title=Quicksort&oldid=900258906. [Online; accessed 11-June-2019].

Appendix A: Term and program evaluation

While the focus of this thesis is on scopechecking and typechecking the example languages in order to produce intrinsically-typed terms, the intrinsically-typed syntax types themselves are arguably not very interesting in isolation - it is not necessarily obvious whether they are suitable for any particular kind of analysis or interpretation as syntactic objects in their own right beyond serving as a correctness specification for typechecking. In this appendix, an intrinsically type-safe evaluation function is given for programs in the Proc language presented in Chapter 8 of this thesis. This evaluation function takes intrinsically-typed terms as input and produces intrinsically-typed values as output, as in [3], demonstrating that the presented intrinsically-typed syntax types are suitable at least as input to a certain kind of intrinsically type-safe interpreter. The code accompanying this development also includes an evaluator for STLC, which is the same as the STLC evaluator code in [3] with minor modifications to support values of the Cell type described below.

The evaluation function is defined in a similar style to the ones in [3]: the type of a value in a heap is indexed over the types of the heap and the value, and a heap is represented as a collection of frames that each correspond to a particular scope in the scope graph of the program, each containing a value for each of the associated scope's bound declarations and a reference to an associated frame for each of its parent scopes. The module system in Code.Module includes code to generate a program's initial heap, and the Code.STLC and Code.Proc languages define evaluators over terms that are run with these intial heaps.

The purpose of the code in this appendix is mostly to serve as a sanity check that the language defined in Chapter 8 is realistic in the sense that it can be given unsurprising

dynamic semantics by an already-established pattern. The explanatory content in this appendix is somewhat shallower than much of the content in the rest of this thesis; the goal is not to instill the reader with a functional understanding of the interpreter construction pattern described in [3], but rather to justify briefly that the contributions of this thesis are relevant in that setting.

Module system

The _⊨_ parameter to the TypedModuleSystem module in Code.Module is the type of intrinsically-typed values in a heap with frames representing scopes in the graph of program *p*; informally, an Agda value of type *ht* ⊨ *t* represents an object language value of type *t* that may reference any frame in *ht* (e.g. a closure value holds a reference to the frame it closes over).

```
module TypedModuleSystem
  (p : Program)
  (_⊨_ : ScopeGraph.HeapType (Program.graph p) → Type → Set)
  {{⊨-weakenable : ∀ {t} → Weakenable (_⊨ t)}}
  {_⊢_∶_ : Program.Scope p → Term → Type → Set}
  (_⊢_∶ᶠ_ : ∀ s e t → Listable (s ⊢ e ∶ t))
  where
  open Program p
  open ScopeGraph graph
```

A program's initial heap contains a frame for the main expression, a frame for each module, and a frame for each declaration in each module.

```
modsHeapType : HeapType
```

```
modsHeapType = List.map (λ where (_ , i) → mod i) (indexed modules)

termRootsHeapType : HeapType
termRootsHeapType =
  List.concatMap
    (λ where
      ((_ , m) , i) →
        List.map
          (λ where (_ , j) → term-root i j)
          (indexed (declarations m)))
    (indexed modules)

programHeapType : HeapType
programHeapType = main-root :: modsHeapType ++ termRootsHeapType
```

The potential for cyclic imports introduces a challenge in the construction of an initial heap with this heap type: the heap must contain a "value" for each declaration in each module, but some declarations in some modules may be mutually recursive, so there is not necessarily any linear order in which the declarations can be evaluated in order to construct all of the values for each module frame. The solution implemented here is to soften the notion of "value" to include a constructor for an unevaluated term paired with a reference to the frame it will be evaluated in, similar to the notion of a "thunk" in a lazy language, and include the original intrinsically-typed terms declared in a program directly in the initial heap of the program.

The Cell type represents the type of "values" that can be in a heap frame during evaluation.

```
data Cell (ht : List Scope) (t : Type) : Set where
```

```
    val : ht ⊨ t → Cell ht t
    term : ∀ {s e} → s ⊢ e ꞉ t → s ∈ ht → Cell ht t
```

The Heap module from Code.Scope is parameterized over the type of "values" in the heap and includes the types and definitions from [3] section 4.3, including the Heap type.

```
open Heap (λ where ht (x , t) → Cell ht t)
```

The fine details of the construction of the program heap are somewhat involved, requiring some careful manipulation of list indices representing references to frames in a heap, but at a high level it just recurses over the modules and definitions in the program and puts each declaration in the appropriate frame with the term constructor. The definition of programHeap is within the definition of the ProgramTyping record in Code.Module, so it has access to the mainTyping and moduleTyping fields. (The ++⁺ (prefix rather than infix) function is the append function for the All type, from Data.List.Membership.Propositional.Properties, and All.tabulate from Data.List.Relation.Unary.All maps a function taking a list index as input over every index in the list in order to produce an All value.)

```
programHeap : Heap programHeapType
programHeap =
  (term mainTyping (here refl) :: [] , importLinks mainImports) ::
    ++⁺ (All.tabulate moduleFrame) (All.tabulate termFrame)
  where
    moduleFrame : ∀
      {s} → s ∈ modsHeapType → HeapFrame s programHeapType
```

```
termFrame : ∀
  {s} → s ∈ termRootsHeapType → HeapFrame s programHeapType
```

Proc

In Code.Proc.Typed, value types are declared for expressions, procedures, and declarations. The ExprVal type is straightforward, with the ptr constructor taking both a pointer to some value and a reference to the frame that the pointer was valid in; the proc constructor for ProcVal is similar to ptr and to a closure constructor for an intrinsically-typed STLC value type, taking an explicit equality proof to ease pattern matching.

```
data ExprVal (ht : HeapType) : {l : Size} → Type → Set where
  null : ∀ {l t} → ExprVal ht {l} (* t)
  ref : ∀ {l x} (p : VarPath x s) → s ∈ ht → ExprVal ht {l} (* varType p)
  _[[_]] : ∀ {l t} → ExprVal ht {l} (* (t [[]])) → ℤ → ExprVal ht {l} (* t)
  unit : ∀ {l} → ExprVal ht {l} unit
  bool : ∀ {l} → Bool → ExprVal ht {l} bool
  int : ∀ {l} → ℤ → ExprVal ht {l} int
  arr : ∀ {l t} → List (ExprVal ht {l} t) → ExprVal ht {↑ l} (t [[]])

data ProcVal (ht : HeapType) (ts : List Type) (t : Type) : Set where
  proc : ∀
    {i xts} →
    ts ≡ List.map proj₂ xts → Proc i xts t → i ∈ ht →
    ProcVal ht ts t
```

The DeclVal type is the actual type of values used in the cells in the heap.

```
data DeclVal (ht : HeapType) : DeclType → Set where
  var : ∀ {t} → ExprVal ht t → DeclVal ht (var t)
  proc : ∀ {ts t} → ProcVal ht ts t → DeclVal ht (proc ts t)
```

The Code.Proc.Eval module defines the evaluator for the Proc language. Only a couple interesting parts of the code are included here; the rest generally follow the pattern from [3], with some slight modifications to account for the Cell type in heap frames.

```
module Code.Proc.Eval p where
open Raw.Program p
open ScopeGraph graph
open VarPath graph
```

The evaluator involves several mutually-recursive functions, with termination established by a natural number "fuel" argument as in [3]. The M monad, also from that paper, represents a monadic evaluation context indexed over an extension to the given heap type - e.g. evalExpr represents a monadic computation that returns an ExprVal in some heap type *ht′* that is a consistent extension of the heap type *ht*, accounting for the possibility that expression evaluation may allocate new frames on the heap.

```
evalExpr : ∀ {s t ht} → ℕ → Typed.Expr s t → M s (λ ht′ → ExprVal ht′ t) ht
evalStmt : ∀ {s ht} → ℕ → Typed.Stmt s → M s (const ⊤) ht
evalDecl : ∀ {s t ht} → ℕ → Typed.Decl s t → M s (λ ht′ → DeclVal ht′ t) ht
```

Notably, evalExpr and evalDecl both have Val types in their return types, not Cell, indicating that they must deal with the form of laziness introduced by the term constructor to Cell in some way. The force function evaluates a Cell to a DeclVal, invoking evalDecl

if given an unevaluated term. The usingFrameM function is from [3], evaluating a computation in the M monad in a frame specified by an index into the type of the current heap.

```
force : ∀ {s t ht} → ℕ → Cell ht t → M s (λ ht′ → DeclVal ht′ t) ht
force k (val x) = return x
force k (term (typing e _) i) = usingFrameM i (evalDecl k e)
```

The access function retrieves the value pointed to by a Ptr, or fails with failM (defined as an alias of timeoutM) if the pointer is invalid. An associated function update, implemented similarly, sets the value pointed to by a Ptr. The ℤ→fin? function, defined locally, has type ∀ {n} → ℤ → Maybe (Fin n) and checks to see whether an integer is a positive number within a given (inferred) bound.

```
access : ∀ {s t ht} → ℕ → ExprVal ht (* t) → M s (λ ht′ → ExprVal ht′ t) ht
access zero _ = timeoutM
access (suc k) null = failM
access (suc k) (ref p i) = do
  var v ← usingFrameM i (getValM (varRef p)) »= force k
  return v
access (suc k) (v ⟦ n ⟧) = do
  arr vs ← access k v
  case ℤ→fin? n of λ where
    nothing → failM
    (just n′) → return (List.lookup vs n′)

update : ∀ {s t ht} → ℕ → ExprVal ht (* t) → ExprVal ht t → M s (const ⊤) ht
```

The variable dereference case of evalExpr simply saves the resolution path for the variable being referenced along with a pointer to the frame that it is valid in.

```
evalExpr (suc k) (& p) = ref p <$> getFrameM
```

The case for the * operator evaluates the given expression to a pointer and then calls the access function to retrieve the value at that pointer.

```
evalExpr (suc k) (* e) = evalExpr k e >>= access k
```

The code for the application constructor is a little unpleasant, and there may be a more idiomatic way to write it; the issue is mainly that operating over values of type All as collections can require careful reasoning about the type-level indices of All, and it's often challenging to set up the relevant predicates so that the necessary reasoning is definitional. In this case, evalExpr is mapped over the collection of argument expressions to produce a collection of values of type ExprVal, which is then converted into a collection of values of type DeclVal; the sequenceAll function from Code.Util is a version of the standard Haskell monadic sequence function for the All type, and the line below the one containing it is just applying the *eq* proof from the proc constructor to show that the list of values obtained by evaluating the argument expressions does in fact have the same type as the parameter list. (The functions map$^+$ and map$^-$ functions from Data.List.Relation.Unary.All.Properties convert back and forth between the isomorphic types All *P* (List.map *f as*) and All (*P* ∘ *f*) *as*; subst is from Relation.Binary.PropositionalEquality with the type ∀ {*A*} (*P* : *A* → Set) {*x y*} → *x* ≡ *y* → *P x* → *P y*.)

```
evalExpr (suc k) (p ⟨ es ⟩) = do
  proc (proc eq (proc {{eq'}} s e) i) ← getValM (procRef p) >>= force k
```

```
  es′ , i ← sequenceAll (All.map (λ e′ {_} → evalExpr k e′) es) ^ i
  let es″ = map⁺ (map⁻ (subst (All _) eq (All.map (val ∘ var) es′)))
  j ← initFrameM es″ (i :: [])
  _ , j ← usingFrameM j (evalStmt k s) ^ j
  usingFrameM j (evalExpr k e)
```

Executing a Proc program is then as simple as calling evalDecl on mainTyping, evaluating it in the main-root scope in the generated programHeap for the program. For convenience, evalProgram is a version of runProgram that doesn't include the final Heap or heap extension witness.

```
module _ pt where
  open ProgramTyping pt

  runProgram :
    ℕ →
    Maybe (∃ λ ht′ → Heap ht′ × DeclVal ht′ mainType × programHeapType ⊆ ht′)
  runProgram k = evalDecl k (typedDecl mainTyping) (here refl) programHeap

  evalProgram : ℕ → Maybe (∃ λ ht′ → DeclVal ht′ mainType)
  evalProgram = Maybe.map (Σ.map₂ (proj₁ ∘ proj₂)) ∘ runProgram
```

Appendix B: Code artifact

The code artifact distributed along with this paper is a ZIP archive file named `thesis-code.zip` (184KB) containing the code described by this thesis. The archive is also available online at `http://web.cecs.pdx.edu/~cas28/thesis`. The code is compatible with Agda version 2.6.0.1 and standard library version 1.0.1.