Université Paris Diderot (Paris 7)
École doctorale : Sciences Mathématiques de Paris Centre

DOCTORAT
Informatique

Une approche unifiante pour programmer sûrement avec de la syntaxe du premier ordre contenant des lieurs

english title:
Namely, Painless

A unifying approach to safe programming with first-order syntax with binders

Nicolas Pouillard

Thèse dirigée par François Pottier
et soutenue le 13 Janvier 2012 devant le Jury composé de :

Président	M. Roberto Di Cosmo
Rapporteurs	M. Andrew Pitts
	M. Dale Miller
Examinateurs	M. Daniel Hirschkoff
	M. Conor McBride
Directeur	M. François Pottier

À mon papa...

Abstract

This dissertation describes a novel approach to safe meta-programming. A meta-program is a program which processes programs or similar data. Compilers and theorem provers are prime examples of meta-programs which could benefit from this approach. To this end, this work focuses on the representation of names and binders in data structures.

Programming errors are really easy to make with usual techniques. We propose an abstract interface to names and binders that rules out these errors. This interface is implemented as a library in Agda. It allows defining and manipulating term representations in nominal style. Thanks to abstraction, other styles are supported as well: the de Bruijn style, the combinations of these styles, and more.

Whereas indexing the types of names and terms with a natural number is a well-known technique to better control de Bruijn indices, we index them with worlds. Worlds are at the same time more precise and more abstract than natural numbers. Via logical relations and parametricity, we are able to demonstrate in what sense our library is safe, and to obtain theorems for free about world-polymorphic functions. For instance, we prove that a world-polymorphic term transformation function must commute with any renaming of the free variables. The proof is entirely carried out in Agda.

The usability of our technique is shown on several examples including normalization by evaluation which is known to be challenging. We show that our world-indexed approach can express a wide range of data types by embedding several definition languages from the literature.

Résumé

Cette thèse décrit une nouvelle approche pour la méta-programmation sûre. Un méta-programme est un programme qui manipule des programmes ou assimilés. Les compilateurs et systèmes de preuves sont de bons exemples de méta-programmes qui bénéficieraient de cette approche. Dans ce but, ce travail se concentre sur la représentation des noms et des lieurs dans les structures de données.
Les erreurs de programmation étant courantes avec les techniques usuelles, nous proposons une interface abstraite pour les noms et les lieurs qui élimine ces erreurs. Cette interface est implémentée sous forme d'une bibliothèque en Agda. Elle permet de définir et manipuler des représentations de termes dans le style nominal. Grâce à l'abstraction, d'autres styles sont aussi disponibles : le style de De Bruijn, les combinaisons de ces styles, et d'autres encore.
Nous indiçons les noms et les termes par des mondes. Les mondes sont en même temps précis et abstraits. Via les relations logiques et la paramétricité, nous pouvons démontrer dans quel sens notre bibliothèque est sûre, et obtenir des "théorèmes gratuits" à propos des fonctions monde-polymorphiques. Ainsi une fonction monde-polymorphique de transformation de termes doit commuter avec n'importe quel renommage des variables libres. La preuve est entièrement conduite en Agda.
Notre technique se montre utile sur plusieurs exemples, dont la normalisation par évaluation qui est connue pour être un défi. Nous montrons que notre approche indicée par des mondes permet d'exprimer un large panel de type de données grâce a des langages de définition embarqués.

Remerciements

Trois années de noms et de termes. Des liens puis des noms, des noms et des indices, puis des indices seulement, puis finalement encore des noms. Aujourd'hui j'écris ces remerciments et ce sont vos noms qui me viennent à l'esprit.

Je porte mes remerciements tout d'abord à ma famille, mes parents, pour m'avoir soutenu dans mes longues études. Merci aussi à mon épouse, Gaëlle, qui m'a non seulement soutenu mais épaulé, écouté, poussé, aidé, ... Bref, elle est aussi responsable de cet aboutissement.

Je tiens particulièrement à remercier mes relecteurs pour leur travail remarquable. Leurs relectures de ce document m'ont permis de nombreuses améliorations rendant le présent document plus accessible et plus complet.

Je remercie le LRDE qui m'a aiguillé vers la recherche, je remercie en particulier Akim Demaille pour son cours de compilation qui m'a initié à la conception de langages.

Je remercie d'avance tous ceux que je ne mentionne pas individuellement ici : membres de ma famille, camarades, collègues, ami(e)s, simples connaissances. Vous qui justement lisez ces lignes, je vous en remercie.

Un grand merci à toute l'équipe Gallium de l'INRIA Rocquencourt et tous les membres que j'y ai rencontrés. Cette équipe m'a accueilli jeune ingénieur et m'a rendu jeune chercheur. Plus particulièrement je remercie : Michel Mauny pour m'avoir accepté en stage mais aussi sur le contrat qui a suivi ; j'y ai appris une foule d'anecdotes sur l'évolution des dialectes successifs de Caml ; grâce à lui et malgré lui j'ai appris à apprécier la programmation paresseuse (sans jeux de mots) ce qui m'a ouvert de nouveaux horizons. Puis c'est au tour de François Pottier, qui m'a accepté en thèse et m'a habilement guidé et soutenu pendant ces années. Il a su être disponible et à l'écoute. Outre une montagne de connaissances scientifiques, j'ai appris de lui l'art d'écrire des articles bien que j'ai encore beaucoup à apprendre à ce sujet. Xavier Leroy pour son accueil et sa bienveillance quant à la bonne marche de l'équipe ; Damien Doligez pour son intarissable culture et ses théories sur tout ; Didier Rémy pour ses discussions sur les détails du fonctionnement de TEX :) ; Alain Frisch pour m'avoir enseigné tellement de choses sur OCaml et la programmation fonctionnelle en général pendant les mois durant lesquels j'ai partagé son bureau ; Yann Régis-Gianas pour avoir tracé

une voie qu'il m'a suffit d'emprunter ; Berke Durak pour m'avoir accompagné dans le développement d'ocamlbuild ; Benoit Razet pour toutes nos discussions sur les automates et les structures de flots de données ; Gérard Huet pour ses anecdotes variées qui m'ont beaucoup appris ; Jean-Baptiste Tristan pour avoir participé et accueilli chez lui l'équipe des "jeunes" de Gallium au concours ICFP ; Tous ceux qui ont participé aux concours ICFP ; Arthur Charguéraud pour son secret de l'efficacité : les axiomes Coq et `notepad++` :) ; Zaynah Dargaye, Boris Yakobowski et Paolo Herms pour les discussions et les bons moments passés avec eux ; Benoît Montagu pour m'avoir ouvert son bureau et avoir partagé des discussions fructueuses sur nos thèses respectives ; Julien Cretin pour sa singularité mais aussi pour tous les sujets sur lesquels nous pouvons discuter agréablement ; Alexandre Pilkiewicz de m'avoir traîné jusqu'au parcours sportif un certain nombres de fois ; Jonathan Protzenko car grâce à lui le web évolue de jour en jour :) ; Tahina Ramananandro pour nos discussions politiques et nos concours de jeux de mots fumeux ; Dana Xu pour ses encouragements ; Gabriel Scherer et Valentin Robert pour leurs nombreux commentaires sur des parties de ce document.

À Luc Maranget pour les discussions nautiques et ses slogans devenus cultes : "Les programmeurs veulent des types plus riches !", "Les types existentiels existent-ils ?", "Ils nous parlent seulement des types, où sont les termes ?" ; Jean-Jacques Levy pour son talent à poser tout haut les questions auxquelles tout le monde pense.

J'ai une pensée aussi pour tous les "camarades de navette" avec qui j'ai pu discuter pendant un des nombreux trajets entre Paris et Rocquencourt.

À Jean-Philippe Bernardy, pour avoir rendu la paramétricité accessible et claire dans un cadre simple et élégant. Ses travaux ont amplement aidé les miens en apportant l'outil de preuve adapté à ce problème.

Contents

Chapter 1

Introduction

Foreword

> This document describes my work during these past three years which has been done under the direction of François Pottier. This work extends the research topic of safe meta-programming by providing a novel approach to the representation of programs with names and binders. – Nicolas Pouillard

Safe meta-programming is a research topic which is part of the broader topic of programming language design. One of the goals of the research in programming language design is to provide programmers with better tools allowing to produce correct software applications in a reasonable amount of time. Since correct software applications (programs for short) are not supposed to crash, language designs have to incorporate safety features to detect programming errors as soon as possible. Modern language designs often come with some static disciplines such as lexical scoping, strong typing and the like. These static disciplines are used to reject the programs we do not want to run.

A program may process various forms of data such as numbers, text, spreadsheets, images, sound, and many more. Some programs process other programs, they are called meta-programs. Meta-programs cover a wide class of programs ranging from compilers, to static analysers, code generators, etc. Proof systems, proof assistants and theorem provers are also meta-programs since propositions and proofs share the same kind of structure as programs. Moreover since language designs come with static disciplines, we claim that meta-programs should follow these disciplines when processing programs. For instance if a program `A` written in a programming language `P` generates a program `B` written in a programming language `Q`, we want `A` to follow the static discipline of `P`, moreover we want `B` which does not exist yet to follow the static discipline of `Q`.

This first chapter aims at introducing all the notions to understand the research problem that we are focusing on. This chapter requires little knowledge in the design of programming languages, but does require some knowledge about computers, mathematics and programming. Hence this introduction chapter can be safely skipped by readers accustomed with the research topic.

1.1 Programs and Programming Languages

To introduce the notions of programs and programming languages, let us start with an example of a simple program:

```
print "Hello! 2 times 21 is equal to " >>
print (show (2 * 21))
```

Most generally, programs are written to be executed on a computer. Running a program often results in the execution of different tasks and yields a final result. In our previous example the program is meant to print on the screen: "`Hello! 2 times 21 is equal to 42`". Printing on the screen is done with the construct called `print`. This printing construct is used twice. The two uses are *sequenced* using the construct `>>`, which will perform the action on the left and then the action on the right. The number `42` is computed from the arithmetic expression `2 * 21` which is then turned into a printable form using the construct called `show`.

This program, such as any program, is written in a programming language. Programming languages are described by a finite set of constructs, rules explaining how the constructs can be used and rules giving the behavior of programs. In our further examples the programming language contains constructs such as `print`, `>>`, `show`, `*`, character strings such as `"Hello! 2 times 21 is equal to "`, numbers such as `2` and `21`, and parentheses. With these constructs our programming language is not much better than a simple calculator.

This tiny introductory programming language is built to be a subset of the AGDA language. AGDA [Norell, 2007] is a total functional programming language, based on dependent type theory. AGDA is used thoroughly in this document. First it is used to introduce the research topic of meta-programming. Second it is used to implement our solution to the problem. Third it is used to formalize our system and show its soundness. Alas, advanced languages such as AGDA are far from simple to completely describe. The first set of programming constructs we introduce is roughly common to most languages and is described simply in great detail. The second set of constructs follows the development of functional programming languages (such as HASKELL, ML, or LISP). They are described more quickly and may require further reading about these topics. An excellent introduction to functional programming in Haskell is Graham Hutton's "Programming in Haskell" [Hutton, 2007]. Moreover the third set of constructs is specific to dependently typed and total functional languages or even restricted to Agda only. To fully grasp these constructs we recommend further reading on these topics. In particular Ulf Norell's [2007] PhD thesis on the theoretical and practical development of Agda is recommended.

1.2 Program syntax

Examples The following program makes use of the four basic arithmetic operators to compute the value 42. Operators are attributed with their usual semantics, addition is `+` and multiplication is `*`. Subtraction and division (∸ and ÷) are restricted to natural numbers, meaning that they return `0` when going out of range.

```
(1 + 1) * (28 ∸ 14 ÷ 2)
```

The following program illustrates character string literals and their concatenation with `++`. This program prints "`Hello world!`" on the screen:

```
print ("Hello" ++ " " ++ "world!")
```

Our next program illustrates the use of `show` to turn a number into a character string containing its decimal representation. This program prints "`show 1 renders as: 1`" on the screen:

```
print ("show 1 renders as: " ++ show 1)
```

The following program illustrates the use of >> to sequence two actions from left to right. This program prints "`Print me first.`" and then prints "`Then print me.`" on the screen.

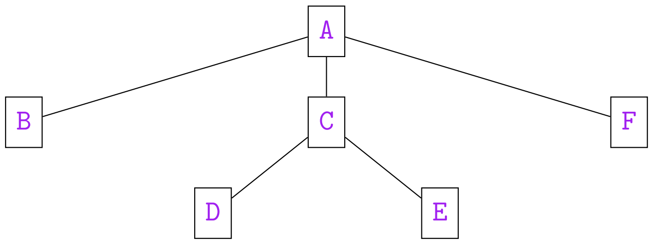
```
print "Print me first." >> print "Then print me."
```

We have lists in our programming language. The simplest list value is the empty list, written `[]`. The following program is a list with three elements, namely `"a"`, `"bc"` and `"5"`:

```
[ "a" , "b" ++ "c" , show (2 + 3) ]
```

We go from lists to trees by using the construct called `node`. The first argument of `node` is the label of the node and the second is the list of children of the node. The following program represents a six-node tree depicted hereafter:

```
node "A"
  [ node "B" []
  , node "C" [ node "D" [] , node "E" [] ]
  , node "F" [] ]
```



Syntax of our arithmetic calculator The syntax of a programming language describes when a program has a valid "shape". In particular it does not check if the program has a valid "meaning". For instance, the program `1 + "Hello!"` has a valid syntax but no valid meaning in our tiny language.

Any non-empty sequence of digits (`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`) is a valid literal number and a valid program as well. A valid literal character string is a sequence of characters different from the double quote character (`'"'`) and surrounded by two double quote characters (we avoid talking about escape sequences because we do not need them here). If two programs p_1 and p_2 are syntactically valid, then for any operator $\bullet$ among $+$, $\dot{-}$, $*$, $\div$, `++`, $>>$ the program $p_1 \bullet p_2$ is syntactically valid. These operators are said to be *infix*, meaning that they appear between the two operands. If p is a syntactically valid program, then `print` p, `show` p and (p) are syntactically valid. If p_1, p_2, ..., p_n are syntactically valid programs, then $[\ p_1\ ,\ p_2\ ,\ \ldots\ ,\ p_n\ ]$ is syntactically valid. Finally if p_1 and p_2 are syntactically valid programs then `node` p_1 p_2 is syntactically valid.

Of course not every textual program is syntactically valid. Consider this example: `3 + * 4`. This program does not follow the rules we just described, hence it is rejected as syntactically invalid.

Syntax also comprises some details about layout and comments. Spaces and newlines are important to some extent. Spaces or parentheses have to be used to separate the words in a program. However, as long as they are separated, how many spaces or newlines are used to separate two words does not matter. There is special support in the syntax to embed comments in our programs. There are two syntaxes used in our language. A comment can start with `{-` and stop with `-}` or can start with `--` and stop at the end of line. Here is a syntactically valid program to illustrate layout and comments:

```
print "Hello..." >> -- this prints Hello...
print   {- extra spaces and comments are ok -} "... world!"
```

1.3 Typing

Typing captures, ahead of time, programs that may go wrong. Typing might be seen as a vigilant companion giving advice on our programs. When typing says that our program is ill-typed, then this is a program we may not want to even try running. If the typing says our program is fine, then we know for sure that the program will not crash.

Such a technology is to be applied systematically before running a program. It is an affordable safety measure which massively helps program development, maintenance, testing, and verification.

However, verifying that a program cannot crash without running it is a difficult problem to say the least. In particular we want the typing part of our system to always terminate, even if the given program does not terminate. We not only want the typing to terminate but to answer if the program is well typed or not. If so we say that such a type system is decidable.

A type system is said to be *sound* if and only if every well-typed program is a non-crashing program. We expect every well-designed type system to be sound.

A type system is said to be *complete* if and only if every non-crashing program is well-typed. The decidability constraint entails that a type system will never be complete for non-trivial languages. For further reading on the topic of type-systems we recommend reading “Types and Programming Languages” [Pierce, 2002] and “Advanced Topics in Types and Programming Languages” [Pierce, 2005].

Typing our arithmetic language The first goal of type systems is to prevent us from running programs that may crash. To do so, they define *types*, an abstraction to classify pieces of data. By extension, types also classify programs computing pieces of data. Types help to define a common “contract” for producers and consumers of a piece of data. This kind of “contracts” in on the shape of data. The more shapes are allowed by a type, the weaker the type is. The strength of a type affects producers and consumers in different ways. If a type is weak, it is easy for producers of this type to be well-typed. If a type is strong, the producers have to meet all the constraints imposed by the type. This is the opposite for consumers. If a type is strong, the data has a known precise shape, and the consumer has an easy job. If a type is weak, then the consumer has to consider all the shapes that the data can take.

To illustrate the notion of typing, we equip our little programming language with such a type discipline. Step by step we introduce new types and how the constructs of our language are typed.

We have only a few types of values. We introduce the type $\mathbb{N}$ as the type of numbers (`0`, `1`, `2`, ...) and of programs computing numbers (such as `2 * 21`).

Given a type τ (such as $\mathbb{N}$) and a program `p` (such as `6 * 7`), we write `p` *has type* τ to formally assert that the program `p` is of type τ. We call these type assertions *typing judgements*. The following paragraphs define the type-system for our tiny language by giving a typing judgement for each construct.

For each number `n`, `n` has type $\mathbb{N}$. Given two programs p_1 and p_2, if both have type $\mathbb{N}$, then $p_1 + p_2$, $p_1 * p_2$, $p_1 \dot{-} p_2$, and $p_1 \div p_2$ all have type $\mathbb{N}$ as well. For instance with these rules one can formally assert: `6 * 7` has type $\mathbb{N}$.

Our second type of basic values is text values. To this end we introduce the type `String` as the type of text values such as `"Hello!"` and of programs computing text values such as `"Hello " ++ show 42`. More precisely for each syntactically valid text value `s`, `s` has type `String`. If two programs p_1 and p_2 have type `String` then the program p_1 `++` p_2 has type `String` as well. If a program `p` has type $\mathbb{N}$ then `show p` has type `String`.

It is now time to show an ill-typed program: `"Hello!" ++ 42`. While this program is syntactically valid, it does not follow our typing rules. There is indeed only one rule about the construct `++`. This rule imposes programs on each side of `++` to be of type `String`. While `"Hello!"` is of type `String`, `42` is not. The only rule that applies to `42` says that it is of type $\mathbb{N}$. Since `String` and $\mathbb{N}$ are different, this program is rejected as being ill-typed.

Our little programming language has list values. However there is no single type for all lists. The type of a list depends on the type of its elements. We require all the elements of a list to have the same type. Given any type `A`, if p_1, p_2, ..., p_n are programs of type `A` then `[` p_1 `,` p_2 `, ... ,` p_n `]` is of type `List A`. Thus there is no single type for lists, but for each type `A` there is a type for lists where elements are of type `A`, namely `List A`. For instance `[ 1 , 2 ]` has type `List` $\mathbb{N}$, and `[ [ 1 , 2 ] , [ 3 ] ]` has type `List (List` $\mathbb{N}$`)`.

Our next type is the type of trees. Like for lists, the type of trees is parameterized by the type of its elements, namely the labels of nodes. For each type `A`, `Tree A` is the type of trees with labels of type `A`. To build a tree we use the construct `node`: given a type `A`, if p_1 has type `A` and p_2 has type `List (Tree A)` then `node` p_1 p_2 has type `Tree A`.

The following program has type `Tree String`:

```
node "A" [ node "B" [] , node "C" [] ]
```

This one has type `Tree ℕ`:

```
node 1 [ node 2 [] , node 3 [] ]
```

This one is ill-typed, though:

```
node 1 [ node "B" [] , node "C" [] ]
```

Finally the type `Interactive` is used for programs interacting with the environment. We currently introduced only two forms of interaction, namely `print` and `>>`. Given a program `p` of type `String`, `print p` has type `Interactive`. This interactive program computes an interactive value which is meant to be triggered. Once the interactive value is triggered it prints the value of the text `p` on the screen.

The second form of interaction enables to sequence two interactive values. Given two programs p_1 and p_2 of type `Interactive` the program p_1 `>>` p_2 has type `Interactive` as well.

1.4 Program representation, meta-programming

When reading, writing, and editing programs we often work with the textual form of programs. We write programs in text files and so their representation is a sequence of characters in a file.

Machines do not directly accept this kind of program for execution. Even languages very close to the hardware such as assembly languages are not directly understood by the machine. Hence programs are processed by other programs. These programs processing other programs are called meta-programs and are described in greater detail in the following section.

Up to here our simple programming language can process numbers, text, lists, trees and interactive values. What about programs? What is required to process programs themselves? If a program is simply a text in a file then our text value can represent programs. Here are two programs, the second is the textual representation of the first:

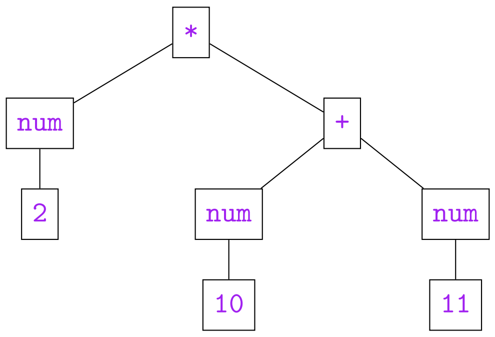
```
2 * (10 + 11) {- this should compute 42 -}
```

```
"2 * (10 + 11) {- this should compute 42 -}"
```

This is common knowledge nowadays that the textual representation is not adapted for any non-trivial processing. A first step called *lexing* gets rid of the lexical issues of layout and comments. The lexing step turns the program text to a list of words, called *tokens*. These tokens are often annotated by a *token class* to distinguish numbers, operators, constructs... Here is the same program as a simple list of tokens:

```
[ "2" , "*" , "(" , "10" , "+" , "11" , ")" ]
```

This representation is still unworkable for non-trivial processing. In particular there is too little structure in a list to reflect the program structure. To solve this issue, syntax trees have been introduced. Here is our same program, graphically depicted as a tree:



We picked names for some constructs such as “num” for numbers, “text” for text values, and “list” for lists. An exception is made for parentheses which no longer add any useful information and hence are not represented in the tree.

The process of translating text or tokens into a tree is called *parsing*. We will not discuss it more.

These syntax trees are generally called *Abstract Syntax Trees* or *AST* for short. Syntax trees are often said “abstract” because they no longer depend on some details of the “concrete” syntax.

Happily we have enough constructs in our language to build trees. We can hence show another representation of the same program as a tree built in our language:

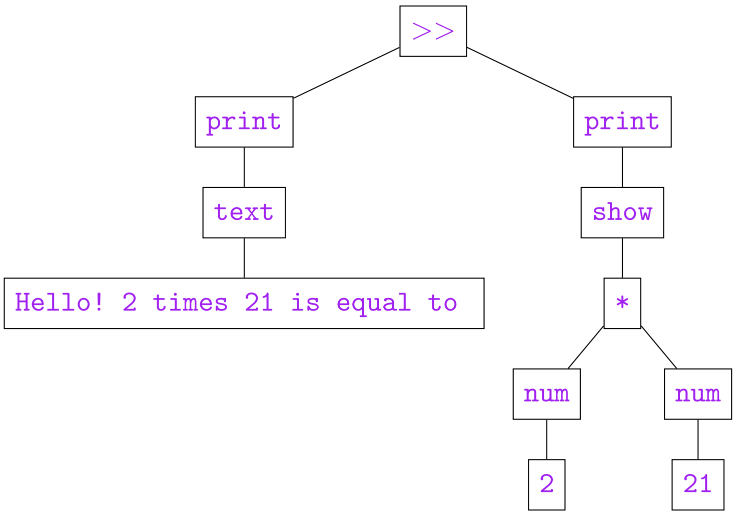
```
node "*"
  [ node "num" [ node "2" [] ]
  , node "+"
      [ node "num" [ node "10" [] ]
      , node "num" [ node "11" [] ] ] ]
```

Representing programs of our language as values of our language itself is good to have but not necessary. However we can expect a programming language good at processing programs to be good at representing as many programs as possible.

How to choose the tree representation for a programming language? Here we made a simple choice: for each syntax rule we have a corresponding tree node labeled by the construct and with as many children as needed.

```
print "Hello! 2 times 21 is equal to " >>
print (show (2 * 21))
```

The syntax tree, graphically:



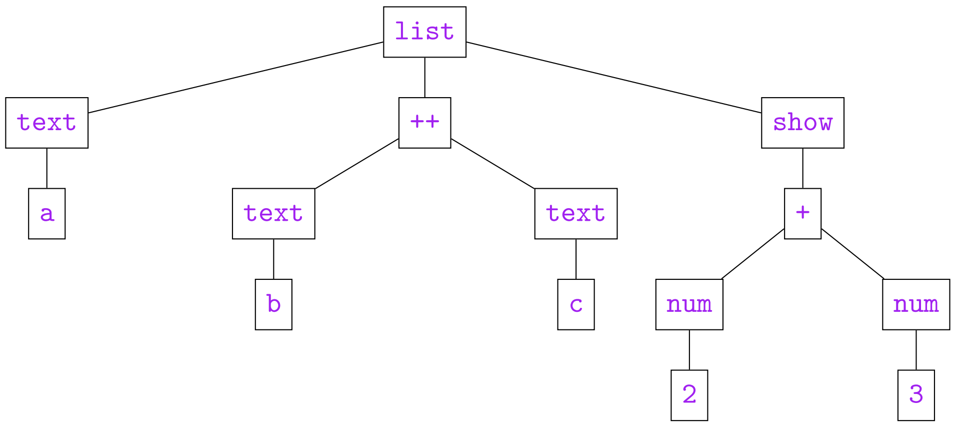
The syntax tree as a program:

```
node ">>"
  [ node "print"
      [ node "text"
          [ node "Hello! 2 times 21 is equal to " [] ] ]
  , node "print"
      [ node "show"
          [ node "*"
              [ node "num" [ node "2" [] ]
              , node "num" [ node "21" [] ] ] ] ] ]
```

An example to illustrate lists:

```
[ "a" , "b" ++ "c" , show (2 + 3) ]
```

The syntax tree, graphically:



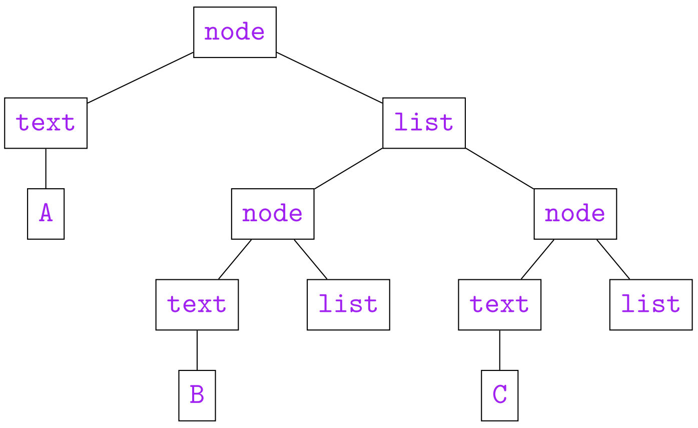
The syntax tree as a program:

```
node "list"
  [ node "text" [ node "a" [] ]
  , node "++"
      [ node "text" [ node "b" [] ]
      , node "text" [ node "c" [] ] ]
  , node "show"
      [ node "+"
          [ node "num" [ node "2" [] ]
          , node "num" [ node "3" [] ] ] ] ]
```

An example to illustrate trees:

```
node "A" [ node "B" [] , node "C" [] ]
```

The syntax tree, graphically:



The syntax tree as a program:

```
node "node"
  [ node "text" [ node "A" [] ]
  , node "list"
      [ node "node"
          [ node "text" [ node "B" [] ] , node "list" [] ]
      , node "node"
          [ node "text" [ node "C" [] ]
          , node "list" [] ] ] ]
```

Meta-programming The broad topic of "programs processing other programs" is called meta-programming. Sometimes meta-programming is given narrower definitions, but we find this one to better account for the different parts of the research field. Meta-programming comprises the generation, analysis, transformation of programs or similar objects such as formulae and proofs. Some programming languages are completely designed around meta-programming to support run-time code generation (MetaML [Taha, 1999], 'C [Engler et al., 1996]). Program translations and optimizations, as done by compilers, are prime examples of meta-programming

1.5 Names and local bindings

Our programming language, while capable of arithmetic computations, interactions, and program representation, lacks the concept of names. We hence introduce a construct to give a name to part of a program and refer to this part using the name. This enables an important goal of re-usability. In a program we should avoid to repeat ourselves. Hence if two parts are the same they may benefit from being written only once. Maybe more importantly, changes to this part of the software logic, during later evolution of the code, only have to be done in one place instead of being duplicated. The syntax of this new construct relies on three keywords `let`, `=` and `in`. If two programs p_1 and p_2 are syntactically valid and `x` is a name then `let x =` p_1 `in` p_2 is syntactically valid. Moreover, names (such as `x`) are valid program themselves. If we want to print the same text twice we can write:

```
let x = print "Hello!" in
x >> x
```

If we replace the occurrences of `x` by its definition (here `print "Hello!"`) we obtain a program with the same behavior:

```
print "Hello!" >> print "Hello!"
```

The gain brought by sharing is significant. To illustrate this fact, we build an artificial example where each added `let` would double the size of the program if we could not use `let`. Albeit artificial this example conveys the fact that sharing makes a big difference. The following example has 54 syntax nodes while the expanded version has 220 syntax nodes (only 124 if we reduce x_0 to `0` first). Note also that the small computation done in x_0 can be performed only once instead of 16 times.

```
let x_0 = 42 ∸ 2 * 21 in
let x_1 = node x_0 [] in
let x_2 = node 1 [ x_1 , x_1 ] in
let x_3 = node 2 [ x_2 , x_2 ] in
let x_4 = node 3 [ x_3 , x_3 ] in
node 4 [ x_4 , x_4 ]
node 4
  [ node 3
      [ node 2
          [ node 1 [ node 0 [] , node 0 [] ]
          , node 1 [ node 0 [] , node 0 [] ] ]
      , node 2
          [ node 1 [ node 0 [] , node 0 [] ]
          , node 1 [ node 0 [] , node 0 [] ] ] ]
  , node 3
      [ node 2
          [ node 1 [ node 0 [] , node 0 [] ]
          , node 1 [ node 0 [] , node 0 [] ] ]
      , node 2
          [ node 1 [ node 0 [] , node 0 [] ]
          , node 1 [ node 0 [] , node 0 [] ] ] ] ]
```

1.6 Scoping

A crucial aspect of the `let` construct is the scope of the newly introduced name. Given a name `x` and two programs p_1 and p_2, the program `let x =` p_1 `in` p_2 defines `x` to be p_1 in p_2. Thus, this makes `x` available in p_2. We say that the scope of the name `x` is the program p_2. We also say that `x` scopes over p_2.

What should we do when a program uses a name that has not been defined? This is obviously an error and so should be detected as soon as possible. Here is an example to illustrate this kind of errors:

```
print x >>   -- x is not defined, hence this is an error
let x = x in -- x is not yet defined
print x      -- here x is defined
```

One may wonder if the same name could be used more than once. The scoping discipline we use allows this. In particular if one reuses a name already defined to define something else the new definition hides the previous one. Here is a correct program to illustrate the various scoping subtle cases:

```
let x = "1" in
print x >>                          -- this prints 1
let x = "2" in
print x >>                          -- this prints 2
print (let x = "3" in x) >>         -- this prints 3
(let x = "4" in print x) >>         -- this prints 4
print x >>                          -- this prints 2 again
let x = x ++ x in
print x >>                          -- this prints 22
let x = let x = "5" in x ++ x in
print x                            -- this prints 55
```

We now introduce some vocabulary about names. In the program `let x = p`$_1$ `in p`$_2$, the name `x` is said to be *bound*. In particular it is bound in `p`$_2$. We call the name `x` in this position a *binder*. In the program `print x`, the name `x` is said to be *free*. We also call it an *occurrence* of `x`.

Sometimes names are also called *variables*. We try to prefer the term *name* over *variable* in this document. More precisely we use the term *variable* to represent the construct which holds just a name. For instance the program `x + x` contains two variables but only one name.

A program without any free names/variables is said to be closed. The set of free names can be defined inductively as follows. The set of free names of a variable `x` is the singleton set with the name `x`. The free names of an operation such as `print p` or `show p` are the free names of `p`. The free names of an operation such as `p`$_1$ `+ p`$_2$ are the union of free names of `p`$_1$ and of `p`$_2$. Finally the free names of `let x = p`$_1$ `in p`$_2$ are the free names of `p`$_2$ minus `x`, union the free names of `p`$_1$.

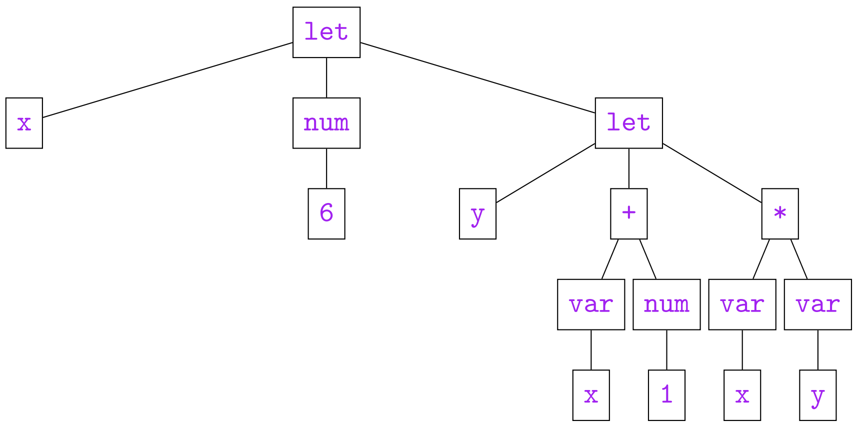
A name `x` is said to be *fresh* for a program `p` if the name `x` is not a member of the free variables of `p`. For example the name `y` is fresh for the program `let y = 42 in x + y`, but the name `x` is not.

Two programs `p`$_1$ and `p`$_2$ are said to be *α-equivalent* if they differ only by a consistent renaming of the bound names. Note that this is not a formal description of α-equivalence, but an informal definition appealing to our intuition of “name irrelevance”. Defining this relation precisely is one of the important part of this thesis. For instance the programs `let x = 21 in x + x` and `let y = 21 in y + y` are α-equivalent. The name of a variable should not have any importance except being a tool to reference a position in the program without ambiguities. α-equivalence is an equivalence relation. This means that the relation is reflexive (every program is α-equivalent to itself), symmetric (if `p`$_1$ is α-equivalent to `p`$_2$, then `p`$_2$ is α-equivalent to `p`$_1$), and transitive (if `p`$_1$ is α-equivalent to `p`$_2$ and `p`$_2$ is α-equivalent to `p`$_3$ then `p`$_1$ is α-equivalent to `p`$_3$). The α-equivalence is also a congruence, meaning that if `p`$_1$ is α-equivalent to `p`$_2$ then if we put the programs in the *context* `C[_]` to produce bigger programs then `C[ p`$_1$ `]` is α-equivalent to `C[ p`$_2$ `]`.

Typing our `let` construct We have introduced local definitions and variables and we now extend our typing discipline to these constructs. Like we have done for the previous constructs, we give the typing informally using a textual description. The program `let x =` p_1 `in` p_2 has type σ if and only if p_1 has some type τ and that p_2 has type σ *assuming* that `x` has type τ. To type a variable we use the assumptions we gathered so far. There are several ways to precisely describe how to manage these assumptions, but we do not detail them in this introduction.

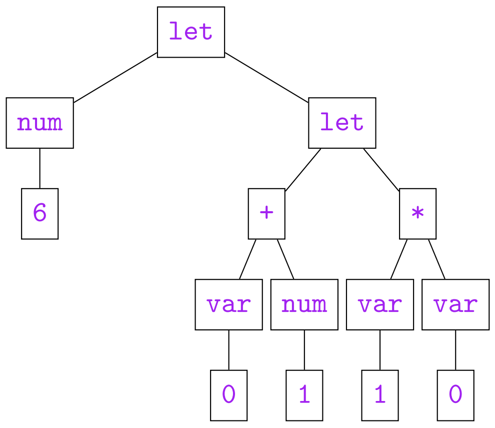
Let us try to check the typing with an example. We check that the program `let x = 42 in let y = show x in print y` has type `Interactive`. To do so we first check that `42` has type $\mathbb{N}$ which is true, then we must check that `let y = show x in print y` has type `Interactive` assuming that `x` has type $\mathbb{N}$. To do so we first check that `show x` has type `String`, which given the rule for `show` amounts to check that `x` has type $\mathbb{N}$, which is true by assumption. We then check that `print y` has type `Interactive` assuming that `y` has type `String`. Given the rule for `print` this amounts to check that `y` has type `String` which is true by assumption.

Meta-programming and representation of variables How do we represent variables and the `let` construct? It is reasonable to start with names being values of type `String`, each variable being a tree node that we call `var`, and the `let` construct being a node with three subtrees. The program `let x = 6 in let y = x + 1 in x * y` can thus be depicted as:



This first approach is the root of the *nominal* approach that we describe in greater detail in chapter 2.

A different approach is to use de Bruijn indices [de Bruijn, 1972]. This representation is said to be nameless because variables are no longer identified by a name but a notion of “distance” to the binding point. This nameless approach solves part of the problem by providing a canonical representation. However a major issue with this nameless representation is its arithmetic flavor. Indeed properties about names and binders are turned into arithmetic formulae. This second approach is covered in chapter 5. Here is the same example using the de Bruijn style:



1.7 Empowering our language

Declarations and definitions We now fast-forward from our tiny subset of AGDA to AGDA itself as we need it in the remainder of this document. We start with declarations, which allow us to declare a symbol that can be used globally. This generally differs from the `let` construct whose scope is local. Moreover the declaration just specifies the type of the symbol. Subsequent phrases have to define this declared symbol. The declarations make use of the character `:` to separate the declared symbol from its type. The following line is read "Dear AGDA, let us declare `hello-world` of type `Interactive`":

```
hello-world : Interactive
```

After the declaration must come the definition. The name of the defined symbol is recalled and a program is given for its definition:

```
hello-world = print "Hello World!"
```

Functions We introduce a new sort of types, namely function types. If σ and τ are types then $\sigma \rightarrow \tau$ is a type as well. This type represents functions whose domain is σ and co-domain is τ. Happily AGDA functions are like mathematical functions and no surprise whatsoever will trouble this. The syntax for definitions enables the definitions of functions in a very simple way: we just give a name to the argument before the equal sign. To apply a user defined function `f` to an argument `x`, the syntax is the lightest syntax possible: `f x`.

```
double : ℕ → ℕ
double n = n + n

-- This program prints: 42
prog₁ : Interactive
prog₁ = print (show (double 21))
```

The name `n` before the equal sign is a binder and scopes over the sub-program found after the equal sign. The scoping works pretty much like with the `let` construct. Here is another example:

```
hello : String → Interactive
hello s = print ("Hello " ++ s ++ "!")

-- This program prints: Hello Functional World!
prog₂ : Interactive
prog₂ = hello "Functional World"
```

In order to receive multiple arguments one simply has to make functions return functions. Indeed the type `ℕ → (ℕ → ℕ)` is the type of a function taking a first number argument and returning a function taking a second number argument to finally deliver its result number. This is a pattern so common that we make the arrow type associate on right such that we can write the type this way: `ℕ → ℕ → ℕ`. The same shortcut goes for function applications. Instead of writing `(f x) y` or worse `(f(x))(y)` we simply make the application associate on left, thus we can write `f x y`. Here is an example:

```
hello₂ : String → ℕ → Interactive
hello₂ s n = print ("Hello " ++ s ++ " " ++ show n ++ "!")

-- This program prints: Hello World 42!
prog₂ : Interactive
prog₂ = hello₂ "World" (2 * 21)
```

Data-types Among the types presented so far, `ℕ`, `List`, `Tree` are user-defined in AGDA. The general mechanism used to define these types is called *inductive families*. Inductive families generalize various forms of definition mechanisms such as *sum and product types*, *regular tree types*, *algebraic* *data types*, and *GADTs* (generalized algebraic data-types). To define an inductive family, we need to declare its name and type, and declare as many *data constructors* as we want. Each data constructor has a name and a type as well. Let us study the definition for Bool, one of the simplest data type possible:

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

The first line declares Bool to have type Set. Indeed Set is a special type which is the type of basic types. Agda treats types like other values of the language. Then we define two data constructors called false and true. These two constructors are the two only values of type Bool.

We now focus on the type of natural numbers, namely ℕ:

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

The first line declares ℕ to have type Set. Then we define two data constructors called zero and suc. The constructor zero is introduced as a value of type ℕ to represent the number 0. The constructor suc (for successor) is introduced as a function from ℕ to ℕ which might seem like a surprising beast. Two points are striking. First we are using the type ℕ in its own definition: this type is said to be *recursive*. The second point is to declare a function without giving it a definition. Indeed suc has no other definition, and thus giving it an argument does not trigger a computation. For instance suc zero stays stuck like that and does not reduce further. The language is built to take advantage of data constructors to define functions by *pattern matching*.

Note that the same name can be used for different data constructors. Agda makes use of type annotations to resolve ambiguities.

To illustrate definitions made with pattern matching the following program defines a function called not which negates a Boolean value:

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

This definition is made of two equations: one for each constructor of the inspected argument. In the case for true we return false and in the case for false we return true.

We now go on a more interesting definition made with pattern matching.

The following program defines a function called `triple` which multiplies by three its argument:

```
triple : ℕ → ℕ

-- 3 * 0 = 0
triple zero    = zero

-- 3 * (1 + n) = 3 + 3 * n
triple (suc n) = suc (suc (suc (triple n)))
```

While this definition is longer than we could expect (`triple n = 3 * n`), it has a pedagogical interest. This definition is made of two equations: one for each constructor of the inspected argument. In the successor case the function `triple` is used in the definition of `triple` itself! This function is said to be recursive.

We can now reveal that all the operations on natural numbers we have seen so far are actually defined within the language using recursive definitions. Here are for instance the definitions for `+` and `*`. To express that we want these operators to be infix, we declare `+` as `_+_`. Each `_` indicates special places where operands should go. Both functions `_+_` and `_*_` pattern-match on their first argument:

```
_+_ : ℕ → ℕ → ℕ
zero  + n = n               -- 0 + n = n
suc m + n = suc (m + n)     -- (1 + m) + n = 1 + (m + n)

_*_ : ℕ → ℕ → ℕ
zero  * n = zero            -- 0 * n = 0
suc m * n = n + m * n       -- (1 + m) * n = n + m * n
```

The notation we used for literal numbers such as `2` or `4` is actually strictly equivalent to `suc (suc zero)` and `suc (suc (suc (suc zero)))` respectively.

We define a last operation on natural numbers, namely division by two. We do not show the definition for `÷`. Indeed, division by a known constant is indeed much simpler to define recursively and illustrates subtler pattern matching. The function is declared as `_/2` and hence has to be used either prefix as `_/2 4` or postfix as `4 /2`. You can notice how spaces are of prime importance in AGDA since they allow to freely choose meaningful names for functions. The function `_/2` proceeds as follows. The two base cases for zero and one are handled in the first two equations. The last equation deals with all numbers strictly greater than one which makes a simple recursive call to obtain our result:

```
_/2 : ℕ → ℕ

-- 0 / 2 = 0
zero /2 = zero

-- 1 / 2 = 0
suc zero /2 = zero

-- (2 + n) / 2 = 1 + n / 2
suc (suc n) /2 = suc (n /2)
```

Polymorphism Sometimes functions do not need to know the nature of parts of their arguments. This means that many types could be acceptable for such functions. An extreme case is the identity function which simply returns its argument. What type should we give it? `ℕ → ℕ`, `String → String`, or `List ℕ → List ℕ` and the list goes on indefinitely. The solution to this issue is called *polymorphism*. We generalize the type of the function type to `∀{A} → A → A`, which reads "for all type `A` a function from `A` to `A`". The polymorphic identity function (`id`) can thus be written:

```
id : ∀{A} → A → A
id x = x

-- The function id can be used at different types
prog₃ : Interactive
prog₃ = print (id "Hello " ++ show (id 42))
```

Lists We define two new data types that we assumed to be special so far, namely `List` and `Tree`. The type for lists –such as the type `ℕ`– is made of two data constructors. A base constructor named `[]` and pronounced "nil" represents the empty list. Another constructor named `_::_` and pronounced "cons" appends one element to the front of a list. The notation `[ n₁ , n₂ , ... ]` is simply a shorthand for the less familiar `n₁ :: n₂ ... :: []`.

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

Plenty of interesting functions can be defined recursively on lists. We show one of them which while being simple can be insightful. The function `length` takes a list and returns its length as a natural number. The function is defined with two equations: one for the empty list whose length is zero, and one for any constructed list whose length is the successor of the length of the *tail* of the list. In short this function replaces the lists constructors by the constructors of natural numbers. This highlights the fact that both types share the same structures. Natural numbers are lists of meaningless elements and lists are natural numbers whose constructors are annotated by elements. The function `length` is polymorphic. It takes a list, forgets the elements to reveal the bare structure behind any list: a natural number.

```
length : ∀{A} → List A → ℕ
length []       = zero
length (x :: xs) = suc (length xs)
```

Let us remark that since we make no use of the name `x` in the second equation for `length` we could have use the special *wildcard* pattern. The wildcard pattern is noted _ and is used to replace a name to state that we do not use this part of the value.

Another common function on lists is the function `map`. The function `map` takes a function `f` from a type `A` to a type `B`, and also takes a list of type `List A` on which it applies `f` on every element to build the resulting list of type `List B`. Since `map` takes a function as argument it is what we call an *higher-order* function. Moreover, the function `map` is also recursive and polymorphic. Here is the definition of `map`:

```
map : {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs
```

We now introduce an extension to pattern matching equations, namely the `with` construct. This construct extends a pattern-matching-based definition with new columns. This construct is of great effect when combined with dependent pattern-matching. However, we present it here on a simpler example, the function to filter a lists. The function `filter`, takes a predicate `p` and a list `xs` and keeps only the elements of `xS` which statisfy the predicate `p`. The `with` construct is used here to select a branch according to the result of the predicate:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
filter p (x :: xs) | true  = x :: filter p xs
filter p (x :: xs) | false = filter p xs
```

Moreover, an ellipsis `...` can be used to elide a redundant equation prefix. Hence we can write `filter`, this way:

```
filter : {A : Set} → (A → Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
...               | true  = x :: filter p xs
...               | false = filter p xs
```

Trees and forests We continue with trees the exploration of our previously assumed basic types. Trees can be defined with a new data type. This data type `Tree` has a single data constructor named `node` with two arguments: the node label of type `A` and the children as a list of subtrees (`List (Tree A)`).

```
data Tree (A : Set) : Set where
  node : A → List (Tree A) → Tree A
```

Commonly we call forests the lists of trees. The name is suggestive and the type is shorter to write. Its definition shows how types are treated such as other values:

```
Forest : Set → Set
Forest A = List (Tree A)
```

Trees and forests are interdependent. They finally build a pair of *mutually recursive* types. This is the occasion to define a pair of mutually recursive functions. We define `sumTree` and `sumForest` which respectively compute the sum of the labels in a tree of natural numbers and in a forest of natural numbers. For Agda to accept these definitions, the two declarations must come before the two definitions.

```
sumTree   : Tree ℕ   → ℕ
sumForest : Forest ℕ → ℕ

sumTree   (node n forest) = n + sumForest forest
sumForest []              = 0
sumForest (tree :: forest) = sumTree tree + sumForest forest
```

Robust program representations So far we have seen three distinct types to represent programs: text values (`String`), lists of tokens (`List String`), and trees (`Tree String`). These three types are based on the type `String` and thus they are loose representations. For instance the tokens can be malformed tokens and the tree labels could be malformed as well. We could introduce data types for tokens and tree labels which rule out the malformed text values. However we can do even better. Here is a data type for our tiny programming language of the beginning:

```
data Program : Set where
  ‘num   : ℕ → Program
  ‘text  : String → Program
  ‘list  : List Program → Program
  ‘node  : Program → Program → Program
  ‘+     : Program → Program → Program
  ‘∸     : Program → Program → Program
  ‘*     : Program → Program → Program
  ‘÷     : Program → Program → Program
  ‘++    : Program → Program → Program
  ‘print : Program → Program
  ‘show  : Program → Program
  ‘>>     : Program → Program → Program
  ‘var   : String → Program
  ‘let   : String → Program → Program → Program
```

The type Program is a recursive type inducing a specialized tree structure. Each construct is modeled by a single data constructor which is named with the corresponding label we have for trees plus an extra ‘ character. The data constructor also specifies how many subtrees are expected and what kind of subtree is expected. Here are two of our previous examples represented with our type Program:

```
prog₄ : Program
prog₄ = ‘* (‘num 2) (‘+ (‘num 10) (‘num 11))
```

```
prog₅ : Program
prog₅ = ‘>>
          (‘print (‘text "Hello! 2 times 21 is equal to "))
          (‘print (‘show (‘* (‘num 2) (‘num 21))))
```

Here are now a few programs that cannot be represented with the data type Program. Indeed while all correct programs can be represented, this simple data type already rules out plenty of wrong programs. Here is first a list of syntactically wrong programs represented as texts:

```
[ "2 *"   -- missing operand
, "(2"    -- missing parenthesis
, "2 {- " -- non-closed comment
, "bla"   -- unknown construct or variable
]
```

Here is now a list of still syntactically wrong programs represented as trees:

```
[ node "+" [ p ]                   -- missing operand
, node "+" [ p₁ , p₂ , p₃ ]        -- extra operand
, node "bla" []                    -- unknown construct
, node "var" [ node "bla" [] ]     -- unknown variable
]
```

Finally, there are still wrong programs accepted by all of our representations. Those are the ill-scoped programs and the ill-typed programs. Here is a list of ill-scoped or ill-typed programs using the type `Program`:

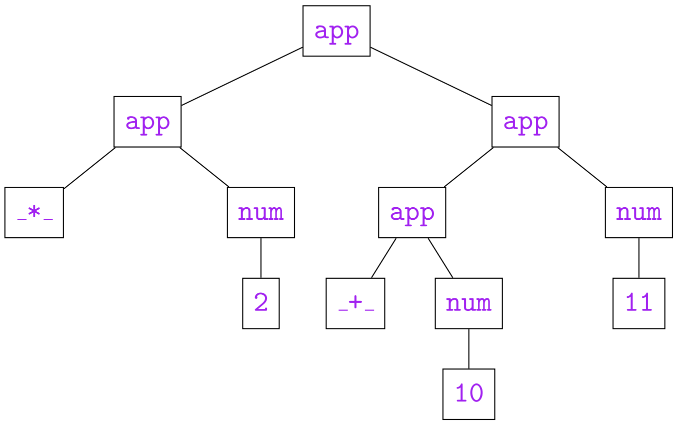
```
[ ‘var "bla"                       -- unknown variable
, ‘+ (‘num 1) (‘text "Hello!")     -- ill-typed
, ‘let "x" (‘var "x") (‘num 1)     -- unknown variable
]
```

The rest of this document focuses on how to improve program representation to better account for the handling of variables in programs.

Declaring the last operations We have shown how various constructs of our tiny programming language can be defined in Agda. We now quickly declare the remaining constructs that can be Agda functions. Thus we declare `_∸_`, `_÷_`, `show`, `_++_`, `print`, and `_>>_`. While we give no definitions for these functions the declarations describe both their syntax and their typing very concisely:

```
_∸_    : ℕ → ℕ → ℕ
_÷_    : ℕ → ℕ → ℕ
show   : ℕ → String
_++_   : String → String → String
print  : String → Interactive
_>>_   : Interactive → Interactive → Interactive
```

The core constructs from our tiny language that we do not define are literals (numbers, texts, lists), variables and the `let` construct. However, in order to use all these functions and constructors we introduced a fairly-discrete construct, namely application. Since application is written as a simple juxtaposition (using spaces or parentheses) we might forget it. In our tree representation we use the label `"app"` for these application nodes. We can now view the example program `2 * (10 + 11)` as a tree where the application nodes replace the special nodes of operations:



Agda types In Agda, the usual function space is written `A → B`, while the dependent function space is written `(x : A) → B` or `∀ (x : A) → B`. An implicit parameter, introduced via `∀{x : A} → B`, can be omitted at a call site if its value can be inferred from the context. There are shortcuts for introducing multiple arguments at once or for omitting a type annotation, as in `∀{A} {i j : A} x → e`.

In Agda, `Set` (or `Set₀`) is the type of small types such as `ℕ`, `List String`, and `Maybe (Bool × ℕ)`. `Set₁` is the type of `Set`, `Set → Bool`, `ℕ → Set`, and `Set → Set`.

There is no specific sort for propositions in Agda: everything is in `Set ℓ` for some `ℓ`. The unit type is a record type with no fields named `⊤`. It also represents the `True` proposition. The empty type is an (inductive) data type with no constructors named `⊥`. It also represents the `False` proposition. The negation `¬ A` is defined as `A → ⊥`.

Lexical conventions We recall that Agda is strict about whitespace: `x≤y` is an identifier, whereas `x ≤ y` is an application. This allows naming a variable after its type (deprived of any whitespace). For example: `x≤y` might be a variable of type `x ≤ y`, that is, a proof of `x ≤ y`.

Various notions of equality Advanced logics and proof systems exhibit various forms of equality. The *definitional equality* is a first form of equality which is deeply rooted in the computation rules of the programming language itself, here, Agda. Two terms are *definitionally equal* if and only if they can both reduce to a common term. In AGDA, the reduction is a combination of β-reductions, η-conversions, and application of the definition equations.

Here are a few examples of definitionally equal terms. We informally use = for this equality since this symbol is the one used for definitions.

```
(λ x → x) suc zero = suc zero
suc = λ x → suc x
zero + n = n
```

Here is an example of two terms which are not definitionally equal even if this equality seems natural:

```
n + zero ≢ n
```

Definitional equality is completely automatic and thus requires no help from the user. However if we want to include some reasoning steps we can use the *propositional equality*. The symbol used in AGDA and further in this document is ≡. To simplify matters, here is the definition of the propositional equality specialized to values of small types (Set₀), namely _≡₀_:

```
data _≡₀_ {A : Set₀} (x : A) : A → Set₀ where
  refl : x ≡₀ x

_≢₀_ : {A : Set₀} → A → A → Set₀
x ≢₀ y = ¬(x ≡₀ y)
```

We do not intend to explain the subtleties of such a definition. We only mention that there is a single constructor for this type requiring both sides to be same, hence definitionally equal.

```
-- This is definitionally true, hence immediately proven by refl
zero + n ≡ n

-- This is provable by a simple induction on n
∀ n → n + zero ≡ n
```

Sometimes, two functions f and g cannot be proved propositionally equal but still produce the same output on every possible input. In this case, the functions f and g are said to be *equal pointwise*. The symbol for pointwise equality is ≗. Here is the definition of pointwise equality in AGDA, specialized to small types (Set₀), non-dependent functions:

```
_≗₀_ : {A B : Set₀} (f g : A → B) → Set₀
f ≗₀ g = ∀ x → f x ≡₀ g x
```

To illustrate pointwise equality we define the function `not²` which iterates the function `not` two times.

```
not² : Bool → Bool
not² x = not (not x)
```

We then show that the function `not²` is equal pointwise to the identity function. Given the definitions for `_≗₀_` and `not²` this exactly amounts to show: `∀ x → not (not x) ≡ x`.

```
not²≗id : not² ≗₀ id
not²≗id true  = refl -- true = not (not true) definitionally
not²≗id false = refl -- false = not (not false) definitionally
```

Equalities are said to be either intensional or extensional. Let us use sorting functions as an example to illustrate the difference. If we have two different sorting functions each implementing one sorting algorithm such as bubble sort and quick sort. The two functions are said to be extensionally equal since they both return the same sorted list for any given list. However bubble sort and quick sort are not intentionally equal. The *intention* is actually opposite. When you write a quick sort function, we intentionally want to make it different than bubble sort!

In Agda both the definitional and the propositional equalities are currently intensional. There is an ongoing will to make the propositional equality more extensional. While this sounds like a radical change, the propositional equality is actually already compatible with extensionality. Indeed, there is no way to contradict extensionality with the actually propositional equality. In short there is no proof than quick sort is different from bubble sort and thus making them provably equal introduce no contradiction.

The *pointwise equality* is by construction extensional and in section 4.2 we explain a generalized definition to relate functions which sends related inputs to related outputs.

Full development online We use some definitions from Agda's standard library: natural numbers, booleans, lists, and applicative functors (`pure` and `_⊛_`).

For the sake of conciseness, the code fragments presented in this document are sometimes not perfectly self-contained. However, a complete Agda development is available online [Pouillard, 2011a].

Outline This document is organized as follows.

The next three chapters delve into the nominal approach. We introduce the nominal approach (chapter 2) and present our system: a safe programming interface for the nominal style. We then present how to use it (chapter 3), and what happens behind the scene to implement it and show its safety (chapter 4).

The next chapter (5) covers a nameless approach known as "de Bruijn indices". We present how to seamlessly extend our system to this new approach. This chapter covers the same aspects: the interface and its implementation, its usage, its soundness.

The remaining chapter (6 discusses different variations and combinations of these variations. At the same time this chapter covers the related work and concludes.

Chapter 2

The nominal approach

This chapter is organized as follows. The first section informally introduces and presents several techniques to represent data structures with bindings in a nominal style. Section 2.2 describe our solution, an interface to program with names and binders.

2.1 Introduction to the nominal approach

2.1.1 Warm-up: the bare nominal approach

The bare approach to abstract syntax with names and binders in nominal style requires very little infrastructure to start with. Names and binders are represented by so-called atoms. The set of atoms is countably infinite and the only required operation is an equality test. Using natural numbers as a concrete representation for atoms is a common and sensible choice.

```
-- A set of atoms (could be ℕ)
Atom : Set

-- Atom is countably infinite; here are some atoms:
-- x,y,z... could be represented by 0,1,2...
x y z f g ... : Atom

-- The equality test on atoms
_==ᴬ_ : (x y : Atom) → Bool
```

Given the type `Atom` we can readily define algebraic data types for abstract syntax with names and binders. Our running example is the untyped λ-calculus defined below. The type `Tmᴬ` (`Tm` for “term” and `ᴬ` for “atom”) is made of three data constructors. The constructor `V` is for variables and simply holds an atom. The constructor `_·_` is the function application construct made of two subterms. The constructor `ƛ` takes an atom and a subterm in which this atom is considered bound. This means that the construct ƛ is introducing a variable. On contrary the atom in the construct V is said to be free. Indeed this atom is supposed to refer to a bound atom. The constructor Let also takes an atom but two subterms. The atom is bound in the second subterm only.

```
data TmA : Set where
  V   : (x : Atom) → TmA
  _·_ : (t u : TmA) → TmA
  ƛ   : (b : Atom) (t : TmA) → TmA
  Let : (b : Atom) (t u : TmA) → TmA
```

It is striking that there is no distinction between the atoms in bound positions and the ones in free positions. There is neither an indication of the scope of the free atoms, nor an indication of which scope extends a variable in a binding position. This calls for further improvements.

We consider it very important to highlight that atoms are used for two distinct purposes whether they are in a binding position or a free position. Starting from section 2.2, we embrace that distinction and provide distinct types for these usages.

Here are two term examples, the identity function and the application function:

```
-- λx. x
idTmA : TmA
idTmA = ƛ x (V x)

-- λf. λx. f x
apTmA : TmA
apTmA = ƛ f (ƛ x (V f · V x))
```

The strength of this approach resides in its simplicity. The representation of terms closely follows the concrete syntax of the language. The main issue is adequacy: there are multiple equivalent representations of the same term.

Indeed the choice of atoms is fairly arbitrary. Two terms can represent the same piece of syntax when they differ only by a consistent renaming of bound names. In this situation two such terms are said to be α-equivalent. Here are for example two α-equivalent terms:

```
tx : TmA
tx = ƛ x (V f · V x)
ty : TmA
ty = ƛ y (V f · V y)
```

The term t^x is α-equivalent to t^y since consistently renaming the bound name x by y in the term t^x yields the term t^y. Respectively consistently renaming y by x in t^y yields t^x. However α-equivalence can be subtle, let us observe a third term t^f which is *not* α-equivalent to neither t^x nor t^y:

```
tᶠ : Tmᴬ
tᶠ = ƛ f (V f · V f)
```

An *inconsistent* renaming of the bound name x by f in t^x yields t^f, while a consistent renaming of the bound name f by x in t^f does *not* yield t^x.

We do not delve more into the details of α-equivalence yet. We focus on the fact that the concrete naming of a term is a representation issue and should not be relevant for the computation. In particular, good functions should be independent of this representation issue. This property can be stated as follows:

Well behaving functions should return α-equivalent outputs given α-equivalent inputs.

To illustrate well behaving functions we give a few examples. The following functions rm^A (removes an atom from a list) and fv (lists the free variables/atoms of a term) are well behaving:

```
rmᴬ : Atom → List Atom → List Atom
rmᴬ _ []        = []
rmᴬ x (y :: ys) =
  if x ==ᴬ y then rmᴬ x ys
             else y :: rmᴬ x ys

-- Since rmᴬ behaves well, this holds:
test-rmᴬ : rmᴬ x [ x ] ≡ rmᴬ y [ y ]
test-rmᴬ = refl -- both reduces to []

fv : Tmᴬ → List Atom
fv (V x)         = [ x ]
fv (t · u)       = fv t ++ fv u
fv (ƛ x t)       = rmᴬ x (fv t)
fv (Let x t u) = fv t ++ rmᴬ x (fv u)

-- Since fv behaves well, this holds:
test-fv : fv tˣ ≡ fv tʸ
test-fv = refl -- both reduces to [ f ]
```

We now illustrate misbehaving functions with two examples. The function `ba` computes the list of bound atoms. The function `cmp-ba` takes two terms and compare their bound variables if both are `ƛ` constructs.

```
ba : Tmᴬ → List Atom
ba (ƛ x t)      = x :: ba t
ba (Let x t u) = x :: ba t :: ba u
ba (t · u)      = ba t ++ ba u
ba (V _)        = []

[x]≢[y] : [ x ] ≢ [ y ]
[x]≢[y] = {! omitted !}

-- Since ba does not behave well:
test-ba : ba tˣ ≢ ba tʸ
test-ba = [x]≢[y]

cmp-ba : Tmᴬ → Tmᴬ → Bool
cmp-ba (ƛ x _) (ƛ y _) = x ==ᴬ y
cmp-ba _       _       = false

-- Since cmp-ba does not behave well:
test-cmp-ba : cmp-ba tˣ tˣ ≢ cmp-ba tˣ tʸ
test-cmp-ba () -- true ≢ false
```

In traditional informal developments, α-equivalence is identified with equality. The upside is that every definition respects α-equivalence. Hence functions automatically map α-equivalent inputs to α-equivalent outputs. The downside is that building functions is more difficult. Indeed what were misbehaving functions before, are not functions anymore when we identify α-equivalence and equality. They are not functions simply because they map equal inputs to different outputs.

One of the goals of our system is to guarantee that well-typed functions are well-behaving. A function like `cmp-ba` is ill-typed in our system. A function like `ba` is typed differently than the function `fv`, effectively disarming the function `ba`.

2.1.2 Using well-formedness judgements

One way to define the scoping rules is to define a well-formedness judgement. It can be done using a recursive predicate over the structure of terms. It is a well-known technique used in the definition of type-systems. The standard presentation makes use of a set of inference rules where judgements are made of an environment, a term and a type. The environment tracks the type of each introduced variables. To specify scoping rules one follows the same presentation but without the types. We directly focus on a formal presentation in AGDA since inferences rules (and grammars) have a direct translation into inductive types. First one defines the environments which are nothing more than lists of atoms:

```
data Env : Set where
  ϵ   : Env
  _,_ : (Γ : Env) (x : Atom) → Env
```

Then comes the definition of the membership predicate. It states when an atom is a member of some environment. This is done through two rules: one applies when the atom is present at the first position of the environment, and a second rule applies when it is present in the tail of the environment. Here is the definition in AGDA where lines drawn with comments give constructors a flavor of inference rules:

```
data _∈_ x : (Γ : Env) → Set where

  here  :                -------------
                          x ∈ (Γ , x)

  there : ∀ {y}
                       →   x ∈ Γ
                       → -------------
                          x ∈ (Γ , y)
```

Finally one can state the scoping rules for the λ-calculus. The definition boils down to using an environment to track bound atoms, using the membership predicate at variable occurrences and pushing the binders onto the environment:

```
data _⊢_ Γ : Tmᴬ → Set where
  V : ∀ {x}          →  x ∈ Γ
                     → ---------
                       Γ ⊢ V x

  _·_ : ∀ {t u}      →  Γ ⊢ t
                     →  Γ ⊢ u
                     → -----------
                       Γ ⊢ t · u

  ƛ : ∀ {t b}        →  Γ , b ⊢ t
                     → -----------
                       Γ ⊢ ƛ b t

  Let : ∀ {t u b}    →  Γ ⊢ t
                     →  Γ , b ⊢ u
                     → ---------------
                       Γ ⊢ Let b t u
```

We can now state that our example terms are well-scoped in the empty environment. Thanks to implicit arguments and to the definition of the membership predicate, the proof that a term is well scoped is the term itself in de Bruijn notation (`zero` is `here` and `suc` is `there`).

```
⊢id : ϵ ⊢ idTmᴬ
⊢id = ƛ (V here)

⊢ap : ϵ ⊢ apTmᴬ
⊢ap = ƛ (ƛ (V (there here) · V here))
```

The use of Agda implicit arguments keep these examples concise. The curious reader who is unfamiliar with Agda might find interesting to look at the same (second) example with all arguments made explicit:

```
⊢ap′ : ϵ ⊢ apTmᴬ
⊢ap′ = ƛ {ϵ} {ƛ x (V f · V x)} {f}
         (ƛ {ϵ , f} {V f · V x} {x}
            (V {ϵ , f , x} {f}
               (there {f} {ϵ , f} {x}
                  (here {f} {ϵ})) ·
             V {ϵ , f , x} {x}
               (here {x} {ϵ , f})))
```

However this technique is not directly what we are looking for. Indeed these scoping properties are explicitly stated on the side of each definition. We are looking for something more integrated into the types. This would enable well-formedness to be enforced in a more pervasive and automatic manner. Happily this technique can be adapted to merge the scoping information and the term.

2.1.3 Well-scoped terms

Merging terms and scoping rules is not a new technique and amounts to removing the term index of the scoping predicate. The resulting term type is indexed by an environment. This environment is extended at binders and queried at variable occurrences:

```
data Tmᴶ Γ : Set where
  V    : ∀ {x} → x ∈ Γ → Tmᴶ Γ
  _·_  : Tmᴶ Γ → Tmᴶ Γ → Tmᴶ Γ
  ƛ    : ∀ b → Tmᴶ (Γ , b) → Tmᴶ Γ
  Let  : ∀ b → Tmᴶ Γ → Tmᴶ (Γ , b) → Tmᴶ Γ
```

One can define our two term examples in this style as well:

```
idTmᴶ : Tmᴶ ϵ
idTmᴶ = ƛ x (V here)

apTmᴶ : Tmᴶ ϵ
apTmᴶ = ƛ f (ƛ x (V (there here) · V here))
```

The function `fv` can easily be adapted to this style, while the function `rmᴬ` can be reused:

```
fv : ∀ {Γ} → Tmᴶ Γ → List Atom
fv (V {x} _)     = [ x ]
fv (t · u)       = fv t ++ fv u
fv (ƛ x t)       = rmᴬ x (fv t)
fv (Let x t u)   = fv t ++ rmᴬ x (fv u)
```

Alas the functions `ba` and `cmp-ba` are not rejected by this style. Their types becomes:

```
ba : ∀ {Γ} → Tmᴶ Γ → List Atom
cmp-ba : ∀ {Γ₁ Γ₂} → Tmᴶ Γ₁ → Tmᴶ Γ₂ → Bool
```

We have described the scoping rules but have not made much progress. Indeed two different but α-equivalent terms can still distinguished. Worse, by introducing environments and membership proofs as explicit objects, we have introduced new issues. Functions receiving terms now also receive or know the environment in which these terms are scoped. This extra information can be used by functions making them less agnostic to choices of free variables. Here are two functions using the environment by pattern-matching:

```
fast-fv : ∀ {Γ} → Tmᴶ Γ → List Atom
fast-fv {ε} _ = []  -- for sure the term is closed
fast-fv     t = fv t

env-length : Env → ℕ
env-length ε       = 0
env-length (Γ , _) = 1 + env-length Γ

env-length-Tmᴶ : ∀ {Γ} → Tmᴶ Γ → ℕ
env-length-Tmᴶ {Γ} _ = env-length Γ
```

If one consider the term as the only input of `env-length-Tmᴶ`, then this function does not behave well since the same term can be scoped in two environments of different length. The issue is that Γ is a concrete input to the functions like `fv`, `ba`, `cmp-ba`, `fast-fv` and `env-length-Tmᴶ`.

In order to fix these issues we make use of abstract types and yet provide an interface with precise types for a safe usage. Environments (the type `Env`) are abstracted as so-called *worlds*. Atoms used in binding positions are abstracted as so-called *binders* and cannot even be compared. Atoms in free positions are glued together with the proof of their membership in the environment and are called *names*. Finally a few extra operations allow to deal with these abstract types in a safe way.

2.2 NomPa: A programming interface in nominal style

Our approach is called NomPa and proposes a programming interface whose types are abstract. This interface (Figure 2.1) admits a concrete implementation (section 4.1). Hiding this implementation keeps the programmer away from the internal details and gives rise to interesting properties via parametricity.

While the complete interface is given in Figure 2.1, each ingredient of the interface is be presented in turn. First here are the building blocks needed to define data types with names and binders in a nominal style.

```
record NomPa : Set₁ where
  constructor mk

  infixr 5 _◁_
  infix 3 _⊆_
  infix 2 _#_

  field
    -- minimal kit to define types
    World  : Set
    Name   : World → Set
    Binder : Set
    _◁_    : Binder → World → World

    -- An infinite set of binders
    zeroᴮ : Binder
    sucᴮ  : Binder → Binder

    -- Converting names and binders back and forth
    nameᴮ   : ∀ {α} b → Name (b ◁ α)
    binderᴺ : ∀ {α} → Name α → Binder

    -- There is no name in the empty world
    ∅        : World
    ¬Name∅   : ¬ (Name ∅)

    -- Names are comparable and exportable
    _==ᴺ_   : ∀ {α} (x y : Name α) → Bool
    exportᴺ : ∀ {α b} → Name (b ◁ α) → Name (b ◁ ∅) ⊎ Name α

    -- The fresh-for relation
    _#_  : Binder → World → Set
    _#∅  : ∀ b → b # ∅
    suc# : ∀ {α b} → b # α → (sucᴮ b) # (b ◁ α)

    -- inclusion between worlds
    _⊆_     : World → World → Set
    coerceᴺ : ∀ {α β} → (α ⊆ β) → (Name α → Name β)
    ⊆-refl  : Reflexive _⊆_
    ⊆-trans : Transitive _⊆_
    ⊆-∅     : ∀ {α} → ∅ ⊆ α
    ⊆-◁     : ∀ {α β} b → α ⊆ β → (b ◁ α) ⊆ (b ◁ β)
    ⊆-#     : ∀ {α b} → b # α → α ⊆ (b ◁ α)
```

Figure 2.1: The NomPa interface

2.2.1 All we need to define nominal syntax

To replace environments in the definition of well-scoped terms, we introduce an abstract notion of worlds. Hence the interface starts by an abstract type for worlds:

```
World : Set
```

We consider it very important to highlight that atoms are used for two distinct purposes whether they are in a binding position or a free position. We embrace that distinction and provide distinct types for these usages.

We first introduce a type for binders. Binders are atoms to be used only in binding positions. While these binders are explicitly accessible in terms it does not mean one can distinguish two α-equivalent terms by simply looking at the binders. Indeed the interface does not provide any means to distinguish two binders. Internally binders are really atoms but the interface keeps them abstract:

```
Binder : Set
```

For atoms to be put in free position, we introduce a type called `Name`. This type is indexed by a world in which the name is scoped. A name glues together an atom and a proof of its membership in the world.

```
Name : World → Set
```

Worlds can be thought of as a set of atoms (names or binders) but will be thought of as relations between names in section 4.3.

Now that we have binders and worlds, we can introduce a way to extend a world with a binder. This is analogous to the constructor `_,_` of section 2.1.2 for extending an environment with an atom.

```
_◁_ : Binder → World → World
```

At this point, we have all we need to build data types with names and binders. The new definition of `Tm` is very close to the previous one. All we have to do is to use `_◁_` instead of `_,_`, rename Γ into α, and use `Name` instead of an atom plus a membership proof:

```
data Tm α : Set where
  V   : Name α → Tm α
  _·_ : Tm α → Tm α → Tm α
  ƛ   : ∀ b → Tm (b ◁ α) → Tm α
  Let : ∀ b → Tm α → Tm (b ◁ α) → Tm α
```

Here binding constructs rely on dependent types, since the binder `b` is used in the type of the subterm. In our previous work [Pouillard and Pottier, 2010], we avoided dependent types on purpose to make the results more widely usable. The non-dependent approach based on links can be built on top of the one we are presenting here. A weak link [Pouillard and Pottier, 2010] from α to β is represented by a binder `b` such that β is equal to `b ◃ α`. Hence we want to first study in detail the dependent approach with its logical relation.

Here is a trivial example of a function that traverses a term and measures its size. It is remarkable for its simplicity: name abstractions are traversed without fuss. The code would be exactly the same in the bare nominal approach of section 2.1.1. In particular this unaltered induction suggests that the expressiveness with respect to the bare nominal approach is kept. It is efficient: no renaming, substitution, or shifting is involved. Polymorphic recursion is exploited: the call to `size t` in the `ƛ` (and `Let`) case is at some inner world.

```
size : ∀ {α} → Tm α → ℕ
size (V _)         = 1
size (t · u)       = 1 + size t + size u
size (ƛ _ t)       = 1 + size t
size (Let _ t u) = 1 + size t + size u
```

2.2.2 Building binders and names

In order to build terms, we need binders and names. Binders are introduced in our system using two simple primitive operations called `zero`^{B} and `suc`^{B}:

```
zeroᴮ : Binder
sucᴮ  : Binder → Binder
```

We introduce a cheap convenience function to turn any natural number into a binder:

```
_ᴮ : ℕ → Binder
zero    ᴮ = zeroᴮ
(suc n) ᴮ = sucᴮ (n ᴮ)
```

In effect, binders are natural numbers with a limited interface. Here we give only zero and successor but all of the operations on natural numbers could be exposed as well. One key limitation, however, is that no information must leak out of a binder. Hence no equality test over binders is provided.

This is clarified when building the logical relation in section 4.3.

To build new names we provide a function, called nameᴮ, to turn binders into names. For completeness we introduce the converse function binderᴺ. However, this function, without any properties on it, is of very little use.

```
nameᴮ   : ∀ {α} b → Name (b ◁ α)
binderᴺ : ∀ {α} → Name α → Binder
```

While nameᴮ can turn any binder into a name this function imposes the shape of the world that the produced name inhabits. Indeed nameᴮ b returns a name in any world whose last introduced binder is b. We will soon see that this is an important limitation and how to overcome it. For the moment, we have enough blocks to define a representation of the identity function!

```
idTm : ∀ {α} → Tm α
idTm = ƛ x (V (nameᴮ x))
  where x = 0 ᴮ
```

Our second example is a representation of the λ-term for false (λx.λx.x). We use the same binder twice on purpose to show a use of name shadowing. To type check the variable occurrence of x, the world x ◁ α is implicitly given to nameᴮ which returns a name in the world x ◁ x ◁ α. Then each of the two "ƛ x" takes away one "x ◁" to leave the final term in α.

```
falseTm : ∀ {α} → Tm α
falseTm = ƛ x (ƛ x (V (nameᴮ x)))
  where x = 0 ᴮ
```

One however faces an issue when building a representation of the λ-term for true (λx.λy.x). The naïve approach does not type-check:

```
-- this does not type-check
trueTm : ∀ {α} → Tm α
trueTm = ƛ x (ƛ y (V (nameᴮ x)))
  where x = 0 ᴮ
        y = 1 ᴮ
```

While nameᴮ x inhabits any world of the form x ◁ β, the context expects a name in the world y ◁ x ◁ α. We introduce means to move a name from one world to another in section 2.2.3.

Closed terms The above well-typed terms (`idTm` and `falseTm`) have no free variables. They are said to be closed. They both have the type: `∀ {α} → Tm α`. This world-polymorphic type reflects this closedness property. If any world fits, the empty world (noted `∅`) fits as well. So `idTm` also has the type `Tm ∅`. Conversely, because the empty world is included in all worlds, we will be able to move any term from the empty world to an arbitrary world. In short `Tm ∅` and `∀ {α} → Tm α` are *isomorphic* types.

To allow exploiting the fact that any term in the empty world is closed, we introduce `¬Name∅`, which witnesses that there is no name in the empty world:

```
∅        : World
¬Name∅ : ¬(Name ∅)
```

This last primitive operation is provided to secure the system against a buggy notion of closed terms but not only! In various situations it enables arguing that certain cases are impossible, like looking up a name in an empty environment.

2.2.3 World widening: Name weakening

It is often necessary to widen the world which a name inhabits. One example was the `trueTm` definition we tried earlier: to type-check it, we need to widen the world of the name `x`. Widening a world causes a loss of static information. For instance we have seen that a term in the empty world is closed: if one widens its world, it is no longer statically known to be closed. We call this operation weakening because it loses some static information. The name "weakening" is also vastly used for the same purpose on typing environments.

To account for the multiple ways in which we could widen a world we introduce a type to represent the inclusion between two worlds. The type `_⊆_` is for witnesses of world inclusion. If a world α is included in a world β then it is permitted to transport a name (and a term as well, as we shall see) from α to β. The primitive operation `coerceᴺ` takes an inclusion witness, a name and it returns the same name at a wider world. We also introduce an alias to `coerceᴺ` called `_⟨-because_-⟩` which is useful to keep the code separated from the typing/proof: the proof, which appears between the angle brackets, can be safely skipped by the reader.

```
_⊆_      : World → World → Set
coerceᴺ  : ∀ {α β} → α ⊆ β → (Name α → Name β)

infix 0 _⟨-because_-⟩
```

```
_⟨-because_-⟩ : ∀ {α β} → Name α → α ⊆ β → Name β
_⟨-because_-⟩ n pf = coerceᴺ pf n
 -- We can now write:
 --    x ⟨-because some proof -⟩
 -- Instead of the more noisy:
 --    coerceᴺ (some proof) x
```

World inclusion rules Rules for world inclusion are given in figure 2.1. World inclusion is reflexive and transitive (`⊆-refl` and `⊆-trans`). The empty world is a least element for world inclusion (`⊆-∅`). For any binder `b`, `b ◁ _` is covariant, meaning it preserves world inclusion (`⊆-◁`).

Finally the rule `⊆-#` states that a world α is included into `b ◁` α under the condition that `b` is not a member of α. This condition is necessary to ensure the soundness of our library. An interpretation of worlds as sets suggests that this condition might be unnecessary. This shows the limitations of this way of thinking. When we introduce logical relations in section 4.3, we explain that interpreting worlds as *relations* provides a richer viewpoint and explains why this condition is necessary. The following code snippet demonstrates that without this condition one can write wrong programs:

```
wrong : Binder → Binder → Bool
wrong x y = nameᴮ x ==ᴺ nameᴮ y ⟨-because wrongProof -⟩
    where postulate
            wrongProof : y ◁ ∅ ⊆ x ◁ y ◁ ∅
```

As this code snippet shows, in the absence of this condition it would be permitted to compare names at different worlds and so to compare binders and finally to make a distinction between two α-equivalent terms.

The fresh-for relation The last inclusion rule (`⊆-#`), uses a fresh-for relation called `_#_`. We introduce two rules to produce witnesses of this relation. The first rule tells that any binder is fresh for the empty world (`_#∅`). The second rule takes the “successor” of a fresh binder (`suc#`). This successor-based way of building new fresh-for witnesses is both simple and efficient. Indeed with these two rules, a binder is not only fresh-for a world but strictly greater than any element of it, allowing for a constant time `suc#` operation. These rules with their types can be found in figure 2.1 as well.

Emptiness of worlds The inclusion relation can express emptiness as well. A world α is empty if it is included in the empty world. We use this definition of emptiness as opposed to an intensional equality with the empty world for two reasons. First `∅ ⊆` α is always true, hence the converse is enough. Second we introduce in section 5.2 another operator on worlds called _+1 for which ∅ +1 is empty but does not reduce to ∅.

With a backward reasoning, combining coerce$^{\mathrm{N}}$ and ¬Name∅ turns any contradiction on names into an inclusion problem, reusing any automation done on inclusion proofs.

```
¬Name : ∀ {α} → α ⊆ ∅ → ¬(Name α)
¬Name α⊆∅ = ¬Name∅ ∘ coerceᴺ α⊆∅
```

Eliminating absurd names We introduce convenient eliminators for absurd names which instead of going to the empty type, goes to any type of our choice:

```
Name-elim : {A : Set} {α} → α ⊆ ∅ → Name α → A
Name-elim pf x with ¬Name pf x
...                  | ()

Name∅-elim : {A : Set} → Name ∅ → A
Name∅-elim = Name-elim ⊆-refl
```

Relational reasoning Sometimes one has to build complex inclusion witnesses. While inference would be of great effect here and is currently kept as future work, we propose a modest syntactic tool to build them, namely the ⊆-Reasoning module. It gives access to the transitivity rule ⊆-trans in a style which focuses more on the intermediate states of the reasoning rather than on the steps. The syntax is a list of worlds interspersed with inclusion witnesses, with two ■ around the last world. We soon present an example of its use with the function trueTm. The code for ⊆-Reasoning is given below for reference and can be safely skipped.

```
module ⊆-Reasoning where
  infix  2 finally
  infixr 2 _⊆⟨_⟩_

  _⊆⟨_⟩_ : ∀ x {y z} → x ⊆ y → y ⊆ z → x ⊆ z
  _ ⊆⟨ x⊆y ⟩ y⊆z = ⊆-trans x⊆y y⊆z

  finally : ∀ x y → x ⊆ y → x ⊆ y
  finally _ _ x⊆y = x⊆y

  syntax finally x y x⊆y = x ⊆⟨ x⊆y ⟩■ y ■
```

Building any term We can now finally build all nominal terms (up to α-equivalence). In section 3.2.8 we precisely show how to build all λ-terms by converting nominal λ-terms from section 2.1.1 to the type `Tm`. Moreover we show an encoding for nominal types in section 6.5. In the meantime we focus on the term `trueTm`. The two fresh-for rules above are the strict minimum for usability. More rules could be safely added but we focus on the minimal set of required rules here. The logical relation (section 4.2) serves as a criterion to judge what can or cannot be added. Having only these two rules imposes a particular style. A variable occurrence crosses only binders with a lower value to reach its own binder. This does not hinder much the expressiveness but requires to freshen some binders if one cannot prove they are fresh-for a given world. As stated previously removing this minor restriction would require adding more rules to the fresh-for and inclusion relations. One can finally build the term `trueTm`:

```
trueTm : ∀ {α} → Tm α
trueTm {α} = ƛ x (ƛ y xᵀ)
  where open ⊆-Reasoning
        x    = 0 ᴮ
        y    = 1 ᴮ
        xᵀ   = V (nameᴮ x ⟨-because pf -⟩)
        pf   = x ◁ ∅
             ⊆⟨ ⊆-# (suc# (x #∅)) ⟩
               y ◁ x ◁ ∅
             ⊆⟨ ⊆-◁ y (⊆-◁ x ⊆-∅) ⟩∎
               y ◁ x ◁ α
             ∎
```

The proof required to "move `x` in the correct world" is involved in comparison with the simplicity of the example. By choosing the empty world instead of world polymorphism we could cut in half the proof. To build larger examples, we build specialized building functions which requires only to specify the distance to the binding site [Pouillard, 2011a].

2.2.4 Comparing and refining names

While two binders cannot be compared, our interface allows comparing two names that inhabit a common world. This may seem contradictory, since one can turn binders into names. In fact, binder comparison cannot be implemented in terms of name comparison because two arbitrary binders can be turned only into names in distinct worlds.

```
_==ᴺ_ : ∀ {α} (x y : Name α) → Bool
```

While world inclusion gives a means to weaken the type of a name, we also need a means to strengthen the type of a name by refining its world. Assume a name `x` is known to be in the scope of a binder `b`: `x` is in the world `b ◁ α`. The function `exportᴺ` tests whether `x` is equal to `b`; if so, `x` is refined to the world `b ◁ ∅` (which only `b` inhabits), otherwise it is refined to α. In short given a name of type `Name (b ◁ α)`, the function `exportᴺ` returns the same name with a refined type telling whether the name stands on the `b` side or on the α side.

```
exportᴺ : ∀ {b α} → Name (b ◁ α) → Name (b ◁ ∅) ⊎ Name α
exportᴺ = maybe inj₂ (inj₁ (nameᴮ _)) ∘′ exportᴺ?
```

Actually the primitive operation in the interface is the function `exportᴺ?` which returns a result of type `Maybe (Name α)`. The function `exportᴺ` is then built on top of `exportᴺ?`.

```
-- A →? B is the type of partial functions from A to B
A →? B = A → Maybe B

exportᴺ? : ∀ {b α} → Name (b ◁ α) →? Name α
```

On top of `exportᴺ?` we also build a convenient eliminator for names which is simply the elimination of the result of `exportᴺ?`.

```
exportWith : ∀ {b α A} → A → (Name α → A) → Name (b ◁ α) → A
exportWith v f = maybe f v ∘′ exportᴺ?
```

Chapter 3

Programming on top of NomPa

3.1 Various examples

3.1.1 Example: computing free variables

We now have enough tools to present a more interesting example, namely a function that constructs a list of the free variables of a term. At variables and applications, the code is straightforward. At a name abstraction, one easily collects the free variables of the body via a recursive call. However, this yields a list of names that inhabit the inner world of the abstraction—a value of type `List (Name (b ◁ α))`. This list cannot be returned, and this is fortunate, since doing so would let the bound name leak out of its scope! We define an auxiliary function, `rm`, which removes all occurrences of a binder `b` in a list of names and at the same time performs type refinement in the style of `exportᴺ`/`exportᴺ?`.

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs) with exportᴺ x -- b is implicit
... {- bound: x≡b -} | inj₁ _     = rm b xs
... {- free:  x≢b -} | inj₂ x′    = x′  :: rm b xs

fv : ∀ {α} → Tm α → List (Name α)
fv (V x)          = [ x ]
fv (fct · arg)    = fv fct ++ fv arg
fv (ƛ b t)        = rm b (fv t)
fv (Let b t u)    = fv t ++ rm b (fv u)
```

The function `rm` applies `exportᴺ` to every name `x` in the list and builds a list of only those x's that successfully export to the world α. It exhibits a typical way of using export$^{\mathtt{N}}$ to perform a name comparison together with a type refinement. This idiom is recurrent in the programs that we have written.

3.1.2 Example: working with environments

Here is another example, where we introduce the use of an environment.

```
occurs : ∀ {α} → Name α → Tm α → Bool
occurs x₀ = occ (λ y → x₀ ==ᴺ y)
  where
   OccEnv  : World → Set
   OccEnv  α = Name α → Bool
   extend  : ∀ {α b} → OccEnv α → OccEnv (b ◁ α)
   extend  = exportWith false

   occ : ∀ {α} → OccEnv α → Tm α → Bool
   occ Γ (V x)          = Γ x
   occ Γ (t · u)        = occ Γ t ∨ occ Γ u
   occ Γ (ƛ _ t)        = occ (extend Γ) t
   occ Γ (Let _ t u)    = occ Γ t ∨ occ (extend Γ) u
```

The function occurs tests whether the name x_0 occurs free in a term. An environment Γ is carried down, augmented when a binder is crossed, and looked up at variable occurrences. Here, this environment is represented as a function of type Name α → Bool. Although this is a simple and elegant representation, others exist. For instance, we could represent the environment as a list of binders: the code for this variant is online [Pouillard, 2011a]; see also below.

The definition of extend states that, in order to look up x in the environment extend Γ, one must first compare x and b and, only if they differ, one must look up x in Γ. If x and b are equal, then this occurrence of x is not free, so occ Γ (V x) must return false. This is concisely implemented by using the function exportWith built on top of export$^{\mathtt{N}}$ (Again thanks to the precise typing using worlds, b is passed as an implicit argument to the function extend and exportWith).

We claim that this code is standard and uncluttered. There is no hidden cost: no renaming is involved. Admittedly, neither functions nor lists are the most efficient representation of environments. It would be nice to be able to implement environments using, say, balanced binary search trees, while preserving well-typedness. We leave this issue to future study.

The type system forces us to use names in a sound way. For instance, in the definition of occ, forgetting to extend the environment when crossing a binder (that is, writing Γ instead of extend Γ) would cause a type error. In the definition of extend, attempting to check whether x occurs in Γ without first comparing x and b would cause a type error. Remember that in this nominal implementation scheme it is permitted for newer bindings to shadow earlier ones; our type discipline guarantees that the code works also in that case.

As suggested previously, one may wish to represent environments as an explicit data structure (an association list keyed by binders) rather than as an opaque object (a lookup function).

```
data DataEnv (A : Set) : (α β : World) → Set where
  ϵ : ∀ {β} → DataEnv A β β
  _,_↦_ : ∀ {α β} (Γ : DataEnv A α β) b (x : A)
          → DataEnv A (b ◁ α) β
```

```
lookup : ∀ {A α β} → DataEnv A α β → Name α → Name β ⊎ A
lookup ϵ              = inj₁
lookup (Γ , _ ↦ v) = exportWith (inj₂ v) (lookup Γ)
```

The type DataEnv A α β is the type of an environment, or environment fragment, where α is called the inner world, where β is called the outer world, and every name in the environment is associated with a datum of type A. The inner world is the world where all the binders of the environment are in scope. The outer world is the world where none of the binders are in scope. The expression lookup Γ x looks up the name x in the environment Γ. The name x must make sense in the scope of Γ, that is, x must inhabit the world α. If x is found among the bindings, then the information associated with x can be returned. If x is not found among the bindings, then x is returned, with a more precise type: indeed, since x is not among the names introduced by Γ, it must make sense outside Γ, that is, in the world β.

We illustrate the use of DataEnv with an alternative definition of the function fv (here the payload type parameter A is instantiated with the unit type ⊤). This variant avoids the need to take the bound atoms off the list by not inserting them in the first place. At variables, we use lookup to check whether the name is free or bound. If the name is free, we return it as a singleton list (using [_]). If it is bound, we ignore it and return an empty list. At every other node, we simply carry out a recursive traversal. Whenever a name abstraction is entered, the current environment Γ is extended with the bound name b.

```
fv' : ∀ {α β} → DataEnv ⊤ α β → Tm α → List (Name β)
fv' Γ (V x)         = [ [_] , const [] ]′ (lookup Γ x)
fv' Γ (t · u)       = fv' Γ t ++ fv' Γ u
fv' Γ (ƛ b t)       = fv' (Γ , b ↦ _) t
fv' Γ (Let b t u) = fv' Γ t ++ fv' (Γ , b ↦ _) u

fv : ∀ {α} → Tm α → List (Name α)
fv = fv' ε
```

3.1.3 Example: Term comparison

Our next example focuses on comparison of terms. We first define `|Cmp|`, where `|Cmp| F` is the type of functions comparing `F`-structures. In our case the index type `Ix` is the type of worlds: `World`.

```
|Cmp| : ∀ {Ix} (F : Ix → Set) (i j : Ix) → Set
|Cmp| F i j = F i → F j → Bool
```

To compare two terms we carry down an environment, telling how to compare two names. At name occurrences we ask our environment, at application nodes we carry down our environment, and we need to extend the environment at name-abstractions. We extend the environment with the function `extendNameCmp` which receives a name comparator `f` for worlds α_1 and α_2, a name x_1 in $b_1 \lhd \alpha_1$, and a name x_2 in $b_2 \lhd \alpha_2$. Then, the function `extendNameCmp` exports both names x_1 and x_2 simultaneously: when the two exports succeed we can use the comparator `f`, when both fail we learn that x_1 is b_1 and that x_2 is b_2 we hence return `true`, when one succeeds and the other fails we return `false`:

```
extendNameCmp : ∀ {α₁ α₂ b₁ b₂} → |Cmp| Name α₁ α₂
                                → |Cmp| Name (b₁ ◃ α₁) (b₂ ◃ α₂)
extendNameCmp f x₁ x₂
  with exportᴺ? x₁    | exportᴺ? x₂
... | just x₁′        | just x₂′    = f x₁′ x₂′
... | nothing         | nothing     = true
... | _               | _           = false

cmpTm : ∀ {α₁ α₂} (Γ : |Cmp| Name α₁ α₂) → Tm α₁ → Tm α₂ → Bool
cmpTm Γ (V x1)        (V x2)        = Γ x1 x2
cmpTm Γ (t1 · u1)     (t2 · u2)     = cmpTm Γ t1 t2 ∧ cmpTm Γ u1 u2
cmpTm Γ (ƛ _ t1)      (ƛ _ t2)      = cmpTm (extendNameCmp Γ) t1 t2
cmpTm Γ (Let b1 t1 u1) (Let b2 t2 u2)
  = cmpTm Γ t1 t2 ∧ cmpTm (extendNameCmp Γ) u1 u2
cmpTm _ _             _             = false
```

Notice that the code above does not need to compare neither names nor binders from the first term with those from the second term. Two bound names are said to be equal when they are bound at the same time. Bound names are said to be compared positionally. This also explains why we do not need a function to compare binders. Notice also that the function `cmpTm` has to accept two terms in different worlds to carry the recursion successfully. In the end we can give the function `_==`^{N}`_` as an initial environment yielding an homogeneous comparison on terms, which coincides with α-equivalence:

```
_==^Tm_ : ∀ {α} → Tm α → Tm α → Bool
_==^Tm_ = cmpTm _==^N_
```

In this final setting bound names are compared positionally and free names are compared for equality.

3.2 Kits and Traversals

We have seen that working with worlds requires explicitly moving names from world to world using operations like `coerce`^{N} and `export`^{N}`?`. These operations quickly become necessary at the level of user-defined types like our type `Tm`. More generally any operation on names would benefit from being lifted to user-defined types.

From our experimentations with this library we outline two points. First, in most of the operations on terms we present, only the binding structure is relevant. The code for the traversal can hence be written only once. Second, the parts specific to each operation can be made reusable to work not only with the generic traversal but with custom traversals as well.

To share code and separate concerns we introduce some infrastructure. While abstract this infrastructure is then concretely applied to our example type `Tm`.

3.2.1 Traversal Kits

We first introduce the notion of *traversal kits*, which encompass: a type of environment `Env` α β to be carried down in the traversal going from the world α to β; a function `trName` to apply the environment at names; a function `trBinder` for binders; and finally a function `extEnv` to push an environment under a binder. For more flexibility the result type Res of trName is a parameter as well, we shall see its use later on in this section. We build traversal kits for: coercing (applying a world inclusion witness), exporting, applying general effectful functions over names, freshening, and combinations of these.

```
record TrKit (Env : (α β : World) → Set)
             (Res : World → Set) : Set where
  constructor mk
  field
    trName   : ∀ {α β} → Env α β → Name α → Res β
    trBinder : ∀ {α β} → Env α β → Binder → Binder
    extEnv   : ∀ {α β} b (Δ : Env α β)
                     → Env (b ◁ α) (trBinder Δ b ◁ β)
```

3.2.2 Coercing kit

Our first kit, called coerceKit, is simple and to the point. The "environment" type is _⊆_. The result type is simply Name. The operation to call at names is coerce^N. The action at binders is the identity, meaning that this kit does not perform any renaming or freshening. Finally the inclusion rule ⊆-◁ fits the job of extEnv.

```
coerceKit : TrKit _⊆_ Name
coerceKit = mk coerceᴺ (const id) ⊆-◁
```

To illustrate the usage of coerceKit, we use it to lift coerce^N from names to terms. In this situation the reuse is negligible, but this code serves a pedagogical purpose.

```
module CoerceTmWithCoerceKit where
  open TrKit coerceKit

  coeTm : ∀ {α β} → α ⊆ β → Tm α → Tm β
  coeTm Δ (V x)       = V (trName Δ x)
  coeTm Δ (t · u)     = coeTm Δ t · coeTm Δ u
  coeTm Δ (ƛ b t)     = ƛ (trBinder Δ b) (coeTm (extEnv b Δ) t)
  coeTm Δ (Let b t u) = Let (trBinder Δ b) (coeTm Δ t)
                            (coeTm (extEnv b Δ) u)
```

The function coeTm takes an inclusion witness and an input term. The inclusion witness is carried down the traversal, used at names with the function trName (equal to coerce^N in the coerceKit) and extended at abstractions with the function extEnv (equal to ⊆- in the coerceKit). The function trBinder is used to choose the binders of the output term which are the ones from the input in the coerceKit. This traversal is actually independent of the kit we use.

Our next kit requires the concept of name supplies. A name supply for the world α is just a pair of a binder called seed and a proof that seed is fresh for α:

```
record Supply α : Set where
  constructor _,_
  field
    seedᴮ : Binder
    seed# : seedᴮ # α
```

We also introduce two helpers called zero^s and suc^s. zero^s is the initial name supply and the function suc^s increments both the seed and the fresh-for proof:

```
zeroˢ : Supply ∅
zeroˢ = 0 ᴮ , 0 ᴮ #∅

sucˢ : ∀ {α} → (s : Supply α) → Supply (Supply.seedᵂ s)
sucˢ (seedᴮ , seed#) = sucᴮ seedᴮ , suc# seed#
```

3.2.3 Renaming kits

Our second kit, the "renaming kit", is more involved and shows how to apply any function over names on the binding structure. To avoid captures we have to freshen the binders in the result and the system indeed imposes that we do so. We first define the record type ix[SubstEnv α β. This record type holds: a function trName to apply at names and a name supply for β:

```
record SubstEnv (Res : World → Set) α β : Set where
  constructor _,_
  field
    trName : Name α → Res β
    supply : Supply β

  open Supply supply public

  -- each binder is translated to a fresh binder
  trBinder : Binder → Binder
  trBinder _ = seedᴮ
```

The so-called `renameKit` is then defined by providing `trName`, `trBinder` and `extEnv`. This last one is the most involved part of the kit and depicted in figure 3.1. The function `trName` is lifted using `exportWith`. The lifted function `trName` takes a name, if this name is bound then the `seed`^{B} is returned, otherwise this name (exported) is given to `trName` to produce a name which is then imported back using `coerce`^{N}. This call to `coerce`^{N} is valid only because `seed`^{B} is known to be fresh for the output world. Here again `SubstEnv` is parameterized by the result type of `trName` for greater flexibility:

```
RenameEnv : (α β : World) → Set
RenameEnv = SubstEnv Name

renameKit : TrKit RenameEnv Name
renameKit = mk trName trBinder extEnv
  where
    open SubstEnv
    extEnv : ∀ {α β} b (Δ : RenameEnv α β)
                          → RenameEnv (b ◁ α) (_ ◁ β)
    extEnv _ (trName   , (seedᴮ      , seed#β))
           = (trName′   , (sucˢ (seedᴮ , seed#β)))
     where
      trName′ = exportWith
                    (nameᴮ seedᴮ) -- bound
                    (coerceᴺ (⊆-# seed#β) ∘ trName) -- free
```

Our renaming kit works for any *total* function over names. However, to lift the function `export`^{N}`?` from names to terms, we need to deal with partial functions as well. We can actually build another renaming kit parameterized over a notion of effectful computation, namely an applicative functor. An applicative functor [McBride and Paterson, 2008] is halfway between a functor and a monad. Like a monad, an applicative functor has

$$\begin{tikzcd}
\mathtt{trName}\ : & \mathtt{Name}\ \alpha \arrow[r] & \mathtt{Name}\ \beta \\
\mathtt{trName'}\ : & \mathtt{Name}\ (\mathtt{b} \triangleleft \alpha) \arrow[r] \arrow[u, "\mathtt{exportWith}"] & \mathtt{Name}\ (\_ \triangleleft \beta) \arrow[u, "\mathtt{coerce}^{\mathtt{N}}"]
\end{tikzcd}$$

Figure 3.1: Lifiting `trName`

a unit called `pure`. The function `pure` allows to embed any pure value as a potentially effectful one. The second operation, called _⊛_, is an effectful application, taking an effectful function and argument and resulting in an effectful result. Here is how we can use applicative functors to apply an effectful function to a list:

```
module MapA {E} (E-app : Applicative E) where
  open Applicative E-app

  mapA : {A B : Set} → (A → E B) → List A → E (List B)
  mapA _ []        = pure []
  mapA f (x :: xs) = pure _::_ ⊛ f x ⊛ mapA f xs
```

To build a more general renaming kit, we reuse the type `SubstEnv` but make use of the flexibility on the result type to plug in the effect type `E` (the applicative functor). The code for `renameAKit` is similar to `renameKit`, so we omit it here and provide only its type:

```
RenameAEnv : (E : Set → Set) (α β : World) → Set
RenameAEnv E = SubstEnv (E ∘ Name)

renameAKit : ∀ {E} → Applicative E →
               TrKit (RenameAEnv E) (E ∘ Name)
renameAKit = {! code similar to renameKit omitted !}
```

3.2.4 Substitution kit

We reuse our `SubstEnv` once more to generalize in another direction the renaming kit to the so-called `substKit` which helps build substitution functions. Substitution functions are built for a structure `F` (like our type `Tm`), provided there is a way of injecting names to `F`-structures and of coercing an F-structure from one world to another. Here again we choose another result type for SubstEnv, namely F .

```
-- Index-respecting functions
F |→| G = ∀ {i} → F i → G i

-- The type for ‘coerce’ on an F-structure
Coerce F = ∀ {α β} → α ⊆ β → F α → F β

substKit : ∀ {F}
             (V       : Name |→| F)
             (coerceF : Coerce F)
           → TrKit (SubstEnv F) F
substKit = {! code similar to renameKit omitted !}
```

3.2.5 Other kits and combinators

We can also build others kits and combinators [Pouillard, 2011a]. For instance ∘-Kit can compose two kits by pairing the two environments and composing their operations. Another combinator, starKit, takes a kit k on environments of type Env and builds a kit working on Star Env (the reflexive and transitive closure operator in Agda). Finally with mapKit one can pre-compose and post-compose a function with a kit to get a new kit. This last one is given below:

```
mapKit : ∀ {Env F G} (f : Name |→| Name) (g : F |→| G)
         → TrKit Env F → TrKit Env G
mapKit f g kit = mk (λ Δ → g ∘ trName Δ ∘ f) trBinder extEnv
  where open TrKit kit
```

3.2.6 Reusable traversal

Putting all these kits to work is a matter of writing a traversal function working for any kit and any effect. It is essentially a map function where the free names are transformed by a user-supplied function. More precisely the function trTm traverses the term, carrying an environment. It uses trName at variable occurrences without putting back the constructor V. In every other case, it just rebuilds the same structure using the corresponding constructor and the operations of our applicative functor. To carry the environment under bindings extEnv is used:

```
module TraverseTm {E}   (E-app : Applicative E)
                  {Env} (trKit : TrKit Env (E ∘ Tm)) where

  open Applicative E-app
  open TrKit trKit

  trTm : ∀ {α β} → Env α β → Tm α → E (Tm β)
  trTm Δ (V x)         = trName Δ x
  trTm Δ (t · u)       = pure _·_ ⊛ trTm Δ t ⊛ trTm Δ u
  trTm Δ (ƛ b t)       = pure (ƛ _) ⊛ trTm (extEnv b Δ) t
  trTm Δ (Let b t u)   = pure (Let _) ⊛ trTm Δ t
                                      ⊛ trTm (extEnv b Δ) u
```

3.2.7 Reusing the traversal

We can now collect the fruit of our work, by combining the traversal and a few kits. For the sake of simplicity we directly do so on the type `Tm`, but in our development [Pouillard, 2011a] we further abstract over `Tm` (as `F`) and `trTm` by defining a sequence of parameterized modules.

```
open TraverseTm

-- Like ‘trTm’ but restores the constructor ‘V’.
-- The result type of ‘trKit’ is refined.
trTm′  : ∀ {E}   (E-app : Applicative E)
           {Env} (trKit : TrKit Env (E ∘ Name))
           {α β} → Env α β → Tm α → E (Tm β)
trTm′ E-app trKit
  = trTm E-app (mapKit id (Applicative._<$>_ E-app V) trKit)
```

While `coerce`$^{\mathtt{N}}$ can be lifted to terms using a renaming function, this would impose freshening. Using `trTm′` with the `coerceKit` yields a more efficient implementation.

```
-- The identity on types froms an applicative functor
id-app : Applicative id
id-app = {! definition omitted !}

coerceTm : ∀ {α β} → α ⊆ β → Tm α → Tm β
coerceTm = trTm′ id-app coerceKit
```

Building the renaming function amounts to use one of our renaming kits and to pick one applicative functor:

```
renameTmA : ∀ {E} → Applicative E →
            ∀ {α β} → Supply β → (Name α → E (Name β))
                               → (Tm α   → E (Tm β))
renameTmA E-app s f = trTm′ E-app (renameAKit E-app) (f , s)

renameTm : ∀ {α β} → Supply β → (Name α → Name β)
                              → (Tm α   → Tm β)
renameTm s f = trTm′ id-app renameKit (f , s)

renameTm? : ∀ {α β} → Supply β → (Name α →? Name β)
                               → (Tm α   →? Tm β)
renameTm? = renameTmA Maybe.applicative
```

Exporting is built by lifting the function `exportᴺ?` using the function `renameTm?`:

```
exportTm? : ∀ {b α} → Supply α → Tm (b ◁ α) →? Tm α
exportTm? s = renameTm? s exportᴺ?
```

Another special case of `renameTm?` is the so-called `closeTm?`. This function takes a term in any world and checks if the term is closed. If so, the same term is returned in the empty world. Otherwise the function fails by returning `nothing`:

```
closeTm? : ∀ {α} → Tm α →? Tm ∅
closeTm? = renameTm? (0 ᴮ , (0 ᴮ)#∅) (const nothing)
```

Building capture avoiding substitution amounts to use the substitution kit with the constructor `V` and the function `coerceTm` as arguments:

```
substTm : ∀ {α β} → Supply β → (Name α → Tm β) → Tm α → Tm β
substTm (s , s#) f = trTm id-app (substKit V coerceTm) (f , s, s#)
```

To illustrate the use of `substTm`, here is a simple function `β-red` which performs a β-reduction when a β-redex appears at the root of the term. The function `exportWith a V` associate `a` to `b` and `x` to `V x` when `x ≢ b`:

```
β-red : ∀ {α} → Supply α → Tm α → Tm α
β-red s (ƛ b f · a) = substTm s (exportWith a V) f
β-red _ t           = t
```

3.2.8 Building any λ-term

A way to illustrate that every λ-term can be encoded using our type Tm is to define a conversion function from another type for λ-terms to the type Tm. We do so by choosing the type Tmᴬ from section 2.1.1 as the source language.

The process is very close to the combination of a specific renaming kit and traversal function. The kit is specific since the source names are of type Atom and not Name. The traversal function is specific since the source and target types are not the same and we picked the identity functor for simplicity.

First, we introduce the environment type which holds a mapping from free atoms to free names and a name supply:

```
module Conv-Tmᴬ→Tm where
  record Env α : Set where
    constructor _,_
    field
      trAtom : Atom → Name α
      supply : Supply α
    open Supply supply public
  open Env
```

We then define how an environment is extended. It works similarly to the one for the renaming kit:

```
  extEnv : ∀ {α} → Atom → (Δ : Env α) → Env (seedᴮ Δ ◁ α)
  extEnv bᴬ Δ = trᴺ , sucˢ (supply Δ)
    where trᴺ = λ xᴬ → if bᴬ ==ᴬ xᴬ
                          then nameᴮ (seedᴮ Δ)
                          else coerceᴺ (⊆-# (seed# Δ)) (trAtom Δ xᴬ)
```

The following function conv is then straightforward, calling trAtom on atoms at the variable case and using extEnv when crossing a binding. This function conv is simpler than the traversal function seen previously since we avoided the use of effectful computations, however the applicative version works as well. The function conv can then be used if we can associate all atoms to names and have the corresponding name supply.

```
  conv : ∀ {α} → Env α → Tmᴬ → Tm α
  conv Δ (V x)         = V (trAtom Δ x)
  conv Δ (ƛ b t)       = ƛ _ (conv (extEnv b Δ) t)
  conv Δ (t · u)       = conv Δ t · conv Δ u
  conv Δ (Let b t u) = Let _ (conv Δ t) (conv (extEnv b Δ) u)
```

To convert closed terms we need an empty environment. However the type Tm^{A} does not ensure closedness. Hence the conversion is partial. To build this partial conversion we define it in two steps. The first step uses the function conv with an empty environment. This empty environment maps any atom to the name 0 ^{N}, the name supply hence starts at 1 ^{N}. As the second step we use the function closeTm? which strengthen the type of the input term if it is closed and fails otherwise.

```
emptyEnv : Env (0 ᴮ ◁ ∅)
emptyEnv = const (0 ᴺ) , sucˢ zeroˢ

conv∅? : Tmᴬ →? Tm ∅
conv∅? = closeTm? ∘ conv emptyEnv
```

3.3 Towards elaborate uses of worlds

3.3.1 Data type of “one hole contexts”

The type Tm is just one basic example of an algebraic data type that involves names and binders. As a more challenging example, consider a type C of one-hole contexts associated with Tm. The type C is indexed with two worlds, which respectively play the role of an outer world and an inner world. The idea is, plugging a term of type Tm β into the hole of a context of type C α β produces a term of type Tm α. The definition of the type C is as follows:

```
data C α : World → Set where
  Hole  : C α α
  _·₁_  : ∀ {β}   → C α β → Tm α → C α β
  _·₂_  : ∀ {β}   → Tm α   → C α β → C α β
  ƛ     : ∀ {β} b → C (b ◁ α) β → C α β
  Let₁  : ∀ {β} b → C α β
                  → Tm (b ◁ α) → C α β
  Let₂  : ∀ {β} b → Tm α
                  → C (α ◁ α) β → C α β
```

Contexts bind names: the hole can appear under one or several binders. This is why, in general, a context has distinct outer and inner worlds. A context contains a list of binders that “connects” the outer and inner worlds: these binders are carried by the constructors ƛ.

A context and a term can be paired to produce a term-in-context:

```
CTm : World → Set
CTm α = ∃[ β ](C α β × Tm β)
```

It is straightforward to define a function plug from CTm α to Tm α, which accepts a pair of a context and a term and plugs the latter into the former. Conversely, one can define a family of focusing functions ($\forall\{\alpha\} \rightarrow$ Tm $\alpha \rightarrow$ CTm α), which split a term into a pair of a context and a term. There are several such functions, according to where one wishes to focus.

3.3.2 Patterns a la ML

As another instance of this idea, if one wished to extend our object language with ML-style patterns, one would index the type Pat of patterns with an outer world and an inner world, and one would use elaborate abstractions of the form ∃[β](Pat α β × Tm β).

3.3.3 Term and types: System F

We use Church's System F as an object language to illustrate how representations of object-level types can be indexed by worlds as well. Object-level terms are now indexed by two worlds: one for type names and one for term names.

```
module SysF where
  infixr 5 _⇒_
  data Ty α : Set where
    V     : (x : Name α)           → Ty α
    _⇒_   : (σ τ : Ty α)           → Ty α
    ‘∀‘   : ∀ b (τ : Ty (b ◁ α))   → Ty α

  data Tm α γ : Set where
    V      : ∀   (x : Name α)              → Tm α γ
    _·_    : ∀   (t u : Tm α γ)            → Tm α γ
    ƛ      : ∀ b (τ : Ty γ)
                 (t : Tm (b ◁ α) γ)        → Tm α γ
    _·τ_   : ∀   (t : Tm α γ) (τ : Ty γ)   → Tm α γ
    Λ      : ∀ b (t : Tm α (b ◁ γ))        → Tm α γ
```

3.4 Advanced example: normalization by evaluation

As an advanced example, we show how to express a normalization by evaluation algorithm in our system. This algorithm has been previously used as a benchmark by several researchers [Shinwell et al., 2003, Pitts, 2006, Licata and Harper, 2009, Cave and Pientka, 2012]. The challenge lies in the way the algorithm mixes computational functions, name abstractions, and fresh name generation.

The object language of interest is again the pure λ-calculus. The algorithm exploits two different representations of object-level terms, which are respectively known as *syntactic* and *semantic* representations. Because these representations differ only in their treatment of name abstractions, they can be given a common definition, which is parameterized over the representation of abstractions:

```
module M (Abs : (World → Set) → World → Set) where
  data T α : Set where
    V     : Name α       → T α
    ƛ     : Abs T α      → T α
    _·_   : T α → T α → T α
```

The parameter `Abs` has kind `(World → Set) → (World → Set)`: it is an indexed-type transformer.

In order to obtain the syntactic representation, we instantiate `Abs` with the nominal abstractions that we have used everywhere so far: an abstraction is a package of a binder and of a term that inhabits the inner world. This yields the type `Term` of syntactic terms.

```
SynAbsᴺ : (World → Set) → World → Set
SynAbsᴺ F α = ∃[ b ](F (b ◃ α))

open M SynAbsᴺ renaming (T to Term)
```

In order to obtain the semantic representation, we instantiate `Abs` with a different notion of abstraction, in the style of higher-order abstract syntax: an abstraction is a computational function, which substitutes a term for the bound name of the abstraction. This yields the type `Sem` of semantic terms. The type `Sem` is not an inductive data type; fortunately, with the `--no-positivity-check` flag, Agda accepts this type definition, at the cost of breaking strong normalization (to minimize the risks we isolated this kind of example to separate modules where this flag is activated).

```
SemAbs : (World → Set) → World → Set
SemAbs F α = ∀ {β} → α ⊆ β → F β → F β

open M SemAbs renaming (T to Sem)
```

It is important to note that our semantic name abstractions involve bounded polymorphism in a world: we define `SemAbs F α` as `∀{β} → α ⊆ β → F β → F β`, as opposed to the more naïve `F α → F α`. This provides a more accurate and more flexible description of the behavior of substitution. Indeed, when instantiating an abstraction `t` with some term `u`, it makes perfect sense for `u` to inhabit a larger world than `t`, that is, for `u` to refer to certain names that are fresh for `t`. The result of the substitution then inhabits the same world as `u`: that is, it potentially refers to these fresh names, in addition to all of the names that occurred free in the abstraction `t`.

The types `SemAbs` (and `Sem`), are covariant with respect to the parameter α, which would not be the case with the naïve definition. In other words, it is possible to define a coerce operation for semantic terms:

```
coerceSem : ∀ {α β} → α ⊆ β → Sem α → Sem β
coerceSem pf (V a)     = V (coerceᴺ pf a)
coerceSem pf (ƛ f)     = ƛ (λ pf′ v → f (⊆-trans pf pf′) v)
coerceSem pf (t · u)   = coerceSem pf t · coerceSem pf u
```

At a semantic abstraction, no recursive call is performed, because the body of the abstraction is opaque: it is a computational function. Instead, we exploit the transitivity of world inclusion and build a new semantic abstraction that inhabits the desired world.

The normalization by evaluation algorithm makes use of environments. Here, environments are functions from names to semantic terms. The function `_,_↦_` shows how to extend such an environment:

```
EvalEnv : (α β : World) → Set
EvalEnv α β = Name α → Sem β
-- α is the inner world
-- β is the outer world

_,_↦_ : ∀ {α β} (Γ : EvalEnv α β) b → Sem β → EvalEnv (b ◁ α) β
_,_↦_ Γ b v = exportWith v Γ
-- meaning: b ↦ v
--          x ↦ Γ x
```

To a name, of type `Name α` an environment, of type `EvalEnv α β` associates a semantic term that lies outside the scope of the environment that is, a semantic term of type `Sem β`. The type `EvalEnv α β` is, again, covariant in its second world, as witnessed by the following coercion function:

```
coerceEnv : ∀{α β₁ β₂}→ β₁ ⊆ β₂ → EvalEnv α β₁ → EvalEnv α β₂
coerceEnv pf Γ = coerceSem pf ∘ Γ
```

The first part of the normalization by evaluation algorithm is a function `eval` that evaluates a syntactic term within an environment to produce a semantic term. When evaluating a λ-abstraction, we build a semantic abstraction, which encapsulates a recursive call to `eval`. The bounded polymorphism required by the definition of semantic abstractions forces us to coerce the environment Γ via `coerceEnv`.

```
eval : ∀ {α β} → EvalEnv α β → Term α → Sem β
eval Γ (ƛ (a , t))
  = ƛ (λ pf v → eval (coerceEnv pf Γ , a ↦ v) t)
eval Γ (V x)        = Γ x
eval Γ (t · u)      = app (eval Γ t) (eval Γ u) where
  app : ∀ {α} → Sem α → Sem α → Sem α
  app (ƛ f)  v = f ⊆-refl v
  app n      v = n · v
```

The second part of the algorithm reifies a semantic term back into a syntactic term. When reifying a semantic abstraction, we build a syntactic abstraction. This requires generating a fresh name, and leads us to parameterizing `reify` with a supply of fresh names.

```
reify : ∀ {α} → Supply α → Sem α → Term α
reify s          (V a)    = V a
reify s          (n · v) = reify s n · reify s v
reify (sᴮ , s#) (ƛ f)   =
    ƛ (sᴮ , reify (sucˢ (sᴮ , s#)) (f (⊆-# s#) (V (nameᴮ sᴮ))))
```

The constructor `V` has type `Name` α `→ Sem` α hence is a valid initial environment of type `EvalEnv` α α. Evaluation under this initial environment, followed with reification, yields a normalization algorithm. This algorithm works with open terms: its argument, as well as its result, are terms in an arbitrary world α. However this function needs a name supply for the world α. The name supply can be incrementally constructed with `zero`$^{\mathsf{s}}$ and `suc`$^{\mathsf{s}}$. In particular we can normalize closed terms by giving `zero`$^{\mathsf{s}}$:

```
nf : ∀ {α} → Supply α → Term α → Term α
nf supply = reify supply ∘ eval V
```

Here is an example of the normalization of a term:

```
idᵀ : Term ∅
idᵀ = ƛ (0 ᴮ , V (0 ᴺ))

test-nf : nf zeroˢ ((idᵀ · (idᵀ · idᵀ)) · idᵀ) ≡ idᵀ
tset-nf = refl
```

Chapter 4

Behind the scene of the NomPa library

4.1 Implementation of NomPa

The implementation of our interface is not surprising. Most of the code is types or proofs. Notice that this section details the internals of our library and that none of these definitions are meant to be known or used by the library client.

Worlds come first and are represented by lists of booleans. Such a list means that n is in the world if and only if the n^{th} value of the list is `true`. Formally the meaning of a world is defined by the membership predicate below. The choice of a list of booleans to represent a world was guided by two facts. First, the operations are structural: this eases the type-level computation and the proof goes through in particular in combination with de Bruijn indices as we details in section 5.2. Second, the elements are kept in order and (modulo trailing occurrences of `false`) two equivalent sets are represented the same way.

```
World : Set
World = List Bool

∅ : World
∅ = []

_∈_ : ℕ → World → Set
_       ∈ []              = ⊥
zero    ∈ (false :: _)    = ⊥
zero    ∈ (true  :: _)    = ⊤
suc n   ∈ (_     :: xs)   = n ∈ xs
```

Worlds are meant to be computationally irrelevant. Meaning that none of the functions of this implementation make use of the worlds to compute the results. Hence, programs have an equivalent dynamic semantics when worlds are erased. While worlds are indeed used in membership, inclusion, and fresh-for proofs, these proofs are meant to be erased as well. Finally we want this erasing for three reasons: first for performances in the context of an optimizing compiler, second to have a separation between the "type system" and the running program, third to gain interesting theorems for free through the parametricity of world polymorphic functions (section 4.2).

Then binders are represented by natural numbers. The operator `_◁_` defines how to extend a world with a binder. Given a binder `n` and a world α, `n ◁ α` updates the world α with the value true at index `n`.

```
Binder : Set
Binder = ℕ

zeroᴮ : Binder
zeroᴮ = zero

sucᴮ : Binder → Binder
sucᴮ = suc

_◁_ : Binder → World → World
zero  ◁ []       = true  :: []
suc n ◁ []       = false :: n ◁ []
zero  ◁ (_ :: α) = true  :: α
suc n ◁ (b :: α) = b     :: n ◁ α

infixr 5 _◁_
```

The semantics of `(y ◁ α)` is to add `y` to the set α. Precisely the semantics is described by the following lemma:

```
◁-sem : ∀ α x y → (x ∈ y ◁ α) ≡ (if x ==ℕ y then ⊤ else x ∈ α)
```

Names are simply represented by a pair of a binder (a number) and a proof that this binder belongs to the indexing world. In Agda we use a record:

```
record Name α : Set where
  constructor _,_
  field
    binderᴺ  : Binder
    b∈α      : binderᴺ ∈ α
infixr 4 _,_
open Name public
```

To produce a name, `nameᴮ` simply packs the given binder and a proof which is always constructible in this simple case:

```
nameᴮ : ∀ {α} b → Name (b ◁ α)
nameᴮ b = b , {! proof omitted !}
```

The equality test `_==ᴺ_` and the exporting function `exportᴺ?` both simply compare the internal representation of names and binders.

```
_==ᴺ_ : ∀ {α} (x y : Name α) → Bool
_==ᴺ_ (x , _) (y , _) = x ==ℕ y

exportᴺ? : ∀ {b α} → Name (b ◁ α) →? Name α
exportᴺ? {b} {α} (x , pf)
 if x ==ℕ b then nothing
           else just (x , {! proof omitted !})
```

World inclusion is modeled as set-theoretic inclusion (membership preservation). This membership preservation proof is used by `coerceᴺ` to build the expected proof. All the inclusion rules (see figure 2.1) are computationally irrelevant and their implementations are omitted here.

```
infix 2 _⊆_
_⊆_ : (α β : World) → Set
α ⊆ β = ∀ x → x ∈ α → x ∈ β

coerceᴺ : ∀ {α β} → α ⊆ β → Name α → Name β
coerceᴺ α⊆β (b , b∈α) = b , α⊆β b b∈α
```

The remaining part is the fresh-for relation (_#_). To cope with the proof of `suc#`, we give two characterizations of this relation. One is a set of syntactic rules (omitted here) and the other is semantic. These presentations are equivalent [Pouillard, 2011a]. For pragmatic reasons the fresh-for relation is stronger than the non-membership relation (that is, `x # α` implies `x ∉ α` but the converse is false). The definition requires binder the be strictly greater that any binder in the world. This allows to build the `suc#`

primitive rule (omitted here) without inspecting the world. Not inspecting the worlds at run-time is necessary since we want to erase worlds.

```
_#_ : Binder → World → Set
x # α = ∀ y → y ∈ α → x > y
```

4.2 Soundness: logical relations and parametricity

Since our library is written in a type-safe language, one may wonder what soundness properties are to be proved. Moreover our names are indexed by worlds and hold membership proofs. The functions over names are given precise types and have been shown to be type-safe. The property that "well-typed programs do not go wrong" comes for free, but does not satisfy us. Indeed we have explained that certain operations (such as comparing two binders) must not be provided to programmers; the logical relation explains why this is the case.

To show that our library respects a model of good behavior with respect to names and binders, we build a model based on logical relations indexed/directed by types. This technique [Bernardy et al., 2010] is independent of this work and enables to define a notion of program equivalence. We use this technique to capture good behavior of functions involving names and binders. Using this technique, the set of specific definitions is kept to a minimum of one per introduced type (`World`, `Name`, and `_⊆_`). One proof per value introduced has to be done, which keeps the development modular and forward-compatible to the addition of more features.

This section is organized as follows. First we recall the basics of logical relations and parametricity. Then we give a toy example to practice a bit. Then relations for worlds, names, and world inclusions are given. Finally we make use of the construction to obtain free theorems [Wadler, 1989] about world-polymorphic functions over the type `Tm`.

4.2.1 Recap of the framework

A relation is said to be type-directed when it is recursively defined on the structure of types. Let $\mathcal{R}$ be such a type-directed relation, and let τ be a type. Then, $\mathcal{R}_\tau$ is a relation on values of type τ, namely $\mathcal{R}_\tau$ `:` τ `→` τ `→ Set`. Recall that `Set` is also the type of propositions in AGDA.

A type-directed relation is called "logical" when the case for functions is defined extensionally. Here "extensionally" means that two functions are related when they produce related results out of related arguments. Let $\mathtt{A}_r$ be a relation for the arguments and $\mathtt{B}_r$ a relation for results. Two functions $\mathtt{f}_1$ and $\mathtt{f}_2$ are related if and only if for every pair of arguments $(\mathtt{x}_1, \mathtt{x}_2)$ related by A_r, the results $f_1\ x_1$ and $f_2\ x_2$ are related by B_r. This definition can be given in AGDA as well:

```
RelatedFunctions Aᵣ Bᵣ f₁ f₂ =
                ∀ {x₁ x₂} → Aᵣ x₁ x₂
                          → Bᵣ (f₁ x₁) (f₂ x₂)
```

We say that a program or a value fits a logical relation when it is related to itself by the relation indexed by its type. We say that a logical relation is universal if every well-typed program fits the logical relation. This notion of universality was originally coined by John Reynolds as the “Abstraction Theorem” [Reynolds, 1983]. We call the “AGDA logical relation” the one defined by Bernardy et al. [2010] for a PTS (pure type system) and naturally extended as suggested to other features of AGDA. While no complete mechanized proof has been done for this we consider the AGDA logical relation as universal.

To simplify matters, the definitions shown here are not universe-polymorphic. The reader can find universe-polymorphic definitions in our complete AGDA development [Pouillard, 2011a].

To formally define a logical relation indexed by types, a common technique is to first inductively define the structure of types. This is known as a “universe of codes” `U`. Then one defines a function called `El` from codes to types. Finally one defines by induction a function called ⟦_⟧ from codes to relations on elements of type described by the given code. In AGDA the function ⟦_⟧ has the following type: `(τ : U) → El τ → El τ → Set`. Because of quantification and dependent-types, types contain variables, and a good deal of complexity is added to this scheme, so we opt for a lighter scheme. We do not define `U`, `El`, and ⟦_⟧.

Instead, for each type constructor κ, we define a relation ⟦κ⟧ (this is the name of the relation, since there are no spaces). For the function type constructor the `RelatedFunctions` definition above is a good start. Actually this is a fine definition for non-dependent functions. The dependent version of `RelatedFunctions`, called ⟦Π⟧ here, passes the relation argument $A_r\ x_1\ x_2$ called x_r to the relation for results B_r. In short the relation for results now depends on the relation for arguments. Here is the definition in AGDA:

```
⟦Π⟧ Aᵣ Bᵣ f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                   → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)
```

Note that this definition generalizes the case of non-dependent functions and universal quantifications as well. For non-dependent functions we simply provide a relation for results which ignores its first argument (equivalent to `RelatedFunctions` and noted _⟦→⟧_ from now on). For universal quantifications, since the arguments are types, all we need is a relation for types (members of Set₀) themselves. Following our convention we call this relation ⟦Set₀⟧. The relation ⟦Set₀⟧ A₁ A₂ is the set of relations between A₁ and A₂:

```
⟦Set₀⟧ : Set₀ → Set₀ → Set₁
⟦Set₀⟧ A₁ A₂ = A₁ → A₂ → Set₀
```

For reference, full definitions for core type theory are given in figure 4.1. As we said, we do not define the function ⟦_⟧. However we want a notation close to what this function would provide. To this end, instead of writing ⟦ τ ⟧ for a given type expression τ, we write τ where we replaced each constructor κ by ⟦κ⟧, each non dependent arrow by ⟦→⟧, each dependent arrow (x : A) → B by ⟨ xᵣ : ⟦ A ⟧ ⟩⟦→⟧ ⟦ B ⟧. By convention we subscript the names by r. Applications are translated to applications. Because of dependent types, this translation has to be extended to all terms but we do not do it here. Finally here are a few examples of the manual use of the function ⟦_⟧:

```
-- What we would like to write but cannot:
⟦ ℕ → ℕ → Bool ⟧ =
-- What we write instead:
⟦ℕ⟧ ⟦→⟧ ⟦ℕ⟧ ⟦→⟧ ⟦Bool⟧ =
-- What this means:
λ f₁ f₂ →
  ∀ {x₁ x₂} (xᵣ : ⟦ℕ⟧ x₁ x₂)
    {y₁ y₂} (yᵣ : ⟦ℕ⟧ y₁ y₂)
  → ⟦Bool⟧ (f₁ x₁ y₁) (f₂ x₂ y₂)

⟦ (A : Set₀) → A → A ⟧ =
⟦Π⟧ ⟦Set₀⟧ (λ Aᵣ → Aᵣ ⟦→⟧ Aᵣ) =
λ f₁ f₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)
    {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
  → Aᵣ (f₁ A₁ x₁) (f₂ A₂ x₂)

-- Using the notation instead of ⟦Π⟧:
⟦ (A : Set₀) → List A ⟧ =
⟨ Aᵣ : ⟦Set₀⟧ ⟩⟦→⟧ ⟦List⟧ Aᵣ =
λ l₁ l₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)
  → ⟦List⟧ Aᵣ (l₁ A₁) (l₂ A₂)
```

We now have the definition of the AGDA logical relation for the core

```
⟦Set₀⟧ : ∀ (A₁ A₂ : Set₀) → Set₁
⟦Set₀⟧ A₁ A₂ = A₁ → A₂ → Set₀

⟦Set₁⟧ : ∀ (A₁ A₂ : Set₁) → Set₂
⟦Set₁⟧ A₁ A₂ = A₁ → A₂ → Set₁

_⟦→⟧_ : ∀ {A₁ A₂ B₁ B₂} → ⟦Set₀⟧ A₁ A₂ → ⟦Set₀⟧ B₁ B₂
                        → ⟦Set₀⟧ (A₁ → B₁) (A₂ → B₂)
Aᵣ ⟦→⟧ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} → Aᵣ x₁ x₂ → Bᵣ (f₁ x₁) (f₂ x₂)

infixr 0 _⟦→⟧_

⟦Π⟧ : ∀ {A₁ A₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (Bᵣ : (Aᵣ ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ((x : A₁) → B₁ x) → ((x : A₂) → B₂ x) → Set₁
⟦Π⟧ Aᵣ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂) → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)

syntax ⟦Π⟧ Aᵣ (λ xᵣ → f) = ⟨ xᵣ : Aᵣ ⟩⟦→⟧ f

⟦∀⟧ : ∀ {A₁ A₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
        {B₁ B₂} (Bᵣ : (⟦Set₀⟧ ⟦→⟧ ⟦Set₀⟧) B₁ B₂)
      → ⟦Set₁⟧ ({x : A₁} → B₁ x) ({x : A₂} → B₂ x)
⟦∀⟧ Aᵣ Bᵣ = λ f₁ f₂ → ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂) → Bᵣ xᵣ (f₁ {x₁}) (f₂ {x₂})

syntax ⟦∀⟧ Aᵣ (λ xᵣ → f) = ∀⟨ xᵣ : Aᵣ ⟩⟦→⟧ f
```

Figure 4.1: Logical relations for core types

```
data ⟦⊥⟧ : ⟦Set₀⟧ ⊥ ⊥ -- no constructors

data ⟦Bool⟧ : ⟦Set₀⟧ Bool Bool where
  ⟦true⟧   : ⟦Bool⟧ true   true
  ⟦false⟧  : ⟦Bool⟧ false  false

data ⟦ℕ⟧ : ⟦Set₀⟧ ℕ ℕ where
  ⟦zero⟧  : ⟦ℕ⟧                zero  zero
  ⟦suc⟧   : (⟦ℕ⟧ ⟦→⟧ ⟦ℕ⟧)  suc   suc

data _⟦⊎⟧_ {A₁ A₂ B₁ B₂} (Aᵣ : ⟦Set₀⟧ A₁ A₂)
                         (Bᵣ : ⟦Set₀⟧ B₁ B₂) :
               A₁ ⊎ B₁ → A₂ ⊎ B₂ → Set₀ where
  ⟦inj₁⟧ : (Aᵣ ⟦→⟧ Aᵣ ⟦⊎⟧ Bᵣ) inj₁ inj₁
  ⟦inj₂⟧ : (Bᵣ ⟦→⟧ Aᵣ ⟦⊎⟧ Bᵣ) inj₂ inj₂
```

Figure 4.2: Logical relations for data types

type theory part. It extends nicely to inductive data types and records. The process is as follows: for each constructor κ of type τ, declare a new constructor $[\![\kappa]\!]$ whose type is $[\![\ \tau\ ]\!]\ \kappa\ \kappa$. This process applies to type constructors and data constructors of data types, and type constructors and fields of record types. For reference, the logical relations for the data types we use in this development are in figure 4.2.

Parametricity results: "Theorems For Free" We have explained how Agda's logical relation is defined ($[\![\_]\!]$) on various types. We can now more precisely explain how to use the "Abstraction theorem" associated to Agda terms.

For every well-typed term `M` *at type* τ*, the theorem* $[\![\ \tau\ ]\!]$ `M M` *is provable.*

In Type Theory (such as Agda and Coq) the specification language is the same as the programming language, which helps a lot. The theorem statement: $[\![\ \tau\ ]\!]$ `M M` is a valid type in Agda and a proof of it is a term of this type.

We warmly recommend the reader to study [Bernardy et al., 2010] to get a more in-depth understanding on the subject. We still try to give some insight on the "Abstraction Theorem". We write $\Gamma \vdash$ `M` $:\ \tau$ for a typing judgement on Agda terms. Where Γ is a typing environment holding previously defined terms and data types. While we explained how $[\![\_]\!]$ is defined on types it extends nicely on terms which is actually necessary in AGDA since terms and types cohabit. The meta-function ⟦_⟧ is also defined on environments by translating each binding of the environment.

```
Γ ⊢ M : τ ⇒ ⟦ Γ ⟧ ⊢ ⟦ M ⟧ : ⟦ τ ⟧ M M
```

This theorem reads like, for all term `M`, well-typed under environment Γ at type τ, the term ⟦ `M` ⟧ is well-typed under environment ⟦ Γ ⟧ at type ⟦ τ ⟧ `M M`. If we read the type ⟦ τ ⟧ `M M` as a proposition then the term ⟦ `M` ⟧ is its proof.

Hence, every well-typed term give rise to a free-theorem thanks to parametricity. However interesting results arose only when using polymorphism. In our setting polymorphism comes at two places. First, by making the types of our interface abstract we force the client to be polymorphic with respect to these types. Second, when the client write a world-polymorphic function this give rise to even more powerful parametricity results. In section 5.4.3 we give a more detailed account to various function types and their respective strength.

4.2.2 An example: Boolean values represented by numbers

We wish to explain how logical relations can help explain in what sense the interface to an abstract type is safe. To do so we introduce a tiny example about booleans represented using natural numbers. We want 0 to represent `false` and any other number to represent `true`. Therefore the boolean disjunction can be implemented using addition. We show that logical relations help build a model, ensure that a given implementation respects this model, and finally show that a client that uses only the interface also respect the model.

Note however that this is a toy example in several ways. There are no polymorphic functions in the interface, so no interesting free theorems are to be expected. While we could prove safety by defining a representation predicate in unary style, the logical relations approach is different. It relies on comparing concrete data as opposed to mapping to abstract data. The unary style would allow for a simpler construction, however this oversimplifies the problem here and would be no longer useful for proving our library.

Our tiny implementation of booleans using natural numbers is given below. It contains a type `B` that we want to keep abstract. It contains obvious definitions for `true`, `false`, and the disjunction _∨_. It intentionally has a dubious function `is42?`. Given that 42 and 41 both represent a truth value, the function `is42?` returns different results. This dubious function breaks our still informal expectations from such a module.

```
B : Set
B = ℕ

false : B
false = 0

true : B
true = 1

_∨_ : B → B → B
m ∨ n = m + n

is42? : B → B
is42? 42 = true
is42? _  = false
```

The next step is to define our expectations. To do so, we give a binary relation which tells when two B values have the same meaning. We do so with an (inductive) data type named ⟦B⟧ which states that 0 is related only with itself, and that any two non zero numbers are related, formalizing the intuition that two non zero numbers mean the same thing:

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

When plugged into the machinery of logical relations this single definition suffices to define a complete model of well-typed programs. However, the plumbing requires some care. While the Agda logical relation is universal, we have no such guarantee about the Agda logical relation *where* the relation for B is no longer ⟦ℕ⟧ but ⟦B⟧. Fortunately changing the relation at a given type (B here) can be done safely. All we have to do is to consider programs abstracted away from B and its operations: true, false and _∨_. This can be done either through a mechanism for abstract types, or by requiring the client to be a function taking the implementation for B and its operations as argument.

However, to use ⟦B⟧ as the relation for B, we have to show that the definitions which make use of the representation of B actually fit the relation. Since ⟦true⟧ and ⟦false⟧ are obvious witnesses for true and false, only _∨_ and is42? are left to be proved. Each time the goal to prove is systematic: wrap the type with ⟦·⟧ on each constructor and put the name of the function twice to state we want it to be related to itself. Here is the definition for _⟦∨⟧_:

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

The type of _⟦∨⟧_ means that given inputs related in the model, the results are related in the model as well. Once unfolded the type looks like:

```
_⟦∨⟧_ : ∀ {x₁ x₂} (xᵣ : ⟦B⟧ x₁ x₂)
          {y₁ y₂} (yᵣ : ⟦B⟧ y₁ y₂)
        → ⟦B⟧ (x₁ ∨ y₁) (x₂ ∨ y₂)
```

The fact that input arguments come as implicit arguments greatly shorten definitions. Now, thanks to the inductive definition of _+_, pattern-matching on the first relation suffices to reduce the goal, and allows this nice-looking definition of _⟦∨⟧_ where we see the usual lazy definition of the left biased disjunction:

```
⟦false⟧  ⟦∨⟧  x  =  x
⟦true⟧   ⟦∨⟧  _  =  ⟦true⟧
```

Let us now consider a proof for the function `is42?`. Fortunately there is no such proof since this function gladly breaks the intended abstraction. Instead we simply prove its negation by exhibiting that given two related inputs (42 and 27) we get non related outputs (since `is42? 42 = 1` and `is42? 27 = 0`).

```
¬⟦is42?⟧ : ¬((⟦B⟧ ⟦→⟧ ⟦B⟧) is42? is42?)
¬⟦is42?⟧ ⟦is42?⟧ with ⟦is42?⟧ {42} {27} ⟦true⟧
...                    | () -- absurd
```

Note that `is42?` is rejected by our model with no considerations about the other exported functions. Here we have a means to build 42 by using _∨_ and `true`. However with another implementation of _∨_ there would be no way to produce 42 and so no way to expose the wrong behavior of `is42?` using the interface. This would enable some proof techniques to actually show that no harm can be done given the limited interface and implementation. Such a proof would break down when extending the interface. In the end, using a model provides a better forward compatibility than syntactic proof techniques and enables proofs to be done in a modular way.

4.3 Relations for NomPa

For NomPa, we apply the same process as with booleans. We define our expectations, by defining relations for introduced abstract types (worlds, binders, names, fresh-for and inclusions). Finally we prove that each value and/or function exported fits the relation.

4.3.1 Relations for NomPa types

For reference the definitions are given in figure 4.3. We now describe them in turn.

While the types and the worlds define the programs we accept as valid, logical relations define when two programs have the same "meaning". For instance, valid names are those which belong to their worlds, names with the same meaning are those related by the relation between their worlds. What matters is not just the *fact* that two worlds α_1 and α_2 are related, what matters is *how* they are related, because this dictates when a name in α_1 and a name in α_2 are related. Thus we define ⟦World⟧ α_1 α_2 to be *a* set of relations between Name α_1 and Name α_2 (we note α_r a relation of type ⟦World⟧ α_1 α_2).

Which relations should be part of this set? The more relations are part of it, the more power this gives to parametricity results... However, in order for the equality test on names to inhabit the logical relation, we must restrict ⟦World⟧ α_1 α_2 to contain only relations that preserve equality in both directions, that is, only functional and injective relations:

```
Preserve-≡ ℛ =
  ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                → x₁ ≡ y₁ ↔ x₂ ≡ y₂
```

The relation ⟦Name⟧ is parameterized by a relation over worlds. Given two worlds α_1 and α_2, and a relation α_r between them (α_r has type ⟦World⟧ α_1 α_2), then a name x_1 in α_1 is related to a name x_2 in α_2 if and only if they are related by α_r: α_r x_1 x_2. In short, the relation between names is completely defined by the relation between their worlds.

The relation ⟦Binder⟧ between two binders is both simple and meaningful. It is simple since it is the full relation, meaning that any two binders are related by ⟦Binder⟧. It is meaningful since it recalls us that one binder is not better than another one. For instance when comparing object-level terms as we do in section 3.1.3, the functions cmpTm and _==$^{\mathtt{Tm}}$_ never compare binders.

This definition is meaningful for a second reason appears when one looks at the free-theorems associated with functions that expect binders. For instance, a well typed function f of type Binder → Binder → Bool can only be a constant function!

```
-- ⟦f⟧ is the parametricity theorem of f
⟦f⟧ : (⟦Binder⟧ ⟦→⟧ ⟦Binder⟧ ⟦→⟧ ⟦Bool⟧) f f

-- f-const is a corollary of ⟦f⟧.
-- f-const shows that f is a constant function.
f-const : ∀ x₁ x₂ y₁ y₂ → f x₁ y₁ ≡ f x₂ y₂
f-const x₁ x₂ y₁ y₂ with ⟦f⟧ {x₁} {x₂} _ {y₁} {y₂} _
... | ⟦true⟧  = refl
... | ⟦false⟧ = refl
```

This proves that there must be no way to distinguish two binders!

For the $\_⟦\subseteq⟧\_$ relation, we exploit the fact there is only one way to use an inclusion witness, namely $\texttt{coerce}^{\texttt{N}}$. Thus, for the purpose of building the "model", we identify inclusions with their use in $\texttt{coerce}^{\texttt{N}}$. That is, two witnesses $\alpha_1 \subseteq \beta_1$ and $\alpha_2 \subseteq \beta_2$ are related if and only if $\texttt{coerce}^{\texttt{N}}\ \alpha_1 \subseteq \beta_1$ and $\texttt{coerce}^{\texttt{N}}\ \alpha_2 \subseteq \beta_2$ are related (see figure 4.3). Another way to look at it is from the perspective of relation inclusion. A relation $\mathcal{R}_1$ is included in a relation $\mathcal{R}_2$ if and only if all pairs related by $\mathcal{R}_1$ are related by $\mathcal{R}_2$ as well. Because the function $\texttt{coerce}^{\texttt{N}}$ behaves such as the identity function, these two definitions coincide. If we expand the definitions for ⟦Name⟧ and $\_⟦\to⟧\_$, we find that $\alpha_1 \subseteq \beta_1$ and $\alpha_2 \subseteq \beta_2$ are related if and only if the relation α_r is included in the relation β_r:

```
_⟦⊆⟧_ αᵣ βᵣ α₁⊆β₁ α₂⊆β₂
  = ∀ {x₁ x₂} → (x₁ , x₂) ∈ αᵣ
               → (coerceᴺ α₁⊆β₁ x₁ , coerceᴺ α₂⊆β₂ x₂) ∈ βᵣ
```

We now have to define operations on worlds that fit the logical relation.

The case for ⟦∅⟧ is trivial and uniquely defined. There is only one relation between empty sets: the empty relation.

```
⟦∅⟧ : ⟦World⟧ ∅ ∅
```

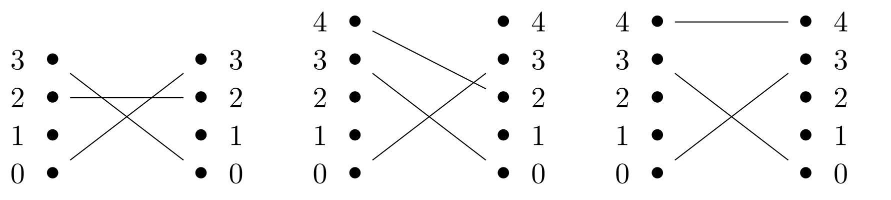
We give below a set-theoretic definition of $\_⟦\triangleleft⟧\_$. Our definition is not based on sets but equivalent to this. This new relation is shown to preserve the equalities.

```
_⟦◁⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦World⟧) _◁_ _◁_
        -- not proper Agda
bᵣ ⟦◁⟧ αᵣ ≝ { (b₁, b₂) } ∪ { (x, y) | (x, y) ∈ αᵣ ∧ x ≢ b₁ ∧ y ≢ b₂ }
```

```
Preserve-≡ : {A B : Set₀} (ℛ : A → B → Set₀) → Set₀
Preserve-≡ ℛ =
  ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                → x₁ ≡ y₁ ↔ x₂ ≡ y₂

--    ⟦World⟧ : ⟦Set₁⟧ World World
record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
  constructor _,_
  field
    ℛ        : Name α₁ → Name α₂ → Set
    ℛ-pres-≡ : Preserve-≡ ℛ

⟦Name⟧ : (⟦World⟧ ⟦→⟧ ⟦Set₀⟧) Name Name
    -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ → Name α₁ → Name α₂ → Set
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

⟦Binder⟧ : ⟦Set₀⟧ Binder Binder
      -- : Binder → Binder → Set
⟦Binder⟧ _ _ = ⊤

_⟦#⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧) _#_ _#_
   -- : ∀ {b₁ b₂} → ⟦Binder⟧ b₁ b₂ → ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂
   --   → b₁ # α₁ → b₂ # α₂ → Set
_⟦#⟧_ _ _ _ _ = ⊤

_⟦⊆⟧_ : (⟦World⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧)
          _⊆_ _⊆_
   -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ →
   --   ∀ {β₁ β₂} → ⟦World⟧ β₁ β₂ →
   --     α₁ ⊆ β₁ → α₂ ⊆ β₂ → Set
_⟦⊆⟧_ αᵣ βᵣ α₁⊆β₁ α₂⊆β₂
  = (⟦Name⟧ αᵣ ⟦→⟧ ⟦Name⟧ βᵣ) (coerceᴺ α₁⊆β₁) (coerceᴺ α₂⊆β₂)
```

Figure 4.3: Relations for NomPa types

Here is a small drawing of the effect of $\_[\![\triangleleft]\!]\_$ on relations between names:


α_r
$\langle 4,2\rangle[\![\triangleleft]\!]\ \alpha_r$
$\langle 4,4\rangle[\![\triangleleft]\!]\ \langle 4,2\rangle[\![\triangleleft]\!]\ \alpha_r$

4.3.2 NomPa values fit the relation

We now give a short overview of the proofs needed to show that our functions fit the relation. Formally, for each primitive operation p which has type τ in our interface, we have to exhibit a definition $[\![$p$]\!]$ which has type $[\![\ \tau\ ]\!]$ p p. All the proof can be found in our Agda development [Pouillard, 2011a]. Thanks to the definition of $[\![$Binder$]\!]$, $\mathtt{zero}^{\mathtt{B}}$ and $\mathtt{suc}^{\mathtt{B}}$ like any function that returns a binder immediately fit the relation. The same reasoning holds for _#∅ and suc# which immediately fit the relation. Thanks to the definition of $\_[\![\triangleleft]\!]\_$, $[\![\mathtt{name}^{\mathtt{B}}]\!]$ holds by definition. Since $[\![$¬Name∅$]\!]$ receives names in the empty world, it trivially holds. For instance here is the type signature for $[\![\mathtt{name}^{\mathtt{B}}]\!]$:

```
⟦nameᴮ⟧ : (∀⟨ αᵣ : ⟦World⟧  ⟩⟦→⟧
            ⟨ bᵣ : ⟦Binder⟧ ⟩⟦→⟧
            ⟦Name⟧ (bᵣ ⟦◁⟧ αᵣ)
          ) nameᴮ nameᴮ
--      : ∀ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂)
--          {b₁ b₂} (bᵣ : ⟦Binder⟧ b₁ b₂)
--        → ⟦Name⟧ (bᵣ ⟦◁⟧ αᵣ) (nameᴮ {α₁} b₁) (nameᴮ {α₂} b₂)
```

For $\_[\![==^{\mathtt{N}}]\!]\_$, once unfolded, the statement tells that the equality test commutes with a renaming. This means that the result of the equality test does not change when its inputs are consistently renamed. The proof for $\_[\![==^{\mathtt{N}}]\!]\_$ is done in two parts. First, we have to relate the boolean valued function $\_==^{\mathtt{N}}\_$ to the fact it decides equality on names. Second, we make full use of the equality preservation property.

```
_⟦==ᴺ⟧_ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧
            ⟦Name⟧ αᵣ ⟦→⟧
            ⟦Name⟧ αᵣ ⟦→⟧
            ⟦Bool⟧
           ) _==ᴺ_ _==ᴺ_
--       : ∀ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂)
--           {x₁ x₂} (xᵣ : ⟦Name⟧ αᵣ x₁ x₂)
--           {y₁ y₂} (yᵣ : ⟦Name⟧ αᵣ y₁ y₂)
--         → ⟦Bool⟧ (x₁ ==ᴺ y₁) (x₂ ==ᴺ y₂)
```

The proof that `exportᴺ?` is in the relation relies on two points. First, the success (returning `just` or `nothing`) of `(exportᴺ? {b} x)` depends only on the equality between `(nameᴮ b)` and `x`. Second, the worlds related by `(bᵣ ⟦◁⟧ αᵣ)` can always be split into two groups, those being related by `(bᵣ ⟦◁⟧ ⟦∅⟧)` and those related by α_r. Said differently there is no link between the two parts.

Thanks to the definition of _⟦⊆⟧_, the proof that `coerceᴺ` fits the relation is a simple application of the hypotheses. Then the real job is to show that all the inclusion rules fit the relation. This means that they all behave as the identity function. In the end all the logical relation proofs associated with all our primitive operations and rules are formally proved in our AGDA development [Pouillard, 2011a].

4.3.3 An example not fitting the relation

For instance consider the introduction of this function to compare two names:

```
_<=ᴺ_ : ∀ {α} → Name α → Name α → Bool
(m , _) <=ᴺ (n , _) = ℕ._<=_ m n
```

This function is accepted by AGDA type-checking rules but does not meet our property that well-behaving functions should send α-equivalent inputs to α-equivalent outputs.

```
¬⟦<=ᴺ⟧ : ¬((∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧
              ⟦Name⟧ αᵣ ⟦→⟧
              ⟦Name⟧ αᵣ ⟦→⟧ ⟦Bool⟧) _<=ᴺ_ _<=ᴺ_)
¬⟦<=ᴺ⟧ ⟦<=⟧ = ¬⟦Bool⟧-true-false (⟦<=⟧ ? {0 , _} {1 , _} ?
                                          {1 , _} {0 , _} ?)
  -- parts (‘?’) of the proofs are omitted for conciseness
```

Chapter 5

The de Bruijn approach

So far, we only considered a nominal representation. This is not the only way to deal with names and binders. We now focus on a different kind of representation, namely de Bruijn indices [de Bruijn, 1972]. This representation is said to be nameless because binders are no longer identified by an atom but by a notion of "distance" to the binding point.

This nameless approach solves part of the problem of representing binders by providing a canonical representation. More precisely binding occurrences are no longer named (we now use `λ.` instead of `λx.`). Bound variables are represented by the "distance" to the binding λ. This distance is the number (starting from 0) of λs to cross in order to reach the binding λ.

This approach to a safer way to program with de Bruijn indices is described in our previous paper called "Nameless, Painless" [Pouillard, 2011b]. The library specialized to de Bruijn indices is called NaPa for **Na**meless **Pa**inless. However we here stick to the general approach.

This chapter is organized as follows. The first section introduces de Bruijn indices through successive variations found in the literature. The second section 5.2, *extends* our programming interface in order to support de Bruijn indices. Also, the implementation is *extended* and not replaced. Section 5.3 explains how to use the interface to write functions on a de Bruijn representation of programs. In section 5.4 we extend the logical relation to support the de Bruijn style and we show how to exploit these properties on concrete examples.

5.1 Introduction to de Bruijn indices

Various techniques have been discovered to build a nameless representation. We have chosen a few of them which gradually set up the stage.

5.1.1 bare: The original approach

We call this one bare, because it relies solely on natural numbers. To make things more concrete here is an example of its use when defining our usual type for terms:

```
data TmB : Set where
  V     : (x     : ℕ )   → TmB
  _·_   : (t u   : TmB)  → TmB
  ƛ     : (t     : TmB)  → TmB
  Let   : (t u   : TmB)  → TmB
```

From the point of view of the binding structure, it is striking that no difference appears between the constructors of the data type. It is completely up to the programmer to manage the scoping difference introduced by ƛ and Let. This is even more worrying in the Let case since we have no clue of difference between the arguments.

Here is an example using this representation to build the λ-term for function application, namely λf.λx.f x.

```
appTmB : TmB
appTmB = ƛ (ƛ (V 1 · V 0))
```

The main advantages of this approach are its simplicity and its expressiveness. The expressiveness is maximal since no restriction is put on the usage of variables.

5.1.2 Maybe: The nested data type approach

The nested data type approach [Bellegarde and Hook, 1994, Bird and Paterson, 1999, Altenkirch and Reus, 1999] is a first step towards better properties about the binding structure of terms. Let us start with the definition of our type for terms with this approach:

```
data TmM (A : Set) : Set where
  V     : (x : A) → TmM A
  _·_   : (t u : TmM A) → TmM A
  ƛ     : (t : TmM (Maybe A)) → TmM A
  Let   : (t : TmM A) (u : TmM (Maybe A)) → TmM A
```

There are three points to look at. The type Tm^{M} is parameterized by another type called A, so we can look at it as a kind of container. Note also that the variable case V does not hold a value of type ℕ but a value of type A. Last but not least the ƛ case holds a term whose parameter is not simply a value of type `A` but a value of type `Maybe A`.

This last point makes the type `Tm`$^{\mathtt{M}}$ a nested data type, also called a non-regular type. This has the consequence of requiring polymorphic recursion to write recursive functions on such a type.

To understand why this is an adequate representation of λ-terms one has to look a bit more at the meaning of `Maybe`. If types are seen as sets, then `Maybe` takes a set and returns a set with one extra element. So each time we cross a `ƛ`, there is one extra element in the set of allowed variables, exactly capturing the fact that we are introducing a variable.

To see the difference with the previous approach, here is the λ-term for function application again:

```
appTmᴹ : Tmᴹ ⊥
appTmᴹ = ƛ (ƛ (V (just nothing) · V nothing))
```

Note the use of the empty type `⊥` to state the closedness of the `appTm`$^{\mathtt{M}}$ term. Stating this kind of properties was impossible to do with the previous approach, without resorting to logical properties on the side.

5.1.3 The `Fin` approach

Another approach already described and used in [Altenkirch, 1993, McBride and McKinna, 2004] is to index everything (terms for example) by a bound. This bound is the maximum number of distinct free variables allowed in the value. This rule is enforced in two parts: variables have to be strictly lower than their bound, and the bound is incremented by one when crossing a name abstraction (a λ-abstraction for instance, called `ƛ` here).

The type `Fin n` is used for variables and represents natural numbers strictly lower than `n`. The name `Fin n` comes from the fact that it defines finite sets of size `n`. We call this approach `Fin` for its use of this type. The definition found in Agda's standard library is the following:

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

Given this type `Fin`, one can define the term data type using this approach:

```
data Tmᶠ n : Set where
  V    : (x : Fin n) → Tmᶠ n
  _·_  : (t u : Tmᶠ n) → Tmᶠ n
  ƛ    : (t : Tmᶠ (suc n)) → Tmᶠ n
  Let  : (t : Tmᶠ n) (u : Tmᶠ (suc n)) → Tmᶠ n
```

As the previous approach, this representation helps enforce some well-formedness properties, for instance `Tmᶠ 0` is the type of closed λ-terms.

Here is the λ-term for application in this approach:

```
appTmᶠ : Tmᶠ 0
appTmᶠ = ƛ (ƛ (V (suc zero) · V zero))
```

We can easily draw a link with the `Maybe` approach. Indeed, the type `Fin (suc n)` has exactly one more element than the type `Fin n`. However, these approaches are not equivalent for at least two reasons. The `Maybe` approach can accept any type to represent variables. This makes the structure more like a container and this can be particularly helpful to define the substitution as the composition of `mapTm : ∀{A B} → (A → B) → Tm A → Tm B` and `joinTm : ∀{A} → Tm (Tm A) → Tm A` as in [Bellegarde and Hook, 1994, Bird and Paterson, 1999, Altenkirch and Reus, 1999]. The `Fin` approach has advantages as well: the representation is concrete and simpler since closer to the `bare` approach. However this apparent simplicity comes at a cost: we will see that its concrete representation is one root of the problem.

5.2 An interface for de Bruijn indices

In order to effectively program with de Bruijn indices we wanted to have a dedicated programming interface. This is indeed in contrast with our previous work [Pouillard and Pottier, 2010] where we had two implementations – nominal and de Bruijn – of a common interface.

We now claim that we need specialized operations to better support nominal and de Bruijn styles. Happily the work done for nominal nicely scales to de Bruijn indices by *adding* new primitive operations on worlds, names, and world inclusions.

Following the development that we have done for the nominal representation, we index names and terms by worlds. Everything we have built so far remains valid (and useful). The types and definitions for `World`, `∅`, `Name`, `_==ᴺ_`, `¬Name∅`, `_⊆_`, `coerceᴺ`, `⊆-refl`, `⊆-trans` and `⊆-∅` are directly re-used.

While binders (`Binder`, `_◁_`, `zeroᴮ`, `sucᴮ`...) do not need to be exposed in order to follow a de Bruijn presentation, we use them in the implementation of de Bruijn primitive operations.

To understand what definitions we need to add to deal with de Bruijn indices, we identify two characteristics of a binding construct such as λ. With de Bruijn indices two things occur at λ-abstractions: 0 is introduced as a new binder, and all the previous binders are incremented by one.

The key feature of this approach is to change how binders are referenced in a particular subterm. Our explicit notion of world can be used to express this change. We extend our notion of worlds with an operator _+1 (formally defined on page 98). Since worlds can be thought of as sets, the operator _+1 has an informal set-theoretic definition of adding one to each element of the set. Postponing a formal definition, here is the declaration:

```
_+1 : World → World
```

With this single extension of world operations (_+1), we can build name abstractions in the style of de Bruijn indices. Here is our type Tm^D revised to follow de Bruijn indices:

```
data Tmᴰ α : Set where
  V   : Name α → Tmᴰ α
  _·_ : Tmᴰ α → Tmᴰ α → Tmᴰ α
  ƛ   : Tmᴰ (0 ᴮ◁ (α +1)) → Tmᴰ α
  Let : Tmᴰ α → Tmᴰ (0 ᴮ◁ (α +1)) → Tmᴰ α
```

The only change between Tm and Tm^D is in the binding constructs. Instead of holding a binder b that is bound in the subterm, 0 ^{B} is bound in the subterm and all the previous binders are moved by one. Because this world operation is idiomatic, it deserves its own notation:

```
_↑1 : World → World
α ↑1 = 0 ᴮ◁ (α +1)
```

We call the operation _↑1 shifting by one. Note that _↑1 conveniently hides the underlying usage of binders. The type Tm^D now reads like:

```
data Tmᴰ α : Set where
  V   : Name α → Tmᴰ α
  _·_ : Tmᴰ α → Tmᴰ α → Tmᴰ α
  ƛ   : Tmᴰ (α ↑1) → Tmᴰ α
  Let : Tmᴰ α → Tmᴰ (α ↑1) → Tmᴰ α
```

We get back the nested data type approach described in section 5.1.2 if one replaces World by Set, Name by the identity, and _↑1 by Maybe. The same connection can be done with the Fin approach described in section 5.1.3 if one replaces World by ℕ, Name by Fin, and _↑1 by suc.

Our internal representation for worlds (section 4.1) using lists of boolean values is chosen to make `_+1` easy to define and to work with:

```
_+1 : World → World
α +1 = false ∷ α
```

We defined `_↑1` on top of the primitive operators `_+1` and `_◁_`. Unfolding `_↑1` and `_◁_` shows the `true` nature of `_↑1`:

```
↑1-is-true : ∀ {α} → α ↑1 ≡ true ∷ α
↑1-is-true = refl
  -- Since α ↑1 equals zero ◁ false ∷ α
  -- which reduces to true ∷ α
```

These one-step definitions `_+1` and `_↑1` are extended to any number to produce `_+ᵂ_` and `_↑_` of type `World → ℕ → World`. The world `α +ᵂ k` is `α +1...+1` (`_+1` iterated `k` times) and the world `α ↑ k` is `α ↑1... ↑1` (`_↑1` iterated `k` times).

To the best of our knowledge, making the distinction between two forms of shifting operations for worlds (namely `_+1` and `_↑1`) has never been investigated in the context of representing names and binders. As we see later in detail in section 5.4.3, this distinction is important and omitting it leads to less accurate types. Less accurate types led us to a broken definition of shifting in our first work on the subject [Pouillard and Pottier, 2010] as we detail in section 5.4.3.

ℓ-bound and ℓ-free names: In a context where we are working under ℓ binders, we call ℓ-bound, a name bound somewhere in the scope of ℓ binders. We call ℓ-free, a name that is free for all ℓ binders. In other words, a de Bruijn index is ℓ-bound if it is strictly less than ℓ; it is ℓ-free otherwise.

`zeroᴺ`: Since binders are not needed in the de Bruijn presentation, names are constructed directly. To start with, the simplest name is `zeroᴺ`. The name `zeroᴺ` represents zero and inhabits any world shifted by one. While this is a primitive operation from a de Bruijn point of view, it can be defined in terms of our nominal interface:

```
zeroᴺ : ∀ {α} → Name (α ↑1)
zeroᴺ = nameᴮ (0 ᴮ)
```

The signatures of the core primitive operations on names are given in figure 5.1. One can add any constant to a name in any world with `addᴺ`. As-

```
addᴺ      : ∀ {α} k  → Name α
                     → Name (α +ᵂ k)
subtractᴺ : ∀ {α} k  → Name (α +ᵂ k)
                     → Name α
cmpᴺ      : ∀ {α} ℓ  → Name (α ↑ ℓ)
                     → Name (∅ ↑ ℓ) ⊎ Name (α +ᵂ ℓ)

syntax addᴺ      k x = x +ᴺ k
syntax subtractᴺ k x = x ∸ᴺ k
syntax cmpᴺ      ℓ x = x <ᴺ ℓ
```

Figure 5.1: Core operations on names

$$\begin{tikzcd}
\mathtt{Name}\ (\emptyset\ +^{\mathtt{W}}\ \mathtt{k}) \arrow[d, "\mathtt{coerce}^{\mathtt{N}}\ \_"'] & \\
\mathtt{Name}\ \emptyset \arrow[r, "\neg\mathtt{Name}\emptyset"'] \arrow[d, "\mathtt{coerce}^{\mathtt{N}}\ \emptyset \subseteq \alpha"'] & \bot \\
\mathtt{Name}\ \alpha \arrow[d, bend right, "\mathtt{add}^{\mathtt{N}}\ \mathtt{k}"'] & \\
\mathtt{Name}\ (\alpha\ +^{\mathtt{W}}\ \mathtt{k}) \arrow[u, bend right, "\mathtt{subtract}^{\mathtt{N}}\ \mathtt{k}"'] \arrow[d, bend right, "\mathtt{coerce}^{\mathtt{N}}\ \_"'] & \\
\mathtt{Name}\ (\alpha \uparrow \mathtt{k}) \arrow[u, bend right, "\mathtt{cmp}^{\mathtt{N}}\ (\textit{if}\ \geq\mathtt{k})"'] \arrow[d, bend left, "\mathtt{cmp}^{\mathtt{N}}\ (\textit{if}\ <\mathtt{k})"] & \\
\mathtt{Name}\ (\emptyset \uparrow \mathtt{k}) \arrow[u, bend left, "\mathtt{coerce}^{\mathtt{N}}\ \_"] \arrow[r, bend left] & \mathtt{Fin}\ \mathtt{k} \arrow[l, bend left]
\end{tikzcd}$$

Figure 5.2: Operations on names

```
⊆-∅+1     : ∅ +1 ⊆ ∅

⊆-↑1-↑1  : ∀{α β}→ α ⊆ β ↔ α ↑1 ⊆ β ↑1

⊆-+1-+1  : ∀{α β}→ α ⊆ β ↔ α +1 ⊆ β +1

⊆-+1-↑1  : ∀{α}→ α +1 ⊆ α ↑1
```

Figure 5.3: New rules for world inclusion

suming worlds are erased at run-time, by parametricity the resulting world of $\texttt{add}^{\texttt{N}}$ clearly shows that this function does exactly its addition job. One can do the opposite operation with $\texttt{subtract}^{\texttt{N}}$. Thanks to its precise type this function is total and the inverse of $\texttt{add}^{\texttt{N}}$.

Given any world α, a name in the world $\alpha \uparrow \ell$ is either strictly lower than ℓ (and so also lives in $\emptyset \uparrow \ell$), or greater or equal to ℓ (thus also lives in $\alpha +^{\texttt{W}} \ell$). This is exactly what the function $\texttt{cmp}^{\texttt{N}}\ \ell$ is about. Given a name x, $\texttt{cmp}^{\texttt{N}}\ \ell$ x returns a disjoint sum of names which can be read in two parts. It first gives which side of the disjoint sum it stands, and second it gives a refined version of the input name x.

Figure 5.2 depicts the different types of names we have and how our operations relates them. Starting from the bottom, names with worlds of the form $\emptyset \uparrow$ k are definitely k-bound. Their value is completely known. The two arrows state the type isomorphism between Name ($\emptyset \uparrow$ k) and Fin k. Above we have names that may be bound or free (Name ($\alpha \uparrow$ k)). A dynamic test ($\texttt{cmp}^{\texttt{N}}$) can tell whether such a name is k-bound or not. Above we have names that are known to be greater than k (Name ($\alpha +^{\texttt{W}}$ k)). They are k-free names. Above we have free names (Name α). On the top we have impossible names since they are said to belong to an empty world. From them we can derive everything.

These primitive operations are enough to show an isomorphism between Fin n and Name ($\emptyset \uparrow$ n). From this, every program involving Fin can be translated into our system. This means that our system does not restrict the programmer more than the Fin approach. However as soon as one uses finer types than Name ($\emptyset \uparrow$ n), then fewer "wrong programs" type-check and more properties hold as we see in section 5.4.3.

We extend the inclusion relation with a set of new rules given in figure 5.3. The ⊆-∅+1 rule states that $\emptyset$ +1 is empty. Inclusion is preserved both ways by the contexts _↑1 and _+1. Finally _+1 can be weakened to _↑1. This accounts for the fact that α ↑1 means $\{0\} \cup (\alpha +1)$ and so is a proper superset of α +1. This set of rules is implemented according to the definition of inclusion, namely ∀ x → x ∈ α → x ∈ β. On top of these base rules, we derive others that we omit here (some are used in the code).

```
-- ⊆-↑1→↑1 is derivable
⊆-↑1→↑1 : ∀{α β}→ α ⊆ β → α ↑1 ⊆ β ↑1
⊆-↑1→↑1 = ⊆-◁ 0
```

5.2.1 Singleton worlds!

We said that our worlds denote finite subsets of ℕ and are more precise than in the Fin approach. Actually they can be as precise as we wish, since any subset of ℕ can be described by our operations on worlds (∅, _+1, and _↑1). In particular they can be singleton worlds. From singleton worlds we build singleton types for names:

```
Worldˢ : ℕ → World
Worldˢ n = ∅ ↑1 +ᵂ n

Nameˢ : ℕ → Set
Nameˢ = Name ∘ Worldˢ

_ᴺˢ : ∀ n → Nameˢ n
_ᴺˢ n = zeroᴺ +ᴺ n
```

Singleton worlds not only exist, they are also preserved by our two updating operations, namely addˢ and subtractˢ.

```
addˢ : ∀ {n} k → Nameˢ n → Nameˢ (k +ℕ n)
addˢ {n} k x = addᴺ k x
  ⟨-because ⊆-assoc-+ ⊆-refl n k -⟩

subtractˢ : ∀ {n} k → Nameˢ (k +ℕ n) → Nameˢ n
subtractˢ {n} k x = subtractᴺ k x
  ⟨-because ⊆-assoc-+′ ⊆-refl n k -⟩
```

5.3 Examples and advanced operations

5.3.1 Some convenience functions

Here are a few functions built on top of the interface (without using the concrete representation of names). sucᴺ is addᴺ 1 and sucᴺ↑ is a variant that includes a coercion from α +1 to α ↑1. The function _ᴺ turns a number n into a name that inhabits any world with at least n + 1 consecutive binders.

```
sucᴺ : ∀ {α} → Name α → Name (α +1)
sucᴺ = addᴺ 1


sucᴺ↑ : ∀ {α} → Name α → Name (α ↑1)
sucᴺ↑ = coerceᴺ ⊆-+1↑1 ∘ sucᴺ


_ᴺ : ∀ {α} n → Name (α ↑ suc n)
_ᴺ {α} n = zeroᴺ +ᴺ n
  ⟨-because α ↑1 +ᵂ n ⊆⟨ ⊆-+-↑ n ⟩
            α ↑1 ↑  n ⊆⟨ ⊆-exch-↑-↑ 1 n ⟩■
            α ↑ suc n ■ -⟩
 where open ⊆-Reasoning
```

The function subtractᴺ?, similarly to the function cmp tells whether a given name is ℓ-bound or ℓ-free. In case the name is free, an “exported” version of it is returned. This function forms the base case of “exporting” functions such as subtractTmᴰ? explained later on page 107. The function predᴺ? is a simple specialization of subtractᴺ?.

```
subtractᴺ? : ∀ {α} ℓ →  Name (α ↑ ℓ)
                      →? Name α
subtractᴺ? ℓ x
  with x <ᴺ ℓ
... | inj₁ _  = nothing
... | inj₂ x′ = just (x′ ∸ᴺ ℓ)

predᴺ? : ∀ {α} → Name (α ↑1) →? Name α
predᴺ? = subtractᴺ? 1
```

On top of subtractᴺ? we also build a convenient eliminator for names which is simply the elimination of the result of subtractᴺ?.

```
subtractWithᴺ : ∀ {α A} ℓ → A → (Name α → A)
                          → Name (α ↑ ℓ) → A
subtractWithᴺ v f = maybe f v ∘′ subtractᴺ?
```

The function shiftName ℓ k pf shifts its argument by k if this name is ℓ-free, otherwise it leaves the ℓ-bound name untouched. This function makes use of cmpᴺ and coerces the outputs to the required type. It also perform a coercion on the fly, giving extra flexibility for free.

```
shiftName : ∀ {α} ℓ k → (α +ᵂ k) ⊆ β
                      → Name (α ↑ ℓ)
                      → Name (β ↑ ℓ)
shiftName ℓ k pf x
  with x <ᴺ ℓ
... | inj₁ x′ = x′      ⟨-because pf₁ -⟩ -- ℓ-bound
... | inj₂ x′ = x′ +ᴺ k ⟨-because pf₂ -⟩ -- ℓ-free
  where
  pf₁ = ⊆-cong-↑ ⊆-∅ ℓ
  pf₂ = ⊆-trans (⊆-exch-+-+ ⊆-refl ℓ k)
                (⊆-ctx-+↑ pf ℓ)
```

The function protect↑ shifts a name transformer. Let f be a function from names to names. The function protect↑ ℓ f is a version of f that is applicable under ℓ binders. That is, if f has type Name α → Name β then protect↑ ℓ f has type Name $(\alpha \uparrow \ell)$ → Name $(\beta \uparrow \ell)$. Let x be a name under ℓ binders. When x is ℓ-bound, it is left untouched by protect↑. When x is ℓ-free, we can subtract ℓ to x, give it to f, and then add ℓ to get the result.

```
protect↑ : ∀ {α β} ℓ
           → (Name α → Name β)
           → (Name (α ↑ ℓ) → Name (β ↑ ℓ))
protect↑ ℓ f x
 with x <ᴺ ℓ
... | inj₁ x′ = x′
                ⟨-because ⊆-cong-↑ ⊆-∅ ℓ -⟩
... | inj₂ x′ = f (x′ ∸ᴺ ℓ) +ᴺ ℓ
                ⟨-because ⊆-+-↑ ℓ -⟩
```

By combining protect↑ and addᴺ one obtains an alternative implementation of shiftName called shiftName′. However shiftName is more efficient since it avoids to subtracting ℓ to add it back after adding k.

```
shiftName′ : ∀ {α β} ℓ k → (α +ᵂ k) ⊆ β
             → Name (α ↑ ℓ) → Name (β ↑ ℓ)
shiftName′ ℓ k pf = protect↑ (coerceᴺ pf ∘ addᴺ k) ℓ
```

5.3.2 Building terms

Building terms with our de Bruijn style library is as easy as building them in a plain de Bruijn index representation. The structure is exactly the same, and the variables are made of numbers (of type ℕ) using _ᴺ. Below we define representations of the identity function as `idTmᴰ`, the application operator as `appTmᴰ`, and the composition function as `compTmᴰ`:

```
idTmᴰ : ∀ {α} → Tmᴰ α
idTmᴰ = ƛ(V (0 ᴺ))


appTmᴰ : ∀ {α} → Tmᴰ α
appTmᴰ = ƛ(ƛ(V (1 ᴺ) · V (0 ᴺ)))


compTmᴰ : ∀ {α} → Tmᴰ α
compTmᴰ = ƛ(ƛ(ƛ(V (2 ᴺ)) · (V (1 ᴺ) · V (0 ᴺ)))))
```

5.3.3 Computing free variables

Our first example of functions over terms computes a list of the free variables in the input term. The function `fvᴰ` below, while straightforward, has the subtle cases of binding constructs (ƛ here). In these cases we have to remove the bound name from the list of free variables produced by the recursive call. In this nameless representation this amounts to removing occurrences of 0 and subtracting 1 to other name occurrences. This is done by the function `rm₀` which calls `predᴺ?` on each element of the list and merges the results. Note that forgetting to remove the bound variable results in a type error. In the same vein the typing of `fvᴰ` ensures that all returned variables do appear free in the given term.

```
rm₀ : ∀ {α} → List (Name (α ↑1))
            → List (Name α)
rm₀ [] = []
rm₀ (x :: xs) with predᴺ? x
...            | just x'   = x' :: rm₀ xs
...            | nothing   = rm₀ xs
```

```
fvᴰ : ∀ {α} → Tmᴰ α → List (Name α)
fvᴰ (V x)         = [ x ]
fvᴰ (fct · arg)   = fvᴰ fct ++ fvᴰ arg
fvᴰ (ƛ t)         = rm₀ (fvᴰ t)
fvᴰ (Let t u)     = fvᴰ t ++ rm₀ (fvᴰ u)
```

5.3.4 Generic traversal

As with the nominal approach (3.2) we build a generic traversal function to avoid code duplication among various operation on terms. However in this nameless style the matter is simplified. The information to carry down during the traversal can be summarized by a single natural number, namely the ℓ parameter that we have already used. This parameter records how many binders have been entered.

```
module TraverseTmᴰ
         {E}   (E-app  : Applicative E)
         {α β} (trName : ∀ ℓ → Name (α ↑ ℓ)
                               → E (Tmᴰ (β ↑ ℓ)))
  where
  open Applicative E-app

  tr : ∀ ℓ → Tmᴰ (α ↑ ℓ) → E (Tmᴰ (β ↑ ℓ))
  tr ℓ (V x)     = trName ℓ x
  tr ℓ (t · u)   = pure _·_ ⊛ tr ℓ t ⊛ tr ℓ u
  tr ℓ (ƛ t)     = pure ƛ   ⊛ tr (suc ℓ) t
  tr ℓ (Let t u) = pure Let ⊛ tr ℓ t ⊛ tr (suc ℓ) u

  trTmᴰ : Tmᴰ α → E (Tmᴰ β)
  trTmᴰ = tr 0
```

Like in section 3.2 we made a special case for variable occurrences: `trName` is not wrapped with the constructor `V`. Mapping names to terms yields capture-avoiding substitution almost for free. However, in the meantime we build `trTmᴰ′` which maps names to names. It does so by applying `pure V` to the name-to-name function:

```
open TraverseTmᴰ

trTmᴰ' :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ)
                  → E (Name (β ↑ ℓ)))
  → Tmᴰ α → E (Tmᴰ β)
trTmᴰ' E-app trName
  = trTmᴰ E-app (λ ℓ x → pure V ⊛ trName ℓ x)
    where open Applicative E-app
```

Renaming functions: In many functions over terms the handling of variables shares a common part. Given a variable under ℓ binders, we test if the variable is ℓ-bound. If so we leave it untouched, otherwise we subtract ℓ and go on a specific processing after which we add ℓ again to the free variables of the result. The traversal function is augmented by this processing of bound variables to build `renameTmᴰA`:

```
-- Like protect↑ but generalized to Applicative functors
protect↑A = {! omitted !}

renameTmᴰA : ∀ {E} (E-app : Applicative E)
               {α β} (θ : Name α → E (Name β))
             → Tmᴰ α → E (Tmᴰ β)
renameTmᴰA E-app θ
  = trTmᴰ' E-app (protect↑A E-app θ)
```

Then by picking either the identity functor (`id-app`), or the `Maybe` one we build two functions (a total and a partial one) to perform any renaming, namely `renameTmᴰ` and `renameTmᴰ?`:

```
renameTmᴰ : ∀ {α β} → (Name α → Name β)
                    → Tmᴰ α → Tmᴰ β
renameTmᴰ = trTmᴰ' id-app ∘ protect↑
-- or
-- renameTmᴰ = renameTmᴰA id-app

renameTmᴰ? : ∀ {α β} → (Name α →? Name β)
                     → Tmᴰ α →? Tmᴰ β
renameTmᴰ? = renameTmᴰA Maybe.applicative
```

Lifting name functions: Any operation on names can now be lifted to terms. The function $\texttt{addTm}^{D}$ lifts the function $\texttt{add}^{N}$ from names to terms. It takes a number `k` and a term and adds `k` to all the free variables:

```
addTm^D : ∀ {α} k → Tm^D α → Tm^D (α +^W k)
addTm^D = renameTm^D ∘ add^N
```

The functions $\texttt{subtractTm}^{D}$ and $\texttt{subtractTm}^{D}\texttt{?}$ respectively lift the functions $\texttt{subtract}^{N}$ and $\texttt{subtract}^{N}\texttt{?}$ from names to terms. They enable to subtract a number `k` to all the free variables of a term:

```
subtractTm^D : ∀ {α} k → Tm^D (α +^W k) → Tm^D α
subtractTm^D = renameTm^D ∘ subtract^N

subtractTm^D? : ∀ {α} ℓ → Tm^D (α ↑ ℓ) →? Tm^D α
subtractTm^D? = renameTm^D? ∘ subtract^N?
```

While $\texttt{coerce}^{N}$ can be lifted to terms in the same way, we can by-pass the `protect↑` dynamic tests and directly "protect" the inclusion witness with an appropriate inclusion rule. Put differently there are two ways to go from a proof `pf` of $\alpha \subseteq \beta$ to (`Name` $(\alpha \uparrow \ell)$ → `Name` $(\beta \uparrow \ell)$): one is to protect the result of coerce (`protect↑` ∘ $\texttt{coerce}^{N}$ `pf`) and the other is to coerce with protected inclusion witness ($\texttt{coerce}^{N}$ ∘ `⊆-cong-↑ pf`):

```
coerceTm^D : ∀ {α β} → α ⊆ β → Tm^D α → Tm^D β
coerceTm^D pf = trTm^D' id-app (coerce^N ∘ ⊆-cong-↑ pf)
-- or less efficiently:
-- coerceTm^D = renameTm^D ∘ coerce^N
```

Lifting $\texttt{add}^{N}$ to terms can be done more efficiently as well. Indeed $\texttt{addTm}^{D}$ internally uses $\texttt{renameTm}^{D}$ which uses `protect↑`. Here the dynamic test performed by `protect↑` is necessary. However when the name is ℓ-free we subtract ℓ and add it back after adding `k`. The function `shiftName` avoids this extra computation, hence the following $\texttt{shiftTm}^{D}$:

```
shiftTm^D : ∀ {α β} k → (α +^W k) ⊆ β → Tm^D α → Tm^D β
shiftTm^D k p = trTm^D' id-app (λ ℓ → shiftName ℓ k p)
-- or less efficiently:
-- shiftTm^D k pf = renameTm^D (coerce^N pf ∘ add^N k)
```

The function $\texttt{closeTm}^{D}\texttt{?}$ is built like in nominal style (section 3.2):

```
closeTm^D? : ∀ {α} → Tm^D α →? Tm^D ∅
closeTm^D? = renameTm^D? (const nothing) -- any free var leads to a failure
```

Capture avoiding substitution To implement capture avoiding substitution for the type Tm^{D}, all we need is a specific function trName for trTm^{D}. Here substitutions are represented as functions from names to terms. The function substVarTm^{D} handles the case for variables. This function is very close to protect↑ but extended to functions returning terms.

```
substVarTmᴰ : ∀ {α β} → (Name α → Tmᴰ β) →
               ∀ ℓ → Name (α ↑ ℓ) → Tmᴰ (β ↑ ℓ)
substVarTmᴰ f ℓ x
with x <ᴺ ℓ
... | inj₁ x′ = V (x′ ⟨-because ⊆-cong-↑ ⊆-∅ ℓ -⟩)
... | inj₂ x′ = shiftTmᴰ ℓ (⊆-+-↑ ℓ) (f (x′ ∸ᴺ ℓ))
```

The main function substTm^{D} instantiates trTm^{D} with the identity applicative functor and substVarTm^{D}:

```
substTmᴰ : ∀ {α β} → (Name α → Tmᴰ β)
                   → (Tmᴰ α → Tmᴰ β)
substTmᴰ = trTmᴰ id-app ∘ substVarTmᴰ
```

As an illustration, the function β-red^{D} performs a β-reduction at the root of the term using the function substTm^{D}:

```
β-redᴰ : ∀ {α} → Tmᴰ α → Tmᴰ α
β-redᴰ (ƛ fct · arg) = substTmᴰ (subtractWithᴺ 1 arg V) fct
β-redᴰ t              = t
```

5.3.5 Nameless term comparison

We show that all the subtle work is done at the level of names in a separate and reusable function called cmpName↑. This function takes a function that compares two free names and builds one that compares two names under ℓ bindings. It does so by comparing both of them to ℓ. If they are both bound they can be safely compared using _==^{N}_ since they now are of the same type. If they are both free, they can be compared using the function received as argument. Otherwise they are different.

```
cmpName↑ : ∀ {α β} ℓ → |Cmp| Name α β
                      → |Cmp| Name (α ↑ ℓ) (β ↑ ℓ)
cmpName↑ ℓ Γ x y with x <ᴺ ℓ | y <ᴺ ℓ
... | inj₁ x′ | inj₁ y′  = x′ ==ᴺ y′
... | inj₂ x′ | inj₂ y′  = Γ (x′ ∸ᴺ ℓ) (y′ ∸ᴺ ℓ)
... | _       | _        = false
```

The function `cmpTm`^{D} structurally compares two terms in a simple way, only keeping track of the number of traversed binders and calling `cmpName↑` at variables:

```
cmpTmᴰ : ∀ {α β} → |Cmp| Name α β → |Cmp| Tmᴰ α β
cmpTmᴰ {α} {β} Γ = go 0 where
  go : ∀ ℓ → |Cmp| Tmᴰ (α ↑ ℓ) (β ↑ ℓ)
  go ℓ (V x)       (V y)       = cmpName↑ ℓ Γ x y
  go ℓ (t · u)     (v · w)     = go ℓ t v ∧ go ℓ u w
  go ℓ (ƛ t)       (ƛ u)       = go (suc ℓ) t u
  go ℓ (Let t u) (Let v w) = go ℓ t v ∧ go (suc ℓ) u w
  go _ _ _                     = false
```

By instantiating the name comparator Γ by the homogeneous name comparison function `_==`^{N}`_` we get an equality test on nameless terms. While this function works on open terms, they are requested to inhabit a common world.

```
_==Tmᴰ_ : ∀ {α} → |Cmp| Tmᴰ α α
_==Tmᴰ_ = cmpTmᴰ _==ᴺ_
```

However in the particular case of comparing terms in a common world, we can define a simpler and a little faster equality test on terms. To do so we exploit the fact that the name abstraction is canonical and thus the term comparison is kept homogeneous during the whole comparison. In particular the function `cmpName↑` is not used.

```
eqTmᴰ : ∀ {α} → |Cmp| Tmᴰ α α
eqTmᴰ (V x)       (V y)       = x ==ᴺ y
eqTmᴰ (t · u)     (v · w)     = eqTmᴰ t v ∧ eqTmᴰ u w
eqTmᴰ (ƛ t)       (ƛ u)       = eqTmᴰ t u
eqTmᴰ (Let t u) (Let v w) = eqTmᴰ t v ∧ eqTmᴰ u w
eqTmᴰ _           _           = false
```

5.4 Logical Relations for de Bruijn indices

5.4.1 Relations for the de Bruijn world operations

We now have to define how our operation on worlds fits the logical relation. Figure 5.4 depicts how the operators `_⟦+1⟧` and `_⟦↑1⟧` apply to the graph of a relation.

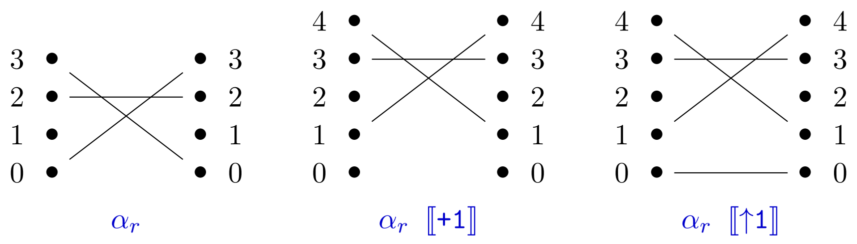


Figure 5.4: Shifting versus adding example

```
_⟦+1⟧   : (⟦World⟧ ⟦→⟧ ⟦World⟧) _+1 _+1
αᵣ ⟦+1⟧ ≝ { (x+1, y+1) | (x, y) ∈ αᵣ } -- not proper Agda

_⟦↑1⟧ : (⟦World⟧ ⟦→⟧ ⟦World⟧) _↑1 _↑1
_⟦↑1⟧ αᵣ = ⟦zeroᴮ⟧ ⟦◁⟧ (αᵣ ⟦+1⟧)
```

5.4.2 De Bruijn functions fit the relation

We now give a short overview of the proofs needed to show that our functions fit the relation. Thanks to the definition of _⟦+1⟧, $\texttt{add}^{\texttt{N}}$ 1 and $\texttt{subtract}^{\texttt{N}}$ 1 fit the relation. These two are later extended to $\texttt{add}^{\texttt{N}}$ k and $\texttt{subtract}^{\texttt{N}}$ k by an induction on k. Here are for instance the type signatures for ⟦$\texttt{add}^{\texttt{N}}$⟧ and ⟦$\texttt{cmp}^{\texttt{N}}$⟧:

```
⟦addᴺ⟧ : (∀⟨ αᵣ : ⟦World⟧      ⟩⟦→⟧
          ⟨ kᵣ : ⟦ℕ⟧          ⟩⟦→⟧
           ⟦Name⟧ αᵣ ⟦→⟧
           ⟦Name⟧ (αᵣ ⟦+ᵂ⟧ kᵣ)) addᴺ addᴺ

⟦cmpᴺ⟧ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧
          ⟨ kᵣ : ⟦ℕ⟧ ⟩⟦→⟧
           ⟦Name⟧ (αᵣ ⟦↑⟧ kᵣ) ⟦→⟧
           ⟦Name⟧ (⟦∅⟧ ⟦↑⟧ kᵣ) ⟦⊎⟧
           ⟦Name⟧ (αᵣ ⟦+ᵂ⟧ kᵣ)
         ) cmpᴺ cmpᴺ
```

The definition of $\texttt{cmp}^{\texttt{N}}$ is not a simple induction on its first argument. It calls _<=_ (which does an induction) and returns a Boolean value. Based on this Boolean value, the function $\texttt{cmp}^{\texttt{N}}$ returns either $\texttt{inj}_1$ or $\texttt{inj}_2$ (the constructors of _⊎_) with the same name (with a different proof). To prove ⟦$\texttt{cmp}^{\texttt{N}}$⟧ we show the equivalence with a simpler inductive function and show that this simpler function is in the relation. Thanks to the ex-

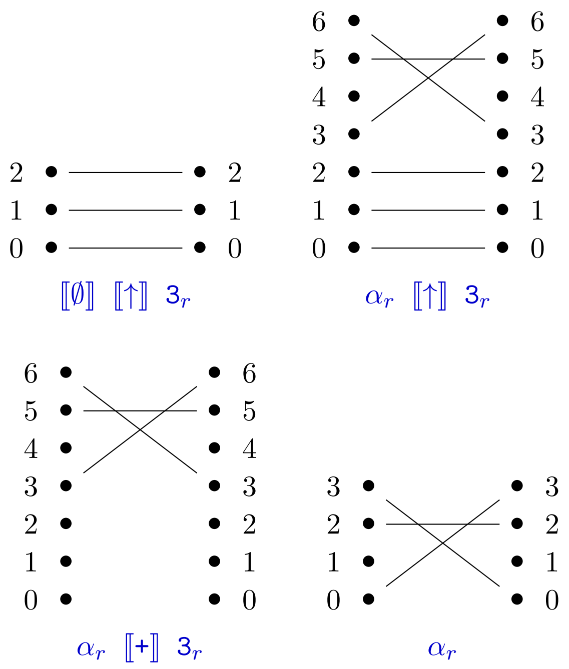


Figure 5.5: Various term types: from concrete to abstract

tensionality of the logical relation, no additional axiom is required to show that $\texttt{cmp}^{\texttt{N}}$ fits the relation.

We have shown that the new inclusion rules (Figure 5.3) fit the relation.

5.4.3 On the strength of free theorems

Every well-typed term comes with a free theorem (section 4.2.1). However depending on the type of the term the strength of the theorem varies a lot. For instance at type $\mathbb{N}$ the theorem only tell that the term is equal to itself. At type $\mathbb{N} \to \texttt{Bool}$ the theorem says no more than the function is deterministic. At type $\forall\{\texttt{A} : \texttt{Set}\} \to \texttt{A} \to \texttt{A}$ it says that the function behaves as the identity function, which is much stronger. We now give a few elements of what can affect the strength of free theorems in our context. We continue to use our type $\texttt{Tm}^{\texttt{D}}$ to represent some data structures with names and binders but it could be any other. Moreover we focus on functions from terms to terms, where the input and the output worlds are the same. We discuss the strength of the abstraction level of the input or output term of such a function. From this perspective figure 5.5 depicts the relation corresponding to these types.

Various term types The weakest type we can give to a function taking a term with names and binders is $\forall\{\ell\} \to \texttt{Tm}^{\texttt{D}}\ (\emptyset \uparrow \ell) \to \ldots$. A term received by such a function can have a statically unknown number of distinct free variables, but we know that these variables are comprised in the interval $[0 \mathrel{..} \ell\text{-}1]$. The free theorem of $\forall\{\ell\} \to \mathtt{Tm}^{\mathtt{D}}\ (\emptyset \uparrow \ell) \to \mathtt{Tm}^{\mathtt{D}}\ (\emptyset \uparrow \ell)$ says no more than "the function is deterministic" because the logical relation $[\![\mathtt{Tm}^{\mathtt{D}}]\!]\ (\emptyset \uparrow \ell)$ is the identity. We also know because of typing that the resulting term cannot have ℓ-free occurrences.

The type $\mathtt{Tm}\ (\alpha \uparrow \ell)$ is more abstract than $\mathtt{Tm}^{\mathtt{D}}\ (\emptyset \uparrow \ell)$, an unknown world is used instead of the empty world. However this type still shows a known amount of bindings. Then, a function of type $\forall\{\alpha\ \ell\} \to \mathtt{Tm}^{\mathtt{D}}\ (\alpha \uparrow \ell) \to \mathtt{Tm}^{\mathtt{D}}\ (\alpha \uparrow \ell)$ has a stronger free theorem than the previous one. It says that the function commutes with a renaming of the variables in the world α (section 5.4.4). This is a common type to deal with open terms under a partial environment:

```
some-op : ∀{α ℓ} → (Fin ℓ → Info) → Tmᴰ (α ↑ ℓ) → Tmᴰ (α ↑ ℓ)
some-op Γ t = {! omitted !}
```

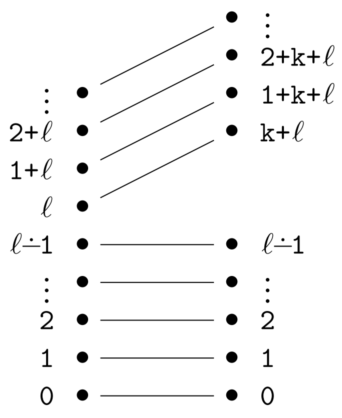
The most abstract type for terms makes no assumptions on their free variables. This can be done by quantifying by an arbitrary world. The free theorem associated with the type $\forall\ \{\alpha\} \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \alpha$ is then stronger and says that the function commutes with any renaming of the free variables. This particular type is studied in detail in section 5.4.4.

Extra arguments Note that adding extra arguments to a function can drastically affect the strength of its free theorem. An extreme example is the type $\forall\{\alpha\} \to (\alpha \equiv \emptyset) \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \alpha$, which is ruined by its $\alpha \equiv \emptyset$ argument. This argument makes the type equivalent to $\mathtt{Tm}^{\mathtt{D}}\ \emptyset \to \mathtt{Tm}^{\mathtt{D}}\ \emptyset$ which has a trivial free-theorem. While the above example is extreme, this is an important point to watch out for when adding extra arguments to a function. Another example is the type $\forall\{\alpha\ \beta\} \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \beta \to \mathtt{Tm}^{\mathtt{D}}\ \beta$ versus $\forall\{\alpha\ \beta\} \to (\mathtt{Name}\ \alpha \to \mathtt{Name}\ \beta \to \mathtt{Bool}) \to \mathtt{Tm}^{\mathtt{D}}\ \alpha \to \mathtt{Tm}^{\mathtt{D}}\ \beta \to \mathtt{Tm}^{\mathtt{D}}\ \beta$. The first one cannot compare the free variables of the two given terms while the second can apply a user-supplied function to do so.

Shifting versus adding Last but not least using $\_{+^{\mathtt{W}}}\_$ instead of $\_{\uparrow}\_$ significantly improves the strength of the associated free theorem. Consider the function `protectedAdd`:

```
protectedAdd : ∀ {α} ℓ k → Name (α ↑ ℓ)
                         → Name (α +ᵂ k ↑ ℓ)
protectedAdd ℓ k = protect↑ ℓ (addᴺ k)
```

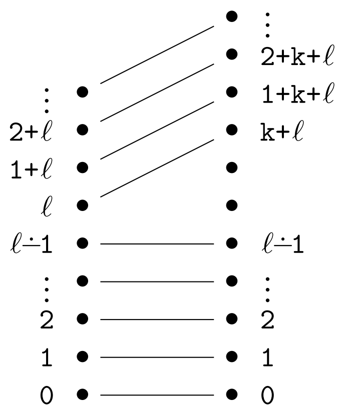
The following diagram depicts the graph behind the function `protectedAdd ℓ k`:



Consider now a weaker type, namely:

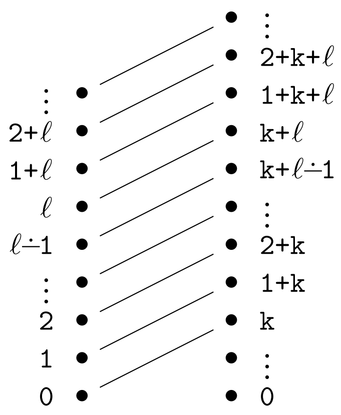
```
protectedAdd↑ : ∀ {α} ℓ k → Name (α ↑ ℓ)
                          → Name (α ↑ k ↑ ℓ)
protectedAdd↑ ℓ k = protect↑ ℓ (addᴺ↑ k)
```

We depict the graph behind the function protectedAdd↑. The graph is actually the same than the graph for the function protectedAdd. We extend the set of nodes to match the codomain stated by the type.



We simply replaced the occurrence of `_+ᵂ_` by `_↑_`. The consequences of this change are disastrous: this type allows more behaviors for its functions. Indeed the `addᴺ k` function can be given the latter type (but not the former) when using the appropriate inclusion witness to exploit the commutativity of `_↑_`:

```
unprotectedAdd : ∀ {α} k ℓ → Name (α ↑ ℓ)
                             → Name (α ↑ k ↑ ℓ)
unprotectedAdd k ℓ
  = coerceᴺ (⊆-exch-↑-↑′ ℓ k) ∘ addᴺ↑ k
  -- this is ok since (α ↑  k) ↑ ℓ ≡ (α ↑ ℓ) ↑  k
  -- whereas          (α +ᵂ k) ↑ ℓ ≢ (α ↑ ℓ) +ᵂ k
```



In our previous work [Pouillard and Pottier, 2010] we had a function called `shiftName` which was incorrectly defined with the `unprotectedAdd` behavior. Indeed the correct expected behavior is the one of `protectedAdd↑`. In this paper we had only the `_↑_` operation, and thus we missed the finer type we can give to `protectedAdd`. It took us a long time to discover this mistake, since we thought that our logical relation argument was enough. The logical relations proofs were indeed correct, the free-theorem were also correct but were not always "saying" what we expected. In summary we want to highlight that all the logical relations results have to be taken with great care. A weaker function type than expected can ruin the intended informal properties.

5.4.4 Using logical relations and parametricity

To formally show that a world-polymorphic function `f` of type ∀ {α} → Tmᴰ α → Tmᴰ α commutes with a renaming of the free variables, we proceed as follows. First we recall the natural definition of logical relation on the type Tmᴰ. Second we present the type `Ren` of renamings as injective functions. Third renaming is shown equivalent to being related at the type Tmᴰ. Finally we prove our commutation lemma by using the free-theorem associated to the function `f`.

The logical relation for the type Tmᴰ is mechanical. It states that two terms are related if they have the same structure and related free variables:

```
data ⟦TmD⟧ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂) :
           TmD α₁ → TmD α₂ → Set where
  ⟦V⟧      : ∀ {x₁ x₂} (xᵣ : ⟦Name⟧ αᵣ x₁ x₂)
             → ⟦TmD⟧ αᵣ (V x₁) (V x₂)
  _⟦·⟧_    : ∀ {t₁ t₂ u₁ u₂}
              (tᵣ : ⟦TmD⟧ αᵣ t₁ t₂)
              (uᵣ : ⟦TmD⟧ αᵣ u₁ u₂)
             → ⟦TmD⟧ αᵣ (t₁ · u₁) (t₂ · u₂)
  ⟦ƛ⟧      : ∀ {t₁ t₂} (tᵣ : ⟦TmD⟧ (αᵣ ⟦↑1⟧) t₁ t₂)
             → ⟦TmD⟧ αᵣ (ƛ t₁) (ƛ t₂)
  ⟦Let⟧    : ∀ {t₁ t₂ u₁ u₂}
              (tᵣ : ⟦TmD⟧ αᵣ t₁ t₂)
              (uᵣ : ⟦TmD⟧ (αᵣ ⟦↑1⟧) u₁ u₂)
             → ⟦TmD⟧ αᵣ (Let t₁ u₁) (Let t₂ u₂)
```

Then we need a notion of renaming. We choose to use injective functions over names. The type for a renaming is Ren, and the functions ⟨_⟩ and ⟪_⟫ respectively convert a renaming to a function over terms, and to a relation over worlds:

```
Ren : (α β : World) → Set

⟨_⟩ : ∀ {α β} → Ren α β → TmD α → TmD β

⟪_⟫ : ∀ {α β} → Ren α β → ⟦World⟧ α β
```

We now observe that given a renaming Φ, it is equivalent for two terms t_1 and t_2 to be related by $[\![\mathtt{Tm}^{\mathtt{D}}]\!]$ ⟪ Φ ⟫ and for t_2 to be equal to t_1 renamed with Φ. Put differently at the type Tm the relation coincides with the α-equivalence (using the identity renaming for Φ).

```
⟦TmD⟧⇔rename :
  ∀ {α β} (Φ : Ren α β) {t₁ t₂}
  → ⟦TmD⟧ ⟪ Φ ⟫ t₁ t₂ ⇔ (⟨ Φ ⟩ t₁) ≡ t₂
```

Finally given a function `f` and a proof `f`$_r$ that `f` is in the logical relation, we can show that any renaming `Φ` commutes with the function `f`. To prove this we apply our `⟦Tm`$^{\mathtt{D}}$`⟧⇔rename` lemma in both directions and use `f`$_r$ with the renaming `Φ` lifted as a `⟦World⟧`.

```
-- Pointwise equality
f ≗ g = ∀ x → f x ≡ g x

ren-comm :
  (f : ∀ {α} → Tmᴰ α → Tmᴰ α)
  (fᵣ:(∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧ ⟦Tmᴰ⟧ αᵣ ⟦→⟧ ⟦Tmᴰ⟧ αᵣ)f f)
  → ∀ {α β} (Φ : Ren α β)
  → ⟨ Φ ⟩ ∘ f ≗ f ∘ ⟨ Φ ⟩
ren-comm f fᵣ Φ t
  = ⟦Tmᴰ⟧⇒rename Φ
        (fᵣ ⟪ Φ ⟫ (rename⇒⟦Tmᴰ⟧ Φ ≡.refl))
```

Chapter 6

Variations and Related Work

This chapter covers various experimentations and connects our approach to more existing work. While some of the connections are not fully developed, they show the path to a complete connection and still give some insight.

The chapter is structured as follows. The first section explores the use of *atoms* as a separate kind of *global* names. In this section we explain the construction of Locally Nameless and Locally Named libraries where the type for global names is `Atom` and the type for local names is our type `Name`. We discuss the limitations of these approaches as programming interfaces, but underline the complementary role they can have when combined with our library.

In section 6.2 we describe another nameless approach from N.G. de Bruijn [de Bruijn, 1972] that is now called "de Bruijn levels". We show how our existing library accommodates this style with only the addition of two inclusion rules.

In section 6.3 we describe a common interface for binders called links. This interface revives the presentation of our previous work [Pouillard and Pottier, 2010]. We show how the "unifying" part of this link-based approach can be integrated in our library. This approach enables greater code re-use without sacrificing much of the flexibility of our approach.

In section 6.4 we briefly explore how different approaches can be combined in order to be used at the same time. To illustrate this we combine nominal and de Bruijn indices in the same data type. This could allow for instance to convert a subterm to de Bruijn indices in order to apply a given algorithm.

In section 6.5 we cover a series of description languages used to describe "nominal types". We show how the use of our explicit worlds can cope with pretty much everything these languages can describe. To do so we explain how each construct of these description languages can be expressed with worlds.

Section 6.6 discusses some more related work. We focus on relatively recent work with which we have not yet drawn a concrete connection so far.

6.1 Dynamically stratified representations

Well known techniques such as *Locally Nameless* [Charguraud, 2011, Aydemir et al., 2008], or *Locally Named* [Pollack et al., 2011, Sato and Pollack, 2010] can be qualified as "dynamically stratified representations". Indeed these techniques use a double representation of variables, one for *local* variables, and one for *global* variables. The idea is to use local variables as the internal representation of the binding structure and to provide operations to switch back and forth from local to global variables. Name abstractions are introduced with the *close* operation. Given a subterm `t` and a global name `X`, a name abstraction is built by picking a local name `x` and replacing `X` by `x` in `t`, to form the body of the newly created name abstraction. Conversely, to *open* a name abstraction, one gives a global name `X` and an abstraction to be opened. The abstraction is internally made of a local name `x` and a subterm `t`. The opening operation returns `t` where `X` replaces `x`.

The main purpose of this double representation is to hide the handling of local names to the client and to let the client use only global names. To be sure to never encounter a local name, we want our terms to be *locally closed*, meaning without any free local variables. This property guarantees that starting from a locally closed term and properly opening abstractions, we never reach a local variable. Hence we only deal with global variables and avoid the hard work implied by local names and their abstractions.

While the representation for local names distinguishes Locally Nameless and Locally Named techniques, both use a nominal approach to represent global names. Hence we reuse *atoms* from section 2.1.1 and package them up as a little module. These atoms form our set of global names. This tiny module contains a type for atoms `Atom`, an equality test `_==ᴬ_`, and an injective function from natural numbers to atoms:

```
module Data.Atom where
  abstract
    Atom : Set
    Atom = ℕ

    _==ᴬ_ : (x y : Atom) → Bool
    _==ᴬ_ = _==ℕ_

    _ᴬ : ℕ → Atom
    _ᴬ = id
```

Since our library supports both de Bruijn indices and the nominal style, the type Name can be used to represent local names. Our type for names is indexed by a world, hence internally, terms are indexed as well. However the manipulated terms being locally closed, the world is always the empty world in the end. Hence we happen to reach a local name, this name inhabits the empty world. Thanks to the function Name∅-elim (based on ¬Name∅), we know that such a case is impossible.

6.1.1 Locally closed terms

To illustrate these techniques and how we can use our library to develop these techniques, we use a type for terms like we have done so far. However we do not define the type Tm^{A} yet (which is not the type Tm^{A} from section 2.1.1): instead, we assume the following types and data constructors.

```
-- The type for terms
Tm^A : Set

-- The type for name-abstractions,
-- abstracting one name over one term
-- example: [x]t
AbsTm : Set

-- Constructor for ‘‘global’’ variables
V^A  : (x : Atom) → Tm^A

-- Constructor for ‘‘local’’ variables
V^N  : (x : Name ∅) → Tm^A

-- Constructor for λ-abstractions
ƛ    : (abs : AbsTm) → Tm^A

-- Function application
_·_ : (t u : Tm^A) → Tm^A

-- Local definition
-- example: ‘‘let x = t in u’’ ⇒ Let t [x]u
Let : (t : Tm^A) → (abs : AbsTm) → Tm^A
```

In order to deal with the type AbsTm, the two functions openTm and closeTm are given to eliminate and introduce values of type AbsTm.

```
closeTm : Atom → Tm^A → AbsTm
openTm  : Atom → AbsTm → Tm^A
```

Generally when writing the function openTm, it is no more difficult to plug a term than it is to plug an atom. This is especially true since the plugged term is locally closed, since no capture can happen. Hence we prefer to define the function openSubstTm for each implementation and define openTm generically.

```
openSubstTm : Tmᴬ → AbsTm → Tmᴬ

openTm : Atom → AbsTm → Tmᴬ
openTm = openSubstTm ∘ Vᴬ
```

6.1.2 Building terms

To build terms easily one can build smart constructors hiding the underlying use of closeTm. Hence we introduce two functions ƛ′ and let′ to build λ-abstractions and local definitions in a more natural way. In particular in the case of let′ this is the occasion to put the arguments in the expected order:

```
ƛ′ : Atom → Tmᴬ → Tmᴬ
ƛ′ x t = ƛ (closeTm x t)

let′ : Atom → Tmᴬ → Tmᴬ → Tmᴬ
let′ x t u = Let t (closeTm x u)
```

We can now build various terms quite easily:

```
idᵀ : Tmᴬ
idᵀ = ƛ′ x (Vᴬ x)
  where x = 0 ᴬ

falseᵀ : Tmᴬ
falseᵀ = ƛ′ x (ƛ′ x (Vᴬ x))
  where x = 0 ᴬ

trueᵀ : Tmᴬ
trueᵀ = ƛ′ x (ƛ′ y (Vᴬ x))
  where x = 0 ᴬ
        y = 1 ᴬ

apᵀ : Tmᴬ
apᵀ = ƛ′ x (ƛ′ y (Vᴬ x · Vᴬ y))
  where x = 0 ᴬ
        y = 1 ᴬ
```

```
ΩT : TmA
ΩT = let′ δ (ƛ′ x (VA x · VA x))
            (VA δ · VA δ)
  where δ = 0 A
        x = 1 A
```

Opening and closing operations need to deal with local variables. However any other operation could be built with only openTm and closeTm, hence without any knowledge of the local variables. Using openTm and closeTm also has three main limitations: convenience, performance and non-structural recursion. Since all of them apply on most functions over terms we take a concrete example with a function size which tries to count the number of constructors in a term:

```
size : TmA → ℕ
size (VA _) = 1
size (fct · arg) = 1 + size fct + size arg
size (ƛ abs) = 1 + size (openTm (0 A) abs)
size (Let t abs) = 1 + size t + size (openTm (0 A) abs)
size (VN x) = Name∅-elim x
```

Convenience While closeTm can be hidden with smart constructors, openTm has to be explicitly used, which hinders pattern-matching. Indeed, without view-patterns (see [?] for the seminal work on view patterns), pattern-matching has to be stopped at abstractions, in order to call openTm. The deeper the pattern, the worse the trouble. Moreover in this presentation we have to provide a global name to open the abstraction, which often requires an atom supply to give different atoms each time. In the size function we constantly open abstractions with the atom 0 A which is acceptable only because we do not care about the identity of variables in this example.

6.1.3 Performances

From a performance point of view openTm and closeTm require to traverse the whole subterm. They thus increase the asymptotic complexity of most algorithms. For a term of size n the asymptotic complexity of openTm and closeTm is naturally between O(n) and O(n*log(n)). To our knowledge the complexity cannot be reduced much without losing properties such as persistence. In Cαml [Pottier, 2006] the opening operation lazily performs the required substitution. We describe Cαml and connect it to our work in section 6.5.2. For traditional algorithms the cost of opening (and closing) is amortized but the presence of delayed substitutions makes the worst-case complexity exponential.

Let us look at how performance degrades on some algorithms. For instance an algorithm on a term that is simple enough to not involve the binding structure and that usually has a linear complexity, has a quadratic complexity when using `openTm` and `closeTm`. Algorithms to extract some information such as the size of a term (our function `size`) or the presence of some constructs in a term are of this kind. Algorithms to transform some non-binding constructs such as constant-folding, control-flow rewriting, assertions insertion or suppression, etc. are of this kind as well. Even some algorithms operating on the binding structure would benefit from avoiding `openTm` and `closeTm`. Here are for instance two functions in nominal style to transform back and forth, local `let` bindings into applied λ-abstractions. With our library no run-time cost is incurred and these two operations work in linear time:

```
module ƛ⇒Let where
  ⟪_⟫ : ∀ {α} → Tm α → Tm α
  ⟪ ƛ b t · u ⟫ = Let b ⟪ u ⟫ ⟪ t ⟫
  ⟪ V x       ⟫ = V x
  ⟪ t · u     ⟫ = ⟪ t ⟫ · ⟪ u ⟫
  ⟪ ƛ b t     ⟫ = ƛ b ⟪ t ⟫
  ⟪ Let b t u ⟫ = Let b ⟪ t ⟫ ⟪ u ⟫

module Let⇒ƛ where
  ⟪_⟫ : ∀ {α} → Tm α → Tm α
  ⟪ Let b t u ⟫ = ƛ b ⟪ u ⟫ · ⟪ t ⟫
  ⟪ V x       ⟫ = V x
  ⟪ t · u     ⟫ = ⟪ t ⟫ · ⟪ u ⟫
  ⟪ ƛ b t     ⟫ = ƛ b ⟪ t ⟫
```

6.1.4 Non-structural recursion

The third main limitation of these approaches – at least from the point of view of a total language such as AGDA – is that `openTm` hinders structural recursion. Structural recursion is a syntactic criterion requiring to make recursive calls only on syntactic subterms of the input. Indeed if `t` is a subterm of a term `u`, `openTm s t` is not a subterm of `u`, preventing any recursive call. Of course we can claim for a better support for this kind of recursion, since `openTm` preserves the *size* of its input.

Our `size` function does not escape the rule and AGDA's termination checker complains by coloring in red two offending recursive calls. In order to workaround this limitation, we give a new parameter to `size` called `fuel` of type ℕ, on which occurs the structural recursion. When we hit 0 we have not enough gas to proceed and return a dummy value. If given enough fuel the function behaves correctly. Generally it suffices to pick an upper bound for the fuel parameter. Here a good upper bound would be the size of the term...

```
-- ‘‘fuel-extended’’ to pass the termination checker
size : (fuel : ℕ) → Tmᴬ → ℕ
size (suc n) (Vᴬ _) = 1
size (suc n) (fct · arg) = 1 + size n fct + size n arg
size (suc n) (ƛ abs) = 1 + size n (openTm (0 ᴬ) abs)
size (suc n) (Let t abs) = 1 + size n t + size n (openTm (0 ᴬ) abs)
size (suc n) (Vᴺ x) = Name∅-elim x
size 0         _ = 0 -- dummy
```

6.1.5 Free atoms using `openTm`

A more interesting example is to compute the list of free variables, more precisely of free atoms. In order to define it independently of the representation of local name abstractions, we use `openTm`. Since the identity of atoms matters here, we take an atom supply as an argument to open abstractions with atoms that are fresh for the current context. Indeed there is no need to thread the atom supply here since we remove the atom afterward. A `fuel` argument is added to pass the termination checker. These two parameters would benefit from being hidden as a monad, at least for conveniently using and combining these functions. The function `fa` uses the function `rmᴬ` which was defined in section 2.1.1.

```
module FreeAtoms where
  fa : (fuel atomSupply : ℕ) → Tmᴬ → List Atom
  fa (suc n) _ (Vᴺ x)        = Name∅-elim x
  fa (suc n) _ (Vᴬ x)        = [ x ]
  fa (suc n) s (fct · arg) = fa n s fct ++ fa n s arg
  fa (suc n) s (ƛ abs)
    = rmᴬ (s ᴬ) (fa n (1 + s) (openTm (s ᴬ) abs))
  fa (suc n) s (Let t abs)
    = fa n s t ++
      rmᴬ (s ᴬ) (fa n (1 + s) (openTm (s ᴬ) abs))
  fa 0         _ _             = [] -- dummy
```

6.1.6 Common parts of the internal representation

The type Tmᴬ is for locally closed terms. This type is based on a more general type for terms with potential free local names. This type is called PreTm and is indexed by a world. The only implementation specific part is the representation of name abstractions. The type AbsPreTm represents a name abstraction and can be implemented in various ways.

```
AbsPreTm : World → Set

data PreTm α : Set where
  Vᴺ   : (x : Name α)                     → PreTm α
  Vᴬ   : (x : Atom)                       → PreTm α
  _·_  : (t u : PreTm α)                  → PreTm α
  ƛ    : (t : AbsPreTm α)                 → PreTm α
  Let  : (t : PreTm α) (u : AbsPreTm α)   → PreTm α

AbsPreTm = {! implementation specific !}
```

Once the types PreTm and AbsPreTm are defined, defining the locally closed variants amounts to choose the empty world.

```
-- Locally closed terms
Tmᴬ : Set
Tmᴬ = PreTm ∅

-- Locally closed abstracted terms
AbsTm : Set
AbsTm = AbsPreTm ∅
```

6.1.7 Locally Nameless

We can now focus on specific details of local variables. We start with *Locally Nameless* [Charguraud, 2011, Aydemir et al., 2008]. To do so we only have to choose the representation of name abstractions. We make use of the type SynAbsᴰ defined in Figure 6.1:

```
AbsPreTm = SynAbsᴰ PreTm
```

Substituting definitions for AbsPreTm and SynAbsᴰ in the definition of PreTm yields the type Tmᴰ with an extra constructor for atoms:

```
SynAbsᴺ   F α   = ∃[ b ](F (b ◃ α))
SynAbsᴰ   F α   = F (α ↑1)
                ≅ ∃[ β ](α ↪ᴰ β × F β)
SynAbsᴸ   F b α = F (sucᴮ b) (b ◃ α)
SynAbs    F α   = ∀ {β} → α ⊆ β → F β → F β
SynAbs+ᵂ  F α   = ∀ {β} k → (α +ᵂ k) ⊆ β → F β → F β
SynAbsᴺ#  F α   = ∃[ b ](b # α × F (b ◃ α))
                ≅ ∃[ β ](α ↪ᴺ β × F β)
```

Figure 6.1: Various forms of name abstraction

```
-- Once AbsPreTm and SynAbsᴰ are expanded:
data PreTm α : Set where
  Vᴺ   : (x : Name α)                      → PreTm α
  Vᴬ   : (x : Atom)                        → PreTm α
  _·_  : (t u : PreTm α)                   → PreTm α
  ƛ    : (t : PreTm (α ↑1))                → PreTm α
  Let  : (t : PreTm α) (u : PreTm (α ↑1)) → PreTm α
```

We then define a module `TraversePreTm` (code omitted) like in section 5.3.4 to generically derive various operations including the coercion operation, and the substitution operation:

```
-- These two functions are derived from TraversePreTm
coercePreTm : ∀ {α β} → α ⊆ β → PreTm α → PreTm β
substPreTm : ∀ {α β} → (Name α → PreTm β)
                     → PreTm α → PreTm β
```

We can finally define `openSubstTm` and `closeTm`. The function `openSubstTm` is a simple use of substitution on terms using the function `substPreTm`. The argument to `substPreTm` replaces `0` by `t` using the function `exportWith`:

```
openSubstTm : Tmᴬ → AbsTm → Tmᴬ
openSubstTm t = substPreTm (exportWith t (Name-elim ∅+1⊆∅))
```

The function `closeTm` is essentially the combination of a coercion from the empty world to the world `∅ ↑1` and a replace operation of an atom by a name referring to the newly introduced binder at the root. To make this replacing operation we simply carry on a name which we initialize with `0 ᴺ` and increment when going down under the abstractions with `sucᴺ↑`. While we could have fused these operations we did not for the clarity of the code.

```
closeTm : Atom → Tmᴬ → AbsTm
closeTm a t = ! (0 ᴺ) (coerce∅PreTm t) where
  -- ‘‘! y t’’ replaces ‘‘a’’ by ‘‘y’’ in ‘‘t’’
  ! : ∀ {α} → Name α → PreTm α → PreTm α
  ! y (Vᴬ x)      = if x ==ᴬ a then Vᴺ y else Vᴬ x
  ! y (Vᴺ x)      = Vᴺ x
  ! y (t · u)     = ! y t · ! y u
  ! y (ƛ t)       = ƛ (! (sucᴺ↑ y) t)
  ! y (Let t u) = Let (! y t) (! (sucᴺ↑ y) u)
```

6.1.8 Locally Named

Local names bindings can be changed from de Bruijn indices to a nominal style by updating `AbsPreTm`. We use `SynAbsᴺ` from section 3.4 and Figure 6.1:

```
AbsPreTm = SynAbsᴺ PreTm
```

The resulting type `PreTm` is similar to the type `Term` from section 3.4 and to the type `Tmᴬ` from chapter 2. However, name-abstractions are packed into pairs and there is an extra constructor for atoms.

As usual we define a traversal function along the lines of section 3.2 with an extra argument to deal with the constructor `Vᴬ`:

```
TraversePreTm.tr :
  ∀ {E}    (E-app : Applicative E)
    {Env} (trKit : TrKit Env (E ∘ PreTm))
          (trAtm : ∀ {α β} → Env α β → Atom → E (PreTm β))
    {α β} → Env α β → PreTm α → E (PreTm β)
```

From this traversal we obtain various operations including substitution:

```
substPreTm : ∀ {α β} → Supply β → (Name α → PreTm β)
                                → PreTm α → PreTm β
```

From the function `substPreTm` we quickly get the function `openSubstTm`:

```
openSubstTm : Tmᴬ → AbsTm → Tmᴬ
openSubstTm t (_ , u)
  = substPreTm (0 ˢ) (exportWith t Name∅-elim) u
```

The function `closeTm` is a bit more challenging. We define a new traversal kit called `renameKit+⊆` on a new environment type `SubstEnv+⊆`. This environment type packages a renaming environment and an inclusion witness to maintain the fact that we freshened the output term and hence can import anything from the root output world.

```
record SubstEnv+⊆ Res β₀ α β : Set where
  constructor _,_
  field
    ren : SubstEnv Res α β
    pf  : β₀ ⊆ β
  open SubstEnv ren public
```

This traversal kit behaves such as the renaming kit (`renameKit`) on page 65:

```
renameKit+⊆ : ∀ β₀ → TrKit (SubstEnv+⊆ Name β₀) Name
renameKit+⊆ β₀ = mk trName trBinder extEnv
  where
    Env = SubstEnv+⊆ Name β₀
    open SubstEnv+⊆
    extEnv : ∀ {α β} b (Δ : Env α β) → Env (b ◁ α) (seedᴮ Δ ◁ β)
    extEnv b (ren , pf) =
      (TrKit.extEnv renameKit b ren ,
       ⊆-trans pf (SubstEnv.seed⊆ ren))
```

Finally `closeTm` is defined using the traversal function `TraversePreTm.tr`, the kit `renameKit+⊆`, and the special behavior on atoms defined in the local function `trAtm`:

```
closeTm : Atom → Tmᴬ → AbsTm
closeTm a t = 0 ᴮ , tr (renameEnvˢ 1 , ⊆-refl) t where
  β₀ = _

  trAtm : ∀ {α β} → SubstEnv+⊆ Name β₀ α β → Atom → PreTm β
  trAtm Δ a′ = if a ==ᴬ a′
                 then Vᴺ (coerceᴺ (SubstEnv+⊆.pf Δ) (0 ᴺ))
                 else Vᴬ a′

  kit = mapKit id Vᴺ (renameKit+⊆ β₀)
  tr = TraversePreTm.tr id-app kit trAtm
```

While we have a construction for Locally Named terms, this is not yet exactly the approach described by Sato and Pollak [Pollack et al., 2011, Sato and Pollack, 2010]. Indeed we have not yet integrated the height function which yields a different function to close terms. This means that we do not get the nice properties either, for instance our Locally Named representation is not canonical. However the canonical representation is less crucial in our setting since our interface prevents from observing these representation details.

6.1.9 When does this stratified technique pay off?

So far we described these “dynamically stratified representations”, explained how we build them on top of our library, and explained the limitations of an exclusive usage of open/close functions. We claim that this technique can be of great effect when the static discipline of our library becomes too heavy. Contrary to most of the Locally Nameless implementations we argue that the name-abstraction should not be an abstract type. In particular with our library, no safety is lost by keeping name-abstraction concrete since we have precise worlds. Moreover, as we have shown, adding atoms to represent external names is the only change we have to make beforehand to our data type to support this technique.

Generalizing from atoms to arbitrary constants We advise users of our library to have a case for constants in their representation of terms for instance. These “constants” could be numbers, atoms, or a type parameter to be chosen afterward. We illustrate with two situations how this can be used effectively.

In the first situation, the client builds a function `f` which requires an extra environment argument to work on open terms. In the second situation the client is only able to define a function `f-c` for closed terms. One can often reduce the first situation to the second if we have an environment for the empty world. In these situations we end up with a function working only on closed terms. However if we pick the type of constants to be atoms, we have a function `f-lc` working on locally closed terms.

```
-- Terms parameterized by constants
Tm : Set → World → Set

-- We suppose that producing an initial environment
-- for arbitrary worlds is difficult.
Env : World → World → Set
```

```
-- We suppose the environment for empty worlds
-- to be easy to produce.
ϵ : Env ∅ ∅

-- Situation 1: A function on open terms
f : ∀ {α β Cst} → Env α β → Tm Cst α → Tm Cst β

-- Situation 2: Like f but works on closed terms
f-c : ∀ {Cst} → Tm Cst ∅ → Tm Cst ∅
f-c = f ϵ

-- Locally closed terms
LCTm = Tm Atom ∅

-- Like f but on locally closed terms
f-lc : LCTm → LCTm
f-lc = f-c
```

As we can see with the definitions of `f-c` and `f-lc` not much happened. In this example we are indeed supposing that nothing is done on constants and that this is correct to treat them parametrically. In this case what has been done is simply to tell the user of the function to convert all the free variables into atoms before calling the function.

We can spare the conversion to the user and do it ourselves. To do so, we show that given enough traversal functions on terms the two following types are equivalent:

```
∀ {Cst} → Tm Cst ∅ → Tm Cst ∅
 ≅
∀ {α Cst} → Tm Cst α → Tm Cst α
```

We focus on the non-immediate direction, that is going from the first type to the second. In this situation we can even define this transformation as a higher-order function. Here we do not use atoms but keep names, and we preserve the original constants as well. We call the function `f` with the type `Name α ⊎ Cst` for constants, where `Cst` is the type of original constants and α is the world of free variables. Once the constants are preserved using the `bindTm` function, the free variables are turned into constants as well using the function `substTm`. Then the function `f` operates on the locally closed term. This yields a locally closed term as well that we coerce to the world α. Finally we restore the original constants and free variables using the function `bindTm` and the eliminator of sums: `[_,_]`. The following function `transform` is in a parameterized module `M` such that it only depends on our library:

```
-- Hides the environment argument of a given function
-- over terms.
module M
  (Tm : Set → World → Set)
  (cst : ∀ {α Cst} → Cst → Tm Cst α)
  (var : ∀ {α Cst} → Name α → Tm Cst α)
  (substTm : ∀ {α Cst} → (Name α → Tm Cst ∅)
                        → Tm Cst α → Tm Cst ∅)
  (bindTm : ∀ {α A B} → (A → Tm B α) → Tm A α → Tm B α)
  (coerce∅Tm : ∀ {α Cst} → Tm Cst ∅ → Tm Cst α)
  (f : ∀ {Cst} → Tm Cst ∅ → Tm Cst ∅)
 where

 transform : ∀ {α Cst} → Tm Cst α → Tm Cst α
 transform
   = bindTm  [ cst , var ]  -- untag constants and variables
   ∘ coerce∅Tm              -- closed terms inhabit any world
   ∘ f                      -- operate on locally closed terms
   ∘ substTm (cst ∘ inj₂)   -- tag variables as constants by ₂
   ∘ bindTm  (cst ∘ inj₁)   -- tag original constants by ₁
```

We can see open and close functions like a way to move the border between the bound names and free names. When we do not call the open function and directly go under a name abstraction, it is as if we implicitly moved the border. This implicit move is one of the main sources of bugs in programs dealing with names and binders. In our system the border is reflected in the types using the worlds. Hence this distinction incurs no cost at run-time and still prevents mistakes caused by confusing bound and free names.

We have shown how to build dynamically stratified representations on top of our library. We have reused our technique to represent binders and local names, but used an atom type for global names. We have built both a Locally Nameless implementation and a basic Locally Named implementation including the traditional open and close functions. These techniques have shown their advantages as a tool for meta-theory, to reason about programming languages. We have shown the limitations that we think these techniques have for programming. We think these techniques are akin to dynamic typing. Indeed the tagging of names as local or global helps write functions without considering the free variables of the input. We have then shown how we can recover this benefit by having terms parameterized by constants. As soon as constants are handled parametrically we can transform any function over closed terms to a function on open terms. However we recognize these dynamically stratified approaches may be simpler to use than our approach and still provide more safety than the bare approaches.

6.2 de Bruijn levels

Let us briefly mention another representation called "de Bruijn levels". This is another nameless representation different from de Bruijn indices. λ-abstractions are implicitly labeled starting at 0 from the root, and variable occurrences simply refer to these labels. De Bruijn levels are actually very close to a nominal presentation since every reference to a given binder is represented identically. However, binders are not freely chosen, as in nominal style, but dictated by the position of the term starting from the root. We can accommodate this style by indexing terms by the next available binder:

```
data Tmᴸ s α : Set where
  V   : Name α → Tmᴸ s α
  _·_ : (t u : Tmᴸ s α) → Tmᴸ s α
  ƛ   : Tmᴸ (sucᴮ s) (s ◁ α) → Tmᴸ s α
  Let : Tmᴸ s α → Tmᴸ (sucᴮ s) (s ◁ α) → Tmᴸ s α
```

The type Tmᴸ is a variant of our type Tm which is not only indexed by a world but by a binder s as well. This binder s tells how far we are from "the root". This binder is carried down by a constructor such as _·_. At a λ-abstraction, the binder index is used and becomes bound in the subterm; moreover, the index is incremented in the subterm. Most functions work as in nominal style: for instance, the function fv requires no fundamental changes. The extra cost is this new index appearing in the types of terms. The benefit is that given an initial binder index, the representation is canonical.

6.2.1 Term comparison

Comparing terms in de Bruijn level style is no different than in nominal style. A name-comparator is carried down and extended with the same function extendNameCmp. In the end, the resulting function makes no restriction on the seeds of the input terms. This emphasises again that our comparison function makes no attempt to compare binders between the two arguments. Bound variable are compared positionally. Here is the signature for the function cmpTmᴸ:

```
cmpTmᴸ : ∀ {α β s₁ s₂} → |Cmp| Name α β
                       → (Tmᴸ s₁ α → Tmᴸ s₂ β → Bool)
```

However one may want to exploit the canonical structure of our representation for terms. We have already done so for de Bruijn indices with the function `eqTmᴰ` in section 5.3.5. When both terms starts with the same binder index and world, then their types stays in synch. When reaching variable occurrences the equality test on names can be applied [Pouillard, 2011a].

```
_==Tmᴸ_ : ∀ {α s} → Tmᴸ s α → Tmᴸ s α → Bool
V x₁       ==Tmᴸ V x₂       = x₁ ==ᴺ x₂
(t₁ · u₁) ==Tmᴸ (t₂ · u₂) = t₁ ==Tmᴸ t₂ ∧ u₁ ==Tmᴸ u₂
ƛ t₁       ==Tmᴸ ƛ t₂       = t₁ ==Tmᴸ t₂
Let t₁ u₁ ==Tmᴸ Let t₂ u₂ = t₁ ==Tmᴸ t₂ ∧ u₁ ==Tmᴸ u₂
_          ==Tmᴸ _          = false
```

De Bruijn level operations are similar to nominal operations. A first difference is that binding constructs do not have to hold a binder since the binder is fixed by the index. This gives a nameless flavor to this presentation but our typed presentation shows where the binders have gone: in the types.

To illustrate how binders moved out of the terms we can look at snippets of functions with all the arguments made explicit:

```
fᴺ : ∀ {α} → Tm α → ...
fᴺ {α} (ƛ b t) = ...

fᴸ : ∀ {b α} → Tmᴸ b α → ...
fᴸ {b} {α} (ƛ t) = ...
```

The first snippet is in nominal style and the binder `b` is found in the term. The second snippet uses de Bruijn levels and the binder `b` comes as a separate (implicit) argument. In the end we have the same data in both snippets. We can see this as a good news, as it helps recover the information lost by this nameless representation.

For instance functions such as `fv` and `coerceTm` require almost no change to be adapted from nominal to de Bruijn levels.

6.2.2 Kits and traversals

We develop kits and traversals in a way similar to section 3.2. Since our terms are now indexed by a binder as well, environments are indexed by two worlds and two binders.

```
EnvTypeᴸ : Set₁
EnvTypeᴸ = (s₁ s₂ : Binder) (α β : World) → Set
```

Kits are made of only two functions `trNameᴸ` and `extEnvᴸ`. There is no need for an equivalent of the function `trBinder` since binders now appear in indices and can thus be controlled through the environment type.

```
TrNameᴸ : (Env : EnvTypeᴸ) (Res : Binder → World → Set) → Set
TrNameᴸ Env Res = ∀ {s₁ s₂ α β} → Env s₁ s₂ α β
                               → Name α → Res s₂ β

ExtEnvᴸ : (Env : EnvTypeᴸ) → Set
ExtEnvᴸ Env =
  ∀ {s₁ s₂ α β} → Env s₁ s₂ α β
                → Env (sucᴮ s₁) (sucᴮ s₂) (s₁ ◁ α) (s₂ ◁ β)

record TrKitᴸ (Env : EnvTypeᴸ)
              (Res : Binder → World → Set) : Set where
  constructor mk
  field
    trNameᴸ : TrNameᴸ Env Res
    extEnvᴸ : ExtEnvᴸ Env
```

We have several traversal functions making use of a kit. We give only the most general of them. As in section 3.2 we abstract over an applicative functor, and the kit's result type is a term. This one suffices to derive all the functions presented in the following:

```
module TraverseATmᴸ
         {E} (E-app : Applicative E)
         {Envᴸ} (kitᴸ : TrKitᴸ Envᴸ (λ s α → E (Tmᴸ s α)))
 where
  open Applicative E-app
  open TrKitᴸ kitᴸ

  trTmᴸ : ∀ {α β s₁ s₂} → Envᴸ s₁ s₂ α β → Tmᴸ s₁ α → E (Tmᴸ s₂ β)
  trTmᴸ Δ (V x)       = trNameᴸ Δ x
  trTmᴸ Δ (t · u)     = pure _·_ ⊛ trTmᴸ Δ t ⊛ trTmᴸ Δ u
  trTmᴸ Δ (ƛ t)       = pure ƛ   ⊛ trTmᴸ (extEnvᴸ Δ) t
  trTmᴸ Δ (Let t u) = pure Let ⊛ trTmᴸ Δ t ⊛ trTmᴸ (extEnvᴸ Δ) u
```

The equivalent of `SubstEnv` is called `SubstEnvᴸ`. This one does not require a `Supply` field, a fresh-for proof suffices. The `RenameEnvᴸ` is a specialization to functions returning names.

```
record SubstEnvᴸ Res (s₁ s₂ : Binder) (α β : World) : Set where
  constructor _,_
  field
    trNameᴸ : Name α → Res s₂ β
    s₂#β    : s₂ # β

RenameEnvᴸ = SubstEnvᴸ (const Name)

renameKitᴸ : TrKitᴸ RenameEnvᴸ (const Name)
```

6.2.3 Derived functions

Along the lines of section 3.2, the kit `renameKitᴸ` is extended to a kit `renameAKitᴸ` which supports an effectful application. Like before, from these kits we derive several functions. Functions like `renameTmᴸ`, `renameTmᴸA`, `renameTmᴸ?`, `closeTmᴸ`, and `substTmᴸ` are built like their analog from section 3.2. The name supply argument is now just a fresh-for proof since the seed appears in the index. Here are the type signatures for these functions:

```
renameTmᴸ : ∀ {α β s₁ s₂} → s₂ # β → (Name α → Name β)
                                   → Tmᴸ s₁ α → Tmᴸ s₂ β

renameTmᴸA : ∀ {E} (E-app : Applicative E)
               {α β s₁ s₂} → s₂ # β → (Name α → E (Name β))
                                    → Tmᴸ s₁ α → E (Tmᴸ s₂ β)

renameTmᴸ? : ∀ {α β s₁ s₂} → s₂ # β → (Name α →? Name β)
                                    → Tmᴸ s₁ α →? (Tmᴸ s₂ β)

closeTmᴸ? : ∀ {s α} → Tmᴸ s α →? Tmᴸ s ∅

substTmᴸ : ∀ {α β s s₂} → s₂ # β → (Name α → Tmᴸ s₂ β)
                                 → (Tmᴸ s α → Tmᴸ s₂ β)
```

We then lift addition and subtraction from names to terms using the function `renameTmᴸ`. The resulting functions get interesting types. In particular the input seed is disconnected from the input seed and this these functions enables to re-level the terms:

```
protectedAddTmᴸ : ∀ {α s₁ s₂} k → s₂ # (α +ᵂ k)
                              → Tmᴸ s₁ α → Tmᴸ s₂ (α +ᵂ k)
protectedAddTmᴸ k pf = renameTmᴸ pf (addᴺ k)

protectedSubtractTmᴸ : ∀ {α s₁ s₂} k → s₂ # α
                                   → Tmᴸ s₁ (α +ᵂ k) → Tmᴸ s₂ α
protectedSubtractTmᴸ k pf = renameTmᴸ pf (subtractᴺ k)
```

6.2.4 New inclusion rules

In order to tightly integrate de Bruijn levels, and more generally to mix nominal binders and `_+1` on worlds, we introduce two new inclusion rules. These two rules state that `_+1` distributes over `_◁_`:

```
⊆-dist-+1-◁ : ∀ {α b} → (b ◁ α) +1 ⊆ (sucᴮ b) ◁ (α +1)
⊆-dist-◁-+1 : ∀ {α b} → (sucᴮ b) ◁ (α +1) ⊆ (b ◁ α) +1
```

We generalize these two primitive rules to not only `1` but any constant:

```
⊆-dist-+-◁ : ∀{α b} k → (b ◁ α) +ᵂ k ⊆ (b +ᴮ k) ◁ (α +ᵂ k)
⊆-dist-+-◁ zero     = ⊆-refl
⊆-dist-+-◁ (suc n) = ⊆-trans (⊆-cong-+1 (⊆-dist-+-◁ n))
                              (⊆-dist-+1-◁ _)

⊆-dist-◁-+ : ∀{α b} k → (b +ᴮ k) ◁ (α +ᵂ k) ⊆ (b ◁ α) +ᵂ k
⊆-dist-◁-+ zero     = ⊆-refl
⊆-dist-◁-+ (suc n) = ⊆-trans (⊆-dist-◁-+1 _)
                              (⊆-cong-+1 (⊆-dist-◁-+ n))
```

6.2.5 Addition and subtraction kits

The add kit (`Addᴸ.kit`) enables to add a constant to all the names and binders of a given traversable term. The environment of this kit is made of two proofs, one on worlds the other on binders. This environment could hence be removed by an optimizing compiler. Once compiled the resulting function is not hindered by the safe handling of names and binders.

```
module Addᴸ k where
  Envᴸ : EnvTypeᴸ
  Envᴸ b₁ b₂ α β = (α  +ᵂ k) ⊆ β
                 × (b₁ +ᴮ k) ≡ b₂
  trNameᴸ : TrNameᴸ Envᴸ (const Name)
  trNameᴸ (pf , _) = coerceᴺ pf ∘ addᴺ k

  kitᴸ : TrKitᴸ Envᴸ (const Name)
  kitᴸ = mk trNameᴸ {! proof omitted !}
```

We then briefly mention a generalization of the previous kit which allows not only addition but subtraction as well. This kit works similarly and we just show the environment type and the signature of the function on terms:

```
AddSubtractEnvᴸ s₁ s₂ α β = (s₁ +ᴮ k₁) ≡ (s₂ +ᴮ k₂)
                          × (α  +ᵂ k₁) ⊆ (β  +ᵂ k₂)

addSubtractTmᴸ : ∀ k₁ k₂
                   {α β s₁ s₂} → (s₁ +ᴮ k₁) ≡ (s₂ +ᴮ k₂)
                               → (α +ᵂ k₁) ⊆ (β +ᵂ k₂)
                               → Tmᴸ s₁ α → Tmᴸ s₂ β
```

From this general operation addSubtractTmᴸ we derive several operations. We can coerce terms, add a constant, subtract a constant, change the seed index of a closed term, and combinations of those.

```
coerceTmᴸ   : ∀ {s α β} → α ⊆ β → Tmᴸ s α → Tmᴸ s β
addTmᴸ      : ∀ {α s} k → Tmᴸ s α → Tmᴸ (s +ᴮ k) (α +ᵂ k)
subtractTmᴸ : ∀ {α s} k → Tmᴸ (s +ᴮ k) (α +ᵂ k) → Tmᴸ s α
rerootTmᴸ   : ∀ {α} k₁ k₂ → Tmᴸ (k₁ ᴮ) ∅ → Tmᴸ (k₂ ᴮ) α
```

6.2.6 Conversion with nominal terms

Converting de Bruijn level terms to nominal could not be simpler. The code [Pouillard, 2011a] is a simple recursive copy where AGDA can even infer the binders to choose to build the nominal terms. When converting, we are simply moving a static information to a dynamic place. When reaching a binding construct the binder is known to be the index of the term. This binder also a valid binder to be used to build the resulting binding construct in nominal style. Here is the conversion function we obtain:

```
module Tmᴸ⇒Tmᴺ where
  ⟪_⟫ : ∀ {α s} → Tmᴸ s α → Tmᴺ α
  ⟪ V x     ⟫ = V x
  ⟪ ƛ t     ⟫ = ƛ _ ⟪ t ⟫
  ⟪ t · u   ⟫ = ⟪ t ⟫ · ⟪ u ⟫
  ⟪ Let t u ⟫ = ƛ _ ⟪ u ⟫ · ⟪ t ⟫
```

The conversion from nominal to de Bruijn levels is a bit more complicated. Indeed nominal bound variables have to be renamed to follow the discipline of levels. We implement this function by first writing a traverse function from nominal terms to de Bruijn level terms. Then we make use of the renaming kit with little changes. Here are the functions we obtain:

```
renameTmᴸ⇒Tmᴺ : ∀ {α β s} → s # β → (Name α → Name β)
                                     → (Tmᴺ α → Tmᴸ s β)
renameTmᴸ⇒Tmᴺ = {! code omitted !}

convTmᴺ∅⇒Tmᴸ : ∀ {s β} → s # β → Tmᴺ ∅ → Tmᴸ s β
convTmᴺ∅⇒Tmᴸ s#β = renameTmᴸ⇒Tmᴺ s#β Name∅-elim

convTmᴺ∅⇒Tmᴸ∅ : ∀ s → Tmᴺ ∅ → Tmᴸ s ∅
convTmᴺ∅⇒Tmᴸ∅ s = convTmᴺ∅⇒Tmᴸ (s #∅)
```

Conclusion We have shown how de Bruijn levels can be implemented using our library. Our types are now indexed by a world and a binder. This binder is a seed for a binder supply. This seed is to be used for the binder of the next binding construct. For the subterms of binding constructs we increment the seed and extend the world in a nominal style using `_◁_`. We have shown how close this style was from the nominal style and have derived operations on terms by applying techniques developed for the nominal style. This style also reuses tools we built for de Bruijn indices such as additions on worlds and names (`_+ᵂ_` and `_+ᴺ_`). The extra index can be cumbersome to track, but can also help, as in the equality test on terms. We took advantage of the representation to build a faster equality test and a faster addition function.

6.3 Links: Binders as World Relations

So far with our library, name-abstraction induced a particular form of world for the subterm. In nominal style, we use `_◁_` and in de Bruijn indices style we use `_↑1`. These forms of worlds induce the use of dependent types in nominal style and computation on types in de Bruijn indices style. In nominal style, dependent types occur since binders appear in worlds. In de Bruijn computation on the level of worlds (hence types) occur with the use of arbitrary shifts and additions on worlds α `+ᵂ k` and $\alpha \uparrow \ell$. In our work of 2010 [Pouillard and Pottier, 2010], the interface relies only on world-polymorphism and since world are erased it avoids dependent types. After our work of 2010 we put aside the idea of links and have developed the interface presented so far using dependent types leading to a more precise and concrete interface.

In this section we come back to a link based interface. We start by describing an interface for *binding-links* or simply *links*. A link is a type parameterized by two worlds which supports the following three operations: `weaken`, `strengthen?`, and `nameOf`. The first parameter of the link is called the outer world and the second the inner world. A link can be seen as a binder `b` plus a relation between the inner and the outer world. The outer world describes the binders before the introduction of `b` and the inner world describes the binders after its introduction. The first operation moves a name from the outer world to the inner world, it is a weakening. The second operation tries to move a name from the inner world to the outer world but may fail if we try to move the name being bound by the binding-link. The third operation reveals the name being bound, which belongs to the inner world. In Agda we have the following record:

```
record Link (_↪_ : (α β : World) → Set) : Set where
  constructor mk
  field
    weaken      : ∀ {α β} → α ↪ β → Name α → Name β
    strengthen? : ∀ {α β} → α ↪ β → Name β →? Name α
    nameOf      : ∀ {α β} → α ↪ β → Name β
```

6.3.1 Terms and examples

Using links, one can define data types, like our type `Tm` for terms. As usual the interesting cases are the binding constructs. In particular in the construct `Let` we see how the use of inner and outer worlds control the defined scopes.

```
data Tm α : Set where
  V    : (x : Name α) → Tm α
  _·_  : (t u : Tm α) → Tm α
  ƛ    : ∀ {β} (ℓ : α ↪ β) (t : Tm β) → Tm α
  Let  : ∀ {β} (ℓ : α ↪ β) (t : Tm α) (u : Tm β) → Tm α
```

Using only the function `strengthen?` one can build our usual functions `fv` and `rm` to compute the list of free variables and remove a name from a list of names:

```
rm : ∀ {α β} → α ↪ β → List (Name β) → List (Name α)
rm _ []        = []
rm ℓ (x :: xs) with strengthen? ℓ x
...                | nothing = rm ℓ xs
...                | just x′ = x′ :: rm ℓ xs

fv : ∀ {α} → Tm α → List (Name α)
fv (V x)         = [ x ]
fv (t · u)       = fv t ++ fv u
fv (ƛ ℓ t)       = rm ℓ (fv t)
fv (Let ℓ t u) = fv t ++ rm ℓ (fv u)
```

We now define a comparator for terms. It follows precisely what we have done before and here again only the function `strengthen?` is necessary:

```
extCmpName : ∀ {α β α′ β′} → |Cmp| Name α β → α ↪ α′ → β ↪ β′
                           → |Cmp| Name α′ β′
extCmpName f ℓ₁ ℓ₂ x₁ x₂
 with strengthen? ℓ₁ x₁ | strengthen? ℓ₂ x₂
... | just x₁′           | just x₂′ = f x₁′ x₂′
... | nothing            | nothing  = true
... | _                  | _        = false

cmpTm : ∀ {α β} → |Cmp| Name α β → |Cmp| Tm α β
cmpTm f (V x)        (V x′) = f x x′
cmpTm f (t · u)      (t′ · u′) = cmpTm f t t′ ∧ cmpTm f u u′
cmpTm f (ƛ ℓ t)      (ƛ ℓ′ t′) = cmpTm (extCmpName f ℓ ℓ′) t t′
cmpTm f (Let ℓ t u) (Let ℓ′ t′ u′) = cmpTm f t t′
                                    ∧ cmpTm (extCmpName f ℓ ℓ′) u u′
cmpTm _ _ _ = false

_==Tm_ : ∀ {α} → |Cmp| Tm α α
_==Tm_ = cmpTm _==ᴺ_
```

6.3.2 Implementations

An appealing definition for nominal links would to pair a binder `b` and a proof that β is equal to `b ◃` α this is called a weak link in [Pouillard and Pottier, 2010]. However to support the weakening operation a fresh-for proof is required. Hence a nominal link is made of a binder `b`, a proof that `b #` α, and a proof that β is equal to `b ◃` α, this is called a strong link in [Pouillard and Pottier, 2010]. Our implementation reuses the type `Supply` which already packages a binder plus a fresh-for proof. Our nominal implementation is straightforward. The weakening operation uses a coercion. The strengthening operation boils down to our function `exportᴺ?`. The operation `nameOf` returns the underlying binder converted to a name. In AGDA we have the following module:

```
module Nom where
  record _↪_ α β : Set where
    constructor _,_
    field
      supply : Supply α
    open Supply supply
    field
      β-def  : β ≡ seedᴮ ◁ α

  weaken : ∀ {α β} → α ↪ β → Name α → Name β
  weaken (supply , ≡.refl) = coerceᴺ (Supply.seed⊆ supply)

  strengthen? : ∀ {α β} → α ↪ β → Name β →? Name α
  strengthen? (_ , ≡.refl) = exportᴺ?

  nameOf : ∀ {α β} → α ↪ β → Name β
  nameOf ((seedᴮ , _) , ≡.refl) = nameᴮ seedᴮ

  nomLinks : Link _↪_
  nomLinks = mk weaken strengthen? nameOf
```

We now do the same with de Bruijn indices. A link in de Bruijn style is just a proof that the inner world β is equal to the outer world α shifted by one. The weakening is the successor function and the strengthening the predecessor function. The function `nameOf` returns `0`:

```
module DeBruijn where
  _↪_ : (α β : World) → Set
  α ↪ β = β ≡ (α ↑1)

  weaken : ∀ {α β} → α ↪ β → Name α → Name β
  weaken ≡.refl = sucᴺ↑

  strengthen? : ∀ {α β} → α ↪ β → Name β →? Name α
  strengthen? ≡.refl = predᴺ?

  nameOf : ∀ {α β} → α ↪ β → Name β
  nameOf ≡.refl = zeroᴺ

  deBruijnLinks : Link _↪_
  deBruijnLinks = mk weaken strengthen? nameOf
```

6.3.3 Building links and terms

In order to build terms we need to introduce more operations, though. Indeed we have no initial links to use. To this end we introduce a simple interface to build links. This interface provides two operations: `init` and `next`. More precisely these operations work on links where the inner world is made existential. Here is our interface in Agda:

```
record FreshLink {_↪_} (link : Link _↪_) : Set where
  constructor mk
  open Link link

  Fresh : World → Set
  Fresh α = ∃[ β ](α ↪ β)

  field
    -- An infinite ‘‘chain’’ of binding-links
    init    : Fresh ∅
    next    : ∀ {α} (x : Fresh α) → Fresh (proj₁ x)
```

As we see in the type of the operation `next`, the links are chained. Let `fresh` be a value of type `Fresh` α. The outer world of `next fresh` is the inner world of `fresh`. Here is our straightforward nominal implementation of the `FreshLink` interface based on operations on name supplies:

```
module NomFresh where
  init : Fresh ∅
  init = _ , 0 ˢ , ≡.refl

  next : ∀ {α} (x : Fresh α) → Fresh (proj₁ x)
  next (._ , s , ≡.refl) = _ , sucˢ s , ≡.refl

  nomFreshLinks : FreshLink nomLinks
  nomFreshLinks = mk init next
```

The de Bruijn implementation is so trivial that AGDA infers most of the definition by unification:

```
module DeBruijnFresh where
  init : Fresh ∅
  init = _ , ≡.refl

  next : ∀ {α} (x : Fresh α) → Fresh (proj₁ x)
  next _ = _ , ≡.refl

  deBruijnFreshLinks : FreshLink deBruijnLinks
  deBruijnFreshLinks = mk init next
```

Building terms is a matter of allocating binding-links and weakening names such that they are in the expected world. To illustrate the construction of terms here are two examples:

```
idTm : Tm ∅
idTm = ƛ x xᵀᵐ
  where x = proj₂ init
        xᵀᵐ = V (nameOf x)

apTm : Tm ∅
apTm = ƛ x (ƛ y (xᵀᵐ · yᵀᵐ))
  where x = proj₂ init
        y = proj₂ (next init)
        xᵀᵐ = V (weaken y (nameOf x))
        yᵀᵐ = V (nameOf y)
```

6.3.4 Kits and traversals

We now leverage our technique to build kits and traversals. Adapting the work that we did in the nominal style is mostly straightforward. Kits are made of a function to operate on names and a function to operate on links. To operate on a link the function commEnv, given a link and an environment, returns both a new link and a new environment. The type Comm is the most difficult part of these traversals and is already described in our 2010 work [Pouillard and Pottier, 2010]:

```
Comm : ( _⇝_ : (α β : World) → Set) → Set
Comm _⇝_ = ∀ {α β γ} → (α ↪ β × α ⇝ γ)
                     → ∃[ δ ](γ ↪ δ × β ⇝ δ)
```

A traversal kit is then made of a function trName and a function commEnv. As before the kit is parameterized over the type of environments Env and the type of results Res.

```
record TrKit (Env : (α β : World) → Set)
             (Res : World → Set) : Set where
  constructor mk
  field
    trName  : ∀ {α β} → Env α β → Name α → Res β
    commEnv : Comm Env
```

We now build the renaming kit. Again the solution is close to the nominal one even if it works for de Bruijn style links as well. Here the supply is of type Fresh. The function importFun takes two links and lifts a function over names in outer worlds α and β to a function over names in inner worlds γ and δ. The function importFun then form the core part of the commutation function for renaming environments. As with nominal kits the environment is a mapping of names from the input world to the output world. This mapping freshens the bound variables and uses a name supply to do so.

```
importFun : ∀ {α β γ δ} → α ↪ γ
                        → β ↪ δ
                        → (Name α → Name β)
                        → (Name γ → Name δ)
importFun ℓ₁ ℓ₂ f x
 with strengthen? ℓ₁ x
... | nothing = nameOf ℓ₂
... | just x′ = weaken ℓ₂ (f x′)
```

```
RenameEnv : (α β : World) → Set
RenameEnv α β = Fresh β × (Name α → Name β)

renameKit : TrKit RenameEnv Name
renameKit = mk proj₂ comm
  where
    comm : Comm RenameEnv
    comm (x , (_ , y) , f)
      = _ , y , next (_ , y) , importFun x y f
```

We now define the traversal function for our particular type for terms. As usual the environment is carried down the recursive function. The function `trName` is called at variables. The function `commEnv` is used at name-abstractions. Given the link found in the name-abstraction and the current environment, the function `commEnv` returns both a new link to close the resulting name-abstraction and a new environment to continue the traversal on subterms in scope of the name-abstraction:

```
module Tr {Env   : (α β : World) → Set}
          (trKit : TrKit Env Tm)
  where
  open TrKit trKit

  tr : ∀ {α β} → Env α β → Tm α → Tm β
  tr Γ (V x)         = trName Γ x
  tr Γ (t · u)       = tr Γ t · tr Γ u
  tr Γ (ƛ ℓ t)       with commEnv (ℓ , Γ)
  ...                   | _ , ℓ′ , Γ′ = ƛ ℓ′ (tr Γ′ t)
  tr Γ (Let ℓ t u) with commEnv (ℓ , Γ)
  ...                   | _ , ℓ′ , Γ′ = Let ℓ′ (tr Γ t) (tr Γ′ u)
```

We put our traversal function at work by using the renaming kit to build a renaming function. Since our renaming kit is producing names and our traversal function expecting a kit which produces terms we have to lift it. The function `kitVar` is a kit transformer applying the constructor `V` to the result of the input function `trName`. Then the function `renameTm` gives the kit and the initial arguments to the traversal function.

```
kitVar : ∀ {Env} → TrKit Env Name → TrKit Env Tm
kitVar (mk trName commEnv) = mk (λ Γ → V ∘ trName Γ) commEnv

renameTm : ∀ {α β} → Fresh β → (Name α → Name β)
                               → Tm α → Tm β
renameTm fr f = Tr.tr (kitVar renameKit) (fr , f)
```

With the function renameTm we can lift the function weaken from names to terms. In this function the input link serves both as the initial weakening function for free names and as a name supply for the freshening purposes. Here is the function weakenTm:

```
weakenTm : ∀ {α β} → α ↪ β → Tm α → Tm β
weakenTm ℓ t = renameTm (next (_ , ℓ)) (weaken ℓ) t
```

Our last application is a capture avoiding substitution function. To simplify matters we build a substitution kit specialized for our term type. The only important difference with the renaming kit is the use of the function weakenTm instead of weaken. The kit is then given to our traversal function to produce the function substTm. We then specialize our substitution function to replace a name/link by a closed term in a term with just this free variable. Finally we write the function β-red which reduces the β-redex on the root of the term if there is one.

```
SubstEnv : (α β : World) → Set
SubstEnv α β = Fresh β × (Name α → Tm β)

substKit : TrKit SubstEnv Tm
substKit = mk proj₂ comm
  where
    importFunTm : ∀ {α β γ δ} → α ↪ γ
                              → β ↪ δ
                              → (Name α → Tm β)
                              → (Name γ → Tm δ)
    importFunTm ℓ₁ ℓ₂ f x
        with strengthen? ℓ₁ x
    ... | nothing = V (nameOf ℓ₂)
    ... | just x′ = renameTm (next (_ , ℓ₂)) (weaken ℓ₂) (f x′)

    comm : Comm SubstEnv
    comm (x , (_ , y) , f)
      = (_ , y , (next (_ , y) , importFunTm x y f))

substTm : ∀ {α β} → Fresh β → (Name α → Tm β) → Tm α → Tm β
substTm fr f = Tr.tr substKit (fr , f)

_[_:=_] : ∀ {α} → Tm α → ∅ ↪ α → Tm ∅ → Tm ∅
```

```
t [ x ≔ u ] = substTm init (strengthenWith u V x) t

β-red : Tm ∅ → Tm ∅
β-red (ƛ ℓ t · u) = t [ ℓ ≔ u ]
β-red t = t
```

Conclusion By instantiating links to be either nominal links or de Bruijn links we obtain the corresponding implementation of the β-reduction of λ-terms. In contrast to our previous work [Pouillard and Pottier, 2010] we simplified the presentation to only retain one kind of binding-link. To reuse the terminology of [Pouillard and Pottier, 2010] we keep only the strong links. We think the purpose of this binding-link abstraction is to be more focused on a clean and simple interface than a comprehensive and precise interface. Indeed precision has been achieved in our library through binders, world operations, world inclusions, and operations on names. The binding-link abstraction hides details about the representation. While a nominal binder can shadow the previous one, a link, thanks to the fresh-for proof, cannot. While a de Bruijn binder is canonical, a link is not. While we can perform arithmetic with de Bruijn indices in our library, with links we cannot. Yet terms with binding-links can be equipped with kits and traversals. In short the paradigm is still very expressive but less flexible.

6.4 Combining nominal and nameless styles

The support for de Bruijn indices and de Bruijn levels, has been introduced by only adding new items to the interface. Neither the interface, the implementation nor the logical relation need to be altered, only extended.

Here we present one of the multiple combinations that could be made. We have two constructors for λ-abstraction, one in nominal style and one in de Bruijn indices style. Right after, we show the function `fv` which produces no surprises.

```
data Tm α : Set where
  V     : (x : Name α) → Tm α
  ƛᴺ    : ∀ b (t : Tm (b ◃ α)) → Tm α
  ƛᴰ    : (t : Tm (α ↑1)) → Tm α
  _·_   : (t u : Tm α) → Tm α
```

```
fv : ∀ {α} → Tm α → List (Name α)
fv (V x)          = [ x ]
fv (fct · arg)    = fv fct ++ fv arg
fv (ƛᴺ b t)       = rm b (fv t)
fv (ƛᴰ t)         = rm₀ (fv t)
```

We now show two variants of a term which combines those two styles of bindings. One thing to notice is that even if a name is introduced in a nominal style, one has to shift it to cross de Bruijn style binder. The first variant binds the first name in nominal style and the second name in the de Bruijn style. To reach the nominal binder from the inside `sucᴺ↑` is used. The second variant binds the first name in de Bruijn style and the second in nominal style. To cross the nominal binder, the de Bruijn occurrence needs to be coerced showing that two worlds are included:

```
apTm₁ : ∀ {α} → Tm α
apTm₁ = ƛᴺ x (ƛᴰ (V (sucᴺ↑ (nameᴮ x)) · V (0 ᴺ)))
  where x = 42 ᴮ

apTm₂ : Tm ∅
apTm₂ = ƛᴰ (ƛᴺ x (V (coerceᴺ pf (0 ᴺ)) · V (nameᴮ x)))
  where x  = 42 ᴮ
        pf = {! proof omitted !}
```

An attractive feature of this combined style would be to partially convert to de Bruijn indices. Once converted, the canonical structure can be used. Since the conversion to de Bruijn indices requires to maintain a mapping from nominal names to de Bruijn indices we build a more general renaming function. Our renaming function (called `ren`) is similar to `renameTmᴰ` (section 5.3.4) but accepts nominal bindings as well. It is similar to `renameTm` (section 3.2) but does not require a name supply argument to build the result term. This function `ren` hence both converts to de Bruijn indices and applies the supplied function on free variables. To build the function `ren`, the user-supplied function on names must be adapted to cross name-abstractions. We reuse the function `protect↑1` from section 5.3.4 which is adapted when both the source and target abstractions are de Bruijn indices. We build a variant of the function `protect↑1` called `protect◁↑1` which works when the source abstraction is nominal and the target abstraction is de Bruijn.

```
protect◁↑1 : ∀ {α β b} → (Name α → Name β)
                       → (Name (b ◁ α) → Name (β ↑1))
protect◁↑1 f = exportWith (0 ᴺ) (sucᴺ↑ ∘ f)
-- protect◁↑1 f b = 0
-- protect◁↑1 f x = 1 + f x

ren : ∀ {α β} → (Name α → Name β) → Tm α → Tm β
ren f (ƛᴺ b t) = ƛᴰ (ren (protect◁↑1 f) t)
ren f (V x)     = V (f x)
ren f (ƛᴰ t)    = ƛᴰ (ren (protect↑1 f) t)
ren f (t · u)   = ren f t · ren f u
```

Using this renaming function we build a simple normalization function which just supplies the identity function as the renaming:

```
norm : ∀ {α} → Tm α → Tm α
norm = ren id
```

Using this combined approach one may loose the convenience of nominal bindings. To recover this convenience, we build a function called `openTm` such as in section 6.1. The function `openTm` takes a name supply `s` and an open de Bruijn term `t`. The function renames the de Bruijn index zero to the first name of the name supply. We then define a simple function called `nomview` which converts the toplevel λ-abstraction from de Bruijn to nominal. This function `nomview` enables to look at the terms as though there were only nominal λ-abstractions. However using `nomview` and `openTm` incurs the limitations we discussed in section 6.1, namely convenience, performance, and non-structural recursion.

```
openTm : ∀ {α} (s : Supply α) → Tm (α ↑1) → Tm (seedᴮ s ◁ α)
openTm {α} (b , b#) = ren f where
 f : Name (α ↑1) → Name (b ◁ α)
 f = exportWith (nameᴮ _) (coerceᴺ (⊆-# b#) ∘ predᴺ)

nomview : ∀ {α} → Supply α → Tm α → Tm α
nomview s (ƛᴰ t) = ƛᴺ _ (openTm s t)
nomview _ t       = t
```

A tantalizing change would be to include new constructors to represent the open and close operations internally. This would enable a lazy opening and closing for terms. For instance we could try to add these data constructors:

```
Close : ∀ {α} b → Tm (b ◁ α) → Tm (α ↑1)
Open  : ∀ {α b} → b # α → Tm (α ↑1) → Tm (b ◁ α)
```

However, we did not go very far with these constructors. Pattern-matching on these is already limited given the indices of the result type are not invertible. Using explicit type equalities helps to go a bit further but not much:

```
Close : ∀ {α β} (eq : β ≡ α ↑1) b → Tm (b ◁ α) → Tm β
Open  : ∀ {α β b} (eq : β ≡ b ◁ α) → b # α → Tm (α ↑1) → Tm β
```

We have explored one particular combination, namely combining nominal and de Bruijn indices styles. We have shown how to build terms, but also how to get a canonical form using de Bruijn indices. We have also shown how to convert one de Bruijn binding to a nominal one. Exploring other combinations is left as a topic for further research. This may include combinations with de Bruijn levels, higher-order representations (such as our normalization by evaluation function in section 3.4), nominal binders with fresh-for proofs to maintain distinct binders, etc.

6.5 Nominal types

In this section we successively describe various specification languages to describe languages. More concretely a specification language enables to write type definitions and their binding structure. In section 6.5.1, we describe nominal signatures as used in FreshML-Lite [Shinwell et al., 2003]. We first describe signatures with a single sort of atoms and then generalize to multiple sorts. We then focus on languages with *pattern* types such as FreshML and Fresh OCaml in section 6.5.1, Cαml in section 6.5.2 and Binders Unbound in section 6.5.3.

6.5.1 FreshML & Fresh OCaml types

FreshML [Shinwell et al., 2003] is a programming language, designed by Mark R. Shinwell, Andrew M. Pitts and Murdoch J. Gabbay. The language is designed to make programming with binders simple. The language is an ML-like language augmented with types and language constructs to deal with names and binders. The language has a conventional call-by-value operational semantics and a type system preventing to break name abstraction. This means that two α-equivalent terms cannot be distinguished within the language. The language offers an adequate encoding of nominal terms (to be described shortly) and guarantees that the identity of a bound name cannot be observed. However, it does not prevent a newly generated atom from escaping its scope.

Fresh OCaml [Shinwell and Pitts, 2005] is an extension of the OCaml [Leroy et al., 2005] language. This language extension enriches OCaml with a primitive type for names and one type construction for name abstractions. From a language design point of view, Fresh OCaml brings the features of FreshML to OCaml.

FreshML-Lite is a light version of FreshML without pattern types and restricted to a single sort of atoms. Indeed FreshML and Fresh OCaml have a simple notion of *pattern types* that we do not treat here. Pattern types are covered in section 6.5.2 and 6.5.3 about Cαml and Binders Unbound. About multiple sorts we show how to handle those and then go back to a single sort to simplify matters.

Nominal terms are the values described by *nominal signatures*. Here is the definition of a nominal signature as of Definition 2.1 of “Nominal Unification” [Urban et al., 2004]:

“A nominal signature is specified by: a set of sorts of atoms (typical symbol ν); a disjoint set of sorts of data (typical symbol δ); and a set of function symbols (typical symbol `f`), each of which has an arity of the form $\tau \to \delta$. Here τ ranges over (compound) sorts given by the grammar τ ::= ν | δ | 1 | $\tau \times \tau$ | $<\nu>\tau$. Sorts of the form $<\nu>\tau$ classify terms that are binding abstractions of atoms of sort ν over terms of sort τ. We will explain the syntax and properties of such terms in a moment.”

Our goal now is to show how we can describe any nominal signature using our library. We could expect sorts of atoms, sorts of data and compound sorts to be translated directly as types in Agda, and thus to have type `Set` (no index). However this would be incompatible with our technique which indices names (atoms) and terms by worlds. We call *expression types* what corresponds to the compound sorts in nominal signatures. The term *expression* takes all its sense when combined with *pattern types* that we will see later. In Agda we call the type of expression types $\mathbb{E}$.

Nominal signatures on one atom sort We now describe the interface to build and combine expression types and nominal signatures. For simplicity we assume that only one sort of atom is used, that we call ν. The first component is called ν^{e} and corresponds to atoms of sort ν. The second component is called 1^{e} and corresponds to the unit expression type. We can take the product of two expression types with $\_\times^{\mathsf{e}}\_$. Finally the expression type $<\nu>^{\mathsf{e}}\ \tau$ classifies the terms that are name-abstractions of atoms of sort ν over terms of type expression τ. Note that since we constrained the interface to a single sort of atoms, name-abstraction expects only one argument, the expression type. To declare a function symbol we introduce $\_\mapsto^{\mathsf{e}}\_$ which takes two expression types and returns an Agda type (i.e. `Set`). In Agda we have the following declarations for the interface:

```
module MonoSortedNominalSignature where
  𝔼      : Set₁
  νᵉ     : 𝔼
  1ᵉ     : 𝔼
  _×ᵉ_   : 𝔼 → 𝔼 → 𝔼
  <ν>ᵉ_  : 𝔼 → 𝔼
  _↦ᵉ_   : 𝔼 → 𝔼 → Set

  -- Non-essential but used in a later example
  Listᵉ : 𝔼 → 𝔼
```

We now give an example of the use of this interface to build a nominal signature. We use AGDA's data type definitions to handle the introduction of data sorts, function symbols, and the handling of recursion. This example describes a fragment of ML. We recall as a comment how this nominal signature was originally written [Urban et al., 2004].

```
{- A Nominal Signature:
  Example 2.2 from ‘‘Nominal Unification’’

sort of atoms: vid
sort of data:  exp
function symbols:
  vr  : vid → exp
  app : exp × exp → exp
  lam : <vid>exp → exp
  lv  : exp × <vid>exp → exp
  lf  : <vid>((<vid>exp) × exp) → exp
-}

module NominalSignatureExample where
  open MonoSortedNominalSignature

  data Exp : 𝔼 where
    vr  : νᵉ ↦ᵉ Exp
    app : Exp ×ᵉ Exp ↦ᵉ Exp
    lam : <ν>ᵉ Exp ↦ᵉ Exp
    lv  : Exp ×ᵉ <ν>ᵉ Exp ↦ᵉ Exp
    lf  : <ν>ᵉ((<ν>ᵉ Exp) ×ᵉ Exp) ↦ᵉ Exp
```

As we can see, the definition of a nominal signature can be easily done within our system. The resulting `Exp` type is fully usable. As an example we give the definition of the function `fv`.

```
fv : Exp ↦ᵉ Listᵉ νᵉ
fv (vr x)                    = [ x ]
fv (app (t , u))             = fv t ++ fv u
fv (lam (b , t))             = rm b (fv t)
fv (lv (t , b , u))          = fv t ++ rm b (fv u)
fv (lf (bf , (b , t) , u)) = rm bf (rm b (fv t) ++ fv u)
```

This function `fv` already reveals parts of the definitions for our nominal signature combinators that we now present in detail. We have not much other options than using `Name` as the definition for ν^e. This forces $\mathbb{E}$ to be `World → Set`. This last definition can also be read as "expression types are the world-indexed types" which should come as no surprise as it captures the very intuition behind our system. The unit type expression (1^e) ignores the world index and yields AGDA's unit type. The product of expression types ($\_\times^e\_$) distributes the world to the subcomponents. Name abstraction pairs a binder and a sub-expression type whose world is properly extended. Finally, the combinator to declare a function symbol ($\_\mapsto^e\_$) distributes to the underlying arrow type but also quantifies over the world. Here are the definitions in AGDA.

```
module MonoSortedNominalSignature where
  𝔼 : Set₁
  𝔼 = World → Set

  νᵉ : 𝔼
  νᵉ = Name

  1ᵉ : 𝔼
  1ᵉ _ = ⊤ -- ⊤ is Agda's unit type

  _×ᵉ_  : 𝔼 → 𝔼 → 𝔼
  (E₁ ×ᵉ E₂) α = E₁ α ×ᵉ E₂ α

  <ν>ᵉ_ : 𝔼 → 𝔼
  (<ν>ᵉ E) α = ∃[ b ](E (b ◁ α))

  _↦ᵉ_  : 𝔼 → 𝔼 → Set
  E₁ ↦ᵉ E₂ = ∀ {α} → E₁ α → E₂ α
```

One might want to unfold the type of a few symbols we defined. Modulo currying, the symbols `Exp`, `vr`, `lam` and `fv` correspond to `Tm`, `V`, `ƛ` and `fv` from section 2.2:

```
Exp : World → Set
vr  : ∀ {α} → Name α → Exp α
lam : ∀ {α} → ∃[ b ] Exp (b ◁ α) → Exp α
fv  : ∀ {α} → Exp α → List (Name α)
```

Multi-sorted nominal signatures We now enrich our expression types with multiple sorts of atoms. We call `Sort` the type of atom sorts. This type `Sort` is actually part of the definition of a nominal signature and thus is not part of the interface. Actually, the interface is parameterized over the type for atom sorts. We rename the ν^e as `Nameᵉ` in this version. `Nameᵉ` associates an $\mathbb{E}$ to each value of `Sort`. The unit, list and product for expression types are now derived from the more general combinators `Neutralᵉ`, `Neutral1ᵉ`, `Neutral2ᵉ`, etc. These combinators respectively lift any type `Set`, `Set → Set`, `Set → Set → Set` to $\mathbb{E}$, $\mathbb{E} \to \mathbb{E}$, $\mathbb{E} \to \mathbb{E} \to \mathbb{E}$, etc. The expression type $< \nu >^e \tau$ classifies the terms that are name-abstractions of atoms of sort ν over terms of type expression τ. We keep the other combinators unchanged. In AGDA we have the following declarations for the interface:

```
module MultiSortedNominalSignature
        (Sort : Set) (_==_ : (x y : Sort) → Bool) where

  𝔼         : Sort → Set₁
  Nameᵉ     : Sort → 𝔼
  <_>ᵉ_      : Sort → 𝔼 → 𝔼
  _↦ᵉ_      : 𝔼 → 𝔼 → Set
  Neutralᵉ  : Set → 𝔼
  Neutral1ᵉ : (Set → Set) → (𝔼 → 𝔼)
  Neutral2ᵉ : (Set → Set → Set) → (𝔼 → 𝔼 → 𝔼)

  1ᵉ    : 𝔼
  1ᵉ    = Neutralᵉ ⊤

  Listᵉ : 𝔼 → 𝔼
  Listᵉ = Neutral1ᵉ List

  _×ᵉ_  : 𝔼 → 𝔼 → 𝔼
  _×ᵉ_  = Neutral2ᵉ _×_
```

To illustrate this new interface and the use of multiple sorts we present a simple signature for System F where type variables are distinguished from term variables. Notice that apart from the boring definition for `_==_` our embedded language for signatures is lightweight.

```
module NominalSignatureExample where
  data Sort : Set where
    vtm vty : Sort

  _==_ : (x y : Sort) → Bool
  vtm == vtm = true
  vty == vty = true
  vtm == vty = false
  vty == vtm = false

  open MultiSortedNominalSignature nomPa Sort _==_



  data Ty : 𝔼 where
    var : Nameᵉ vty ↦ᵉ Ty
    arr : Ty ×ᵉ Ty ↦ᵉ Ty
    all : < vty >ᵉ Ty ↦ᵉ Ty

  data Tm : 𝔼 where
    vr  : Nameᵉ vtm ↦ᵉ Tm
    app : Tm ×ᵉ Tm ↦ᵉ Tm
    lam : Ty ×ᵉ < vtm >ᵉ Tm ↦ᵉ Tm
    App : Tm ×ᵉ Ty ↦ᵉ Tm
    Lam : < vty >ᵉ Tm ↦ᵉ Tm
```

We show that computing the free type variables in both types and terms can be programmed without any extra difficulty. Here the type system not only prevents from leaking bound names but also prevents from mixing the two sorts of variables.

```
  fvtyTy : Ty ↦ᵉ Listᵉ (Nameᵉ vty)
  fvtyTy (var x) = [ x ]
  fvtyTy (arr (σ , τ)) = fvtyTy σ ++ fvtyTy τ
  fvtyTy (all (b , τ)) = rm b (fvtyTy τ)
```

```
fvtyTm : Tm ↦ᵉ Listᵉ (Nameᵉ vty)
fvtyTm (vr x) = []
  -- [ x ] would not type-check
fvtyTm (app (t , u)) = fvtyTm t ++ fvtyTm u
fvtyTm (lam (τ , b , t)) = fvtyTy τ ++ fvtyTm t
  -- rm b ... would not type-check
fvtyTm (App (t , τ)) = fvtyTm t ++ fvtyTy τ
fvtyTm (Lam (b , t)) = rm b (fvtyTm t)
```

We can now turn to the implementation of the multi-sorted interface. Expression types are no longer indexed by a single world but by a collection of worlds. This collection of worlds is itself indexed by atom sorts, and represented by a function from sorts to worlds. The combinator `Nameᵉ` is the type `Name` in the world given by looking up the world associated to the sort ν. The combinator `<_>ᵉ_` needs to update the collection of worlds at the given atom sort. To do so we define the function `_[_:=_]` to do this update. The combinators `Neutralᵉ`, `Neutral1ᵉ`, and `Neutral2ᵉ` are distributing the index to the subcomponents. Since `λ A → Ix → A` is an applicative functor (often called Reader) for all index types `Ix`, the neutral combinators correspond to `pure`, `liftA`, and `liftA2` of this applicative functor. Here is the Agda implementation.

```
module MultiSortedNominalSignature
         (Sort : Set) (_==_ : (x y : Sort) → Bool) where

  𝔼 : Set₁
  𝔼 = (Sort → World) → Set

  Nameᵉ : Sort → 𝔼
  Nameᵉ ν Γ = Name (Γ ν)

  _[_:=_] : (Sort → World) → Sort → World → (Sort → World)
  (Γ [ ν := α ]) ν′ = if ν == ν′ then α else Γ ν′

  <_>ᵉ_ : Sort → 𝔼 → 𝔼
  (< ν >ᵉ E) Γ = ∃[ b ](E (Γ [ ν := b ◁ Γ ν ]))
  Neutralᵉ : Set → 𝔼
  Neutralᵉ A _ = A

  Neutral1ᵉ : (Set → Set) → (𝔼 → 𝔼)
  Neutral1ᵉ F E Γ = F (E Γ)

  Neutral2ᵉ : (Set → Set → Set) → (𝔼 → 𝔼 → 𝔼)
  Neutral2ᵉ Op E₁ E₂ Γ = Op (E₁ Γ) (E₂ Γ)


  1ᵉ : 𝔼
  1ᵉ = Neutralᵉ ⊤

  Listᵉ : 𝔼 → 𝔼
  Listᵉ = Neutral1ᵉ List

  _×ᵉ_ : 𝔼 → 𝔼 → 𝔼
  _×ᵉ_ = Neutral2ᵉ _×_
```

FreshML types In addition to nominal signatures, FreshML and Fresh OCaml support function types in their signatures. This can be taken into account by adding an arrow combinator. We introduce it in two flavors:

```
_→ᵉ_ : 𝔼 → 𝔼 → 𝔼
(E₁ →ᵉ E₂) α = E₁ α → E₂ α

_⇒ᵉ_ : 𝔼 → 𝔼 → 𝔼
(E₁ ⇒ᵉ E₂) α = ∀ {β} → α ⊆ β → (E₁ →ᵉ E₂) β

⇒-to-→ : ∀ {E₁ E₂} → (E₁ ⇒ᵉ E₂) ↦ᵉ (E₁ →ᵉ E₂)
⇒-to-→ f = f ⊆-refl
```

We introduced the latter when defining a normalization by evaluation algorithm in section 3.4. Moreover the latter version is generally more useful than the former since it can be “imported” by merging the given inclusion witness. Here is how to coerce such a function:

```
Coe : 𝔼 → Set
Coe E = ∀ {α β} → α ⊆ β → E α → E β

coerce-⇒ᵉ : ∀ {E₁ E₂} → Coe (E₁ ⇒ᵉ E₂)
coerce-⇒ᵉ pf f = f ∘ ⊆-trans pf
```

6.5.2 Cαml types

Cαml [Pottier, 2006] is both a tool and a language to describe data type with a binding structure. Cαml has been developed by François Pottier. The tool takes data type definitions in a file and produces an OCaml [Leroy et al., 2005] module with OCaml data types and generated boiler plate code.

The data types language of Cαml extends ML algebraic data types that we find in OCaml with annotations to describe the scoping rules. More precisely, Cαml types extends nominal signatures with a notion of *pattern types*.

Cαml data types are of two kinds: expression and pattern types. The expression kind is made to be used for, well, expression-like types. For instance our type `Tm` belongs to this kind, but type-expressions, statements, processes, etc. are expression types. We reuse the terminology of the previous section and call $\mathbb{E}$ the type of expression types. As before we reuse the following primitive expression types: $\texttt{Name}^{\texttt{e}}$, $\_\mapsto^{\texttt{e}}\_$, $\texttt{Neutral}^{\texttt{e}}$, $\texttt{Neutral1}^{\texttt{e}}$, $\texttt{Neutral2}^{\texttt{e}}$. We reuse also the following derived expression types: $\texttt{1}^{\texttt{e}}$, $\texttt{List}^{\texttt{e}}$, $\_\times^{\texttt{e}}\_$. In order for Cαml to derive functions about these data types, functions are not allowed to appear in signatures. Name-abstractions are made of a *pattern type* and surrounded by $<\_>$. Pattern types are called $\mathbb{P}$ and also support various combinators to be described shortly.

```
-- A pattern type
ℙ    : Set₁

-- Name abstraction
<_> : ℙ → 𝔼
```

Pattern types appear when we start a name-abstraction. Like expression types, pattern types are closed under products, and have the arrow $\_\mapsto^{\texttt{P}}\_$ to declare a data constructor (function symbol). In addition, patterns can be binders, or expression types with *inner* and *outer* annotations. Since pattern types embed expression types, base types can be lifted to patterns as well. Here are the signatures of the various forms of pattern types:

```
module CαmlTypes where
  open MonoSortedNominalSignature public

  Binderᴾ  : ℙ
  _×ᴾ_    : ℙ → ℙ → ℙ
  _↦ᴾ_    : ℙ → ℙ → Set

  Outerᴾ   : 𝔼 → ℙ
  Innerᴾ   : 𝔼 → ℙ

  Neutralᴾ : Set → ℙ
  Neutralᴾ = Outerᴾ ∘ Neutralᵉ
        -- which is identical to
        --   Innerᴾ ∘ Neutralᴾ

  -- List of patterns as special data type
  -- since we could not reuse List here.
  data Listᴾ (P : ℙ) : ℙ where
    []  : 1ᴾ ↦ᴾ Listᴾ P
    ::  : P ×ᴾ Listᴾ P ↦ᴾ Listᴾ P
```

To appear in pattern types, expression types are qualified as either *outer* or *inner*, and thus are respectively in-scope and out-of-scope of the surrounding abstraction. The disposition of binders and embedded expressions does not matter for Cαml. The *outer* expressions are not affected by any binder of the pattern. Similarly all the *inner* expressions of a pattern are put in scope of all the binders introduced by the pattern.

Example To illustrate the use of expression and pattern types, we define our term type in the Cαml style and call it `Exp`. The `lam` case is made of a name-abstraction holding a pattern product with one binder and one inner term. The `let` case is more interesting and makes use of the `Outer`$^{\mathtt{P}}$ combinator to hold the first subterm out of the scope of the binder. As with nominal signatures we are using a `data` declaration to take care of introducing the constructors and dealing with the recursive nature of the type:

```
data Exp : 𝔼 where
  var    : Nameᵉ ↦ᵉ Exp
  app    : Exp ×ᵉ Exp ↦ᵉ Exp
  lam    : < Binderᴾ ×ᴾ Innerᴾ Exp > ↦ᵉ Exp
  let′   : < Binderᴾ ×ᴾ Outerᴾ Exp ×ᴾ Innerᴾ Exp > ↦ᵉ Exp
```

We continue our illustration of Cαml's expressiveness with local blocks of local definitions. To achieve this we simply reuse the declaration for `let′` and wrap the binding with `Listᵖ`:

```
  let⋆ : < Listᵖ (Binderᵖ ×ᵖ Outerᵖ Exp) ×ᵖ Innerᵖ Exp > ↦ᵉ Exp
```

To go from non-recursive blocks to recursive ones, a single step is required. Replacing `Outerᵖ` by the `Innerᵖ` annotation suffices to make all the bound names accessible to the right hand part of this construct.

```
  rec : < Listᵖ (Binderᵖ ×ᵖ Innerᵖ Exp) ×ᵖ Innerᵖ Exp > ↦ᵉ Exp
```

We finally add some pattern-matching facilities to our small language. To do so, we introduce a pattern type named `Pat`. This type supports wildcard patterns, single binders, and pairs of patterns:

```
data Pat : ℙ where
  wildcard : 1ᵖ ↦ᵖ Pat
  binder   : Binderᵖ ↦ᵖ Pat
  pair     : Pat ×ᵖ Pat ↦ᵖ Pat
```

Once we have patterns, we package them as branches. A branch holds a pattern, an optional guard, and a term associated with the pattern:

```
Branch : 𝔼
Branch = < Pat ×ᵖ Innerᵖ (Maybeᵉ Exp) ×ᵖ Innerᵖ Exp >
```

The type `Exp` is then extended with a new `match` construct, which holds a scrutiny term and a list of branches:

```
  match : Exp ×ᵉ Listᵉ Branch ↦ᵉ Exp
```

In our AGDA development we show how to modularly define a free-variables function. To do so we define a combinator to compute the free variables for each form of expression types and pattern types. For pattern types we need no less than three actions to collect the inner free variables, the outer free variables, and to remove pattern binders from a list of free variables. Then at name-abstractions, free variables is the union of outer free variables and inner ones minus those bound by the pattern. In the end, one can build an `fv` function on terms whose body is made of combinators and exactly follow the definition of terms.

Behind the scenes We now turn to the definitions needed to make these combinators work with our library. We focus on a single sort of atoms since we have shown how we can support multiple sorts. Expression types are not changed and directly reused from section 6.5.1.

The Cαml change to expression types is the name-abstraction. However, before discussing the name-abstraction combinator, we need to cover pattern types. Pattern types are indexed by no less than two worlds, and an operator on worlds. Let `P` be a pattern type, let α and β be worlds, let `Op` be an operator on worlds (`World` → `World`), then `P` α β `Op` is a fully applied pattern type of type `Set`. The world α plays the role of the outer world, meaning before binders introduced by the pattern. The world β is the inner world, meaning after all the binders introduced by the pattern. Thus, the worlds α and β are outer and inner worlds with respect to the complete pattern found between name-abstraction brackets (<_>). These two worlds α and β are respectively used by `Outer`^{P} and `Inner`^{P} to pick the world of the expression type. Then the operator explains what the given pattern type changes to a world. If the pattern does not bind anything (as is the case of `Outer`^{P} and `Inner`^{P}), then the operator is equal to the identity. Since a binder pattern binds one binder, the operator is equal to the corresponding binder insertion: `b ◁ ...`. Pattern products distribute α and β to subcomponents. Subcomponents world operation get composed into a single operation to index the product. Finally the abstraction brackets package an operation and a fully applied pattern type whose outer world is the current world, the inner world is computed by applying `Op` to the current world. Here are the definitions in Agda:

```
ℙ : Set₁
ℙ = (α β : World) (Op : World → World) → Set

Binderᴾ : ℙ
Binderᴾ _ _ Op = ∃[ b ](Op ≡ _◁_ b)

Outerᴾ : 𝔼 → ℙ
Outerᴾ E α _ Op = E α × Op ≡ id

Innerᴾ : 𝔼 → ℙ
Innerᴾ E _ β Op = E β × Op ≡ id

data _×ᴾ_ (P₁ P₂ : ℙ) : ℙ where
  _,_ : ∀ {Op₁ Op₂ α β}
        → P₁ α β Op₁
        → P₂ α β Op₂
        → (P₁ ×ᴾ P₂) α β (Op₂ ∘ Op₁)
```

```
<_> : ℙ → 𝔼
< P > α = ∃[ Op ](P α (Op α) Op)

_↦ᴾ_ : ℙ → ℙ → Set
_↦ᴾ_ P₁ P₂ = ∀ {α β Op} → P₁ α β Op → P₂ α β Op
```

In the end, the world operation index of a pattern type could be replaced by a list of binders. However this abstract presentation enables more flexible uses. For instance here is the de Bruijn version of the combinator `Binder`$^{\mathtt{P}}$:

```
Binderᴰᵖ : ℙ
Binderᴰᵖ _ _ Op = Op ≡ _↑1
```

This de Bruijn combinator is useful in two ways. First one can imagine mixing `Binder`$^{\mathtt{P}}$ and `Binder`$^{\mathtt{Dp}}$ as we have done in section 6.4. A second option would be to easily change the implementation from nominal to de Bruijn by just updating a single definition. This also shows that both nominal and de Bruijn are expressive enough to cover Cαml data types.

A remark on linearity We can remark a subtle difference between Cαml's semantics for pattern types and the semantics provided by our definitions. In Cαml if two binders in a same pattern are equal there is no shadowing. Such a pattern could be for instance: `λ (x , x) → ...`. In our setting there is shadowing and thus the order of binders in a pattern is relevant. This order is well-defined and specified on the combinator for products. In a pattern product, all the binders on the left comes before all the binders on the right. Thus those on the right can shadow those on the left. We actually recommend to deal with linearity before going to a well-scoped representation. If non-linear patterns have to be accepted, we recommend to turn the subsequent occurrences into free names and thus keep a linear pattern.

Conclusion In this section we explained how our embedded language to describe nominal signatures can scale to Cαml signatures. This shows that all types that can be defined in Cαml can also be defined in our system. However Cαml also provides generated code to perform usual operations on data structures with names and binders. We do not address this part here but several generic programming techniques could be applied in our situation as well.

6.5.3 Connection with Binders Unbound

Stephanie Weirich, Brent Yorgey, and Tim Sheard develop a library called BINDERS UNBOUND [Weirich et al., 2011]. This library is written in HASKELL and provides type level combinators to express data structures with binders.

Then through a mechanism of "data generic programming", usual functions are generically provided for any combination of types. The main feature of Binders Unbound is to get all of these operations for free right after the type definition. It enables rapid prototyping where you directly focus on the code that matters to your program. The safety features are similar to those in FreshML, Fresh OCaml, and Cαml. Name abstraction cannot be violated but names can escape their scope. However by providing many generic operations the number of bugs the programmer can introduce is reduced. Of course nothing prevents from combining both generic programming and a safer approach were names do not escape their scope. To achieve their goal of generic programming, an embedded language of combinators to define types is developed.

The language of combinators is modeled after expression and pattern types. Given our definitions of section 6.5.1 and section 6.5.2, we can adapt to the new combinators concisely. We focus on a single sort of atoms since we have shown how we can support multiple sorts. The first difference is the combinator `Bind P E` which combines a pattern type `P` and an expression type `E`. The binders in `P` scope over the expression type `E`. This is equivalent to $<$ `P` $\times^{\mathtt{P}}$ `Inner`$^{\mathtt{P}}$ `E` $>$ in Cαml, but the combinator $<\_>$ is not part of Binders Unbound. The combinator `Embed` enables to embed an expression type into a pattern type. The combinator `Embed` is equivalent to `Outer`$^{\mathtt{P}}$ from Cαml. The real new combinators are `Rebind` and `Rec`. `Rebind` $\mathtt{P}_1$ $\mathtt{P}_2$ is a pattern type which binds both $\mathtt{P}_1$ and $\mathtt{P}_2$ like $\mathtt{P}_1$ $\times^{\mathtt{P}}$ $\mathtt{P}_2$ does. However, binders of $\mathtt{P}_1$ also scope over the expressions embedded in $\mathtt{P}_2$. `Rec P` is a pattern types where binders of `P` not only scope externally but scope over expressions embedded in `P`. As in Cαml, functions cannot be used in data types.

We give definitions for `Bind`, `Embed`, `Rebind` and `Rec` using our definitions for Cαml. The second world index (β) of pattern types is not used in Binders Unbound but would enable to combine them with Cαml combinators.

```
Bind : ℙ → 𝔼 → 𝔼
Bind P E α = ∃[ Op ](P α (Op α) Op × E (Op α))
        -- ≅ < P ×ᴾ Innerᴾ P E > α

Embed : 𝔼 → ℙ
Embed = Outerᴾ
```

```
data Rebind (P₁ P₂ : ℙ) : ℙ where
  _,_ : ∀ {Op₁ Op₂ α β}
        → P₁ α β Op₁
        → P₂ (Op₁ α) β Op₂
        → Rebind P₁ P₂ α β (Op₂ ∘ Op₁)

Rec : ℙ → ℙ
Rec P α β Op = P (Op α) β Op
```

Open/close based interface We have shown how the types provided by the library Binders Unbound can be expressed within our system. What about the operations? Their library generically provides the usual operations such as computing free variables, comparing two terms up to α-equivalence, substitutions and performing a renaming. However in order to let the user program the remaining specific operations, the library had to commit to a particular implementation or interface to program with binders. They chose to provide an interface with a default implementation using Locally Nameless [Charguraud, 2011, Aydemir et al., 2008]. We discussed at length in section 6.1 the limitations of an interface based on open and close (here called unbind and bind respectively). In particular when the type of abstractions is kept abstract the convenience is reduced when pattern matching, performance is hurt due to unwanted traversals, and structural recursion is no longer possible.

Telescopes Here is an example of the usefulness of the combinator `Rebind`, namely telescopes. Telescopes are widely used in dependently typed programming languages. A telescope is a sequence of typed bindings. Let Γ be a meta-variable for such a telescope defined as `(A : Set) (x : A)`. Telescopes can be used to represent an environment in a typing judgement as in $\Gamma \vdash$ `x : A`. They can be used to represent a sequence of arguments types as in $\Gamma \rightarrow$ `A`. They can also be used to represent a sequence of typed arguments as in $\lambda\ \Gamma \rightarrow$ `x`. While all these construction could be made by nesting `Bind`, this would tie the telescope to a particular usage. If we want the telescope to be a value on its own, its needs to be a pattern type. Hence the use of `Rebind` instead of `Bind`.

To illustrate `Rebind` and telescopes here are two mutually defined data types. The type `Exp` represents expressions in a small dependently typed language where types and expressions are fused in the same syntactic class. Expressions have variables, dependent function types, λ-abstractions, function applications, and a sort called `set`. The type `Tele` represents nested sequences of bindings associated with an expression, namely telescopes:

```
mutual
  data Exp : 𝔼 where
    V    : Nameᵉ ↦ᵉ Exp
    Π    : Bind Tele Exp ↦ᵉ Exp
    ƛ    : Bind Tele Exp ↦ᵉ Exp
    _·_  : Exp ×ᵉ Listᵉ Exp ↦ᵉ Exp
    set  : 1ᵉ ↦ᵉ Exp

  data Tele : ℙ where
    []  : 1ᵖ ↦ᵖ Tele
    ::  : Rebind (Binderᵖ ×ᵖ Embed Exp) Tele ↦ᵖ Tele
```

We show how to compute the free variables for expressions, lists of expressions, and telescopes. To do so we have to lift the function `rm` to telescopes as well:

```
mutual
  fv : Exp ↦ᵉ Listᵉ Nameᵉ
  fv (V x) = [ x ]
  fv (Π (bind Γ t)) = fvTele Γ ++ rmTele Γ (fv t)
  fv (ƛ (bind Γ t)) = fvTele Γ ++ rmTele Γ (fv t)
  fv (_·_ (t , us)) = fv t ++ fvL us
  fv (set _) = []

  fvL : Listᵉ Exp ↦ᵉ Listᵉ Nameᵉ
  fvL [] = []
  fvL (t :: ts) = fv t ++ fvL ts

  fvTele : ∀ {α β Op} → Tele α β Op → List (Name α)
  fvTele ([] _) = []
  fvTele (:: ((binder b , embed τ) , Γ)) = fv τ ++ rm b (fvTele Γ)

  rmTele : ∀ {α β Op} → Tele α β Op
                     → List (Name (Op α)) → List (Name α)
  rmTele ([] (neutral _)) = id
  rmTele (:: ((binder b , embed _) , Γ)) = rm b ∘ rmTele Γ
```

Conclusion In this section on nominal types we have shown the expressiveness of our system to define various families of data types. Through a lightweight embedded language we can support nominal signatures with multiple sorts and pattern types as in FreshML, Cαml and Binders Unbound. Hence worlds are at lest as expressive as these languages. Using our embedded language to define data types is kept very light since the resulting data types are not much different from those written directly. However the expressiveness of worlds is also a weakness. Indeed generic programming as done in BINDERS UNBOUND is greatly simplified by having a language of combinators instead of our flexible use of worlds. To this end we advocate to have both flexible worlds with a mechanized notion of α-equivalence and combinators to get generic programming. We have not studied enough how to integrate data type generic programming since there are plenty of ways to build universes in AGDA and meta-programming tools are promising but still too young.

6.6 Related work

The difficulty of programming with, or reasoning about, names, binders, and α-equivalence has been known for a long time. It has recently received a lot of attention, due in part to the POPLMARK challenge [Aydemir et al., 2005]. Despite this attention, the problem is still largely unsolved: according to Guillemette and Monnier, for instance, "none of the existing representations of bindings is suitable" [Guillemette and Monnier, 2008].

In the following, we review several systems that are intended to facilitate the manipulation of names and binders. This review is not exhaustive: we focus on relatively recent related work.

In particular we have already covered nominal systems such as FRESHML [Shinwell et al., 2003], FRESH OCAML [Shinwell and Pitts, 2005], Cαml [Pottier, 2006]; de Bruijn indices [de Bruijn, 1972]; systems based on well-scoped de Bruijn indices [Altenkirch, 1993, McBride and McKinna, 2004, Bellegarde and Hook, 1994, Bird and Paterson, 1999, Altenkirch and Reus, 1999]; dynamically stratified representations such as Locally Nameless [Charguraud, 2011, Aydemir et al., 2008] and Locally Named [Pollack et al., 2011, Sato and Pollack, 2010]; and BINDERS UNBOUND [Weirich et al., 2011] which is internally based on Locally Nameless.

FreshML and Pure FreshML Pure FRESHML [Pottier, 2007] is built on top of FRESHML. The semantics of FRESHML dictates that pattern matching against a name abstraction silently replaces the bound atom with a fresh atom. FRESHML guarantees that "name abstractions cannot be broken". However, FRESHML is unsafe: it is possible for a name to escape its scope. Put another way, FRESHML is impure: name generation is an observable side effect.

Pure FRESHML imposes additional static proof obligations, which ensure that freshly created atoms do not escape their scope, and correspond to Pitts' *freshness condition for binders* [2006]. Because these proof obligations are expressed in a specialized logic, they can be discharged automatically.

Because it is safe, Pure FreshML can be implemented either using atoms (such as the original FreshML) or using de Bruijn indices. This is an implementation choice, which the programmer need not know about.

In contrast with Pure FreshML, the approach proposed in the present work does require certain runtime checks: the operation `exportTm` fails if its argument contains a free name that cannot be exported. We claim that often those checks are required and thus would be discharged by Pure FreshML only with equivalent checks. However in some cases, some invariants of the code would make these checks redundant. Pure FreshML might able to show that we can export without any check.

In Pure FreshML, name abstraction is a primitive notion, and the fact that deconstructing an abstraction automatically freshens the bound atom is used to guarantee that all terms effectively live in a single world. In our work, in contrast, name abstraction is explained in terms of more basic notions, and it is possible to deconstruct a name abstraction without substituting a fresh name for the bound name. This leads to a finer-grained understanding of binding, and, in some cases, to greater runtime efficiency.

Nominal System T Pitts' Nominal System T [2010] follows the tradition of FreshML and guarantees that name abstractions cannot be violated. In order to ensure that names do not escape their scope, Pitts uses a dynamic technique: the ν construct, which can be applied to any term, turns every name occurrence that is about to escape its scope into a harmless "anonymous name", known as `anon` (or `new`). One potential advantage of this approach is that the application of ν to a term can be lazily evaluated, whereas our `exportTm` operation requires eagerly traversing the entire term.

We can emulate this behavior in our system by adding a special value to the type for names to represent `anon`. We experimented with this option in our development [Pouillard, 2011a] by defining a type `Name?` α as `Maybe` (`Name` α) and `anon` as `nothing`. Most operations can be easily lifted. In particular `export`$^{\mathtt{N}}$`?` gets the type `Name?` (`b` $\triangleleft$ α) $\rightarrow$ `Name?` α and so becomes "total". Then, exporting terms gets the type `Tm` (`b` $\triangleleft$ α) $\rightarrow$ `Tm` α. However there is no direct equivalent for $\neg$`Name`$\emptyset$, since the empty world is indeed inhabited by `anon`. Instead we have that only `anon` inhabits the empty world.

One may wonder if the use of the `Maybe` type could be avoided? Indeed we tried another option in our development [Pouillard, 2011a] where the type `Name?` α is equal to `Name` (α $\uparrow$`1`) and `anon` is equal to `zero`$^{\mathtt{N}}$. We also lifted most of the operations from `Name` to `Name?`. We also gain "total" functions to export names and terms which would allow for a lazy exporting at the price of delaying "errors".

A separate "data layer" Some systems are said to have a "data layer" separated from the "computation layer" for types. The data layer is generally a LF signature of a Logical Framework [Harper et al., 1993] (LF for short). This generally means that data types are only defined in the data layer. In the data layer there is an arrow type for functions but it should not be confused with computational functions that we expect from a programming language. A type constructor from the data layer can then be injected in the computation layer. The injection from the data to computation layer is a place where these systems differ. The data and computation layers being separated, this implies that we cannot mix first-class computational functions and data types.

In the case of a simply typed LF, we can map the data layer signatures to nominal signatures (section 6.5.1). To do so, we build an injection called $\lceil \_ \rceil$ from LF types to atom sorts. Usually this means an atom sort per data sort. Then the translation called $[\![\_]\!]$ of the signature is straightforward. The arrow case $[\![\texttt{S} \texttt{ -> } \texttt{T}]\!]$ is translated to $< \lceil \texttt{S} \rceil >^{\mathsf{e}} [\![\texttt{T}]\!]$. Finally for each data type δ, a data constructor is added for variables. This constructor has type: $\texttt{V}\delta \texttt{ : Name}^{\mathsf{e}} \lceil \delta \rceil \mapsto^{\mathsf{e}} \delta$. This translation can be used as an intuition of what it means to lift a type from the data layer to the computation layer and also as an initial step to support LF signatures in our system. In particular Licata and Harper follow a similar translation but using a single constructor for variables shared among the complete signature. Here is an example:

```
{- LF signature for System F:
Ty  : type.
arr : Ty -> Ty -> Ty.
all : (Ty -> Ty) -> Ty.

Tm  : type.
app : Tm -> Tm -> Tm.
lam : Ty -> (Tm -> Tm) -> Tm.
App : Tm -> Ty -> Tm.
Lam : (Ty -> Tm) -> Tm.
-}
```

Only two type expressions appear on the left of an arrow. The injection $\lceil \_ \rceil$ is thus defined as follow:

```
⌈ Ty ⌉ = vty
⌈ Tm ⌉ = vtm
```

Applying the previously mentioned translation we obtain the following data types:

```
data Ty : 𝔼 where
  VTy : Nameᵉ vty ↦ᵉ Ty
  arr : Ty ×ᵉ Ty ↦ᵉ Ty
  all : < vty >ᵉ Ty ↦ᵉ Ty

data Tm : 𝔼 where
  VTm : Nameᵉ vtm ↦ᵉ Tm
  app : Tm ×ᵉ Tm ↦ᵉ Tm
  lam : Ty ×ᵉ < vtm >ᵉ Tm ↦ᵉ Tm
  App : Tm ×ᵉ Ty ↦ᵉ Tm
  Lam : < vty >ᵉ Tm ↦ᵉ Tm
```

Elphin, Delphin, Beluga, Beluga$^\mu$ Elphin [Schürmann et al., 2005], Delphin [Poswolsky and Schürmann, 2008, 2009], and Beluga [Pientka, 2008] are closely related to one another in several ways. They aim at building programming languages where data types are built from LF types and terms.

Elphin only has a limited way to include types from the data layer in the computation layer. Both Delphin and Beluga have contexts but their handling is different. Delphin deals with a "current context" while Beluga has explicit contexts. Delphin allows incremental changes of the current context using ν whereas Beluga combines contexts and LF objects at the frontier of data and computation.

Beluga$^\mu$ [Cave and Pientka, 2012] extends Beluga with richer computational types. In Beluga$^\mu$ we can define data types in the computation layer as well. These data types can be recursive and indexed by LF objects of the data layer. While substitution comes for free at the data layer, substitution has to be user defined on data types from the computation layer. We think more connections between our system and Beluga$^\mu$ are to be found. We also notice their use of our system to prove properties of their language in Coq proof assistant. They used a simplified version of our 2010 paper [Pouillard and Pottier, 2010] which corresponds to the simplifications we have made in section 6.3.

At the data level, Elphin, Delphin, Beluga provide substitution and higher-order matching as primitive operations. This ambitious approach can eliminate some boilerplate code, at the cost of a complex meta-theory. By contrast, the meta-theory of our proposal is very simple, as it only extends an existing logical relations argument with a few new primitive types and operations. We argue that some of the boilerplate can still be eliminated via generic programming as this is done by Licata and Harper but also in Binders Unbound.

Licata and Harper's system Licata and Harper [2009] aim to provide substitution for free when possible; and they expose the use of well-scoped de Bruijn indices to the programmer, whereas we offer a choice between nominal and de Bruijn-based representation techniques.

This said, there are numerous similarities between the two systems. Both keep track of the context, or world, within which each name makes sense. Both offer flexible ways of parameterizing or quantifying types over worlds. Both offer ways of moving data from one world to another: Licata and Harper's weakening and strengthening respectively correspond to our import (coerce) and export operations. Both systems support first-class computational functions. Not all functions can be imported or exported, but some can: for instance, in both systems, the example of normalization by evaluation (section 3.4), which requires importing a function into a larger world, is made type-correct by planning ahead and making this function polymorphic with respect to an arbitrary world extension.

Distinctions One traditionally distinguishes several broad approaches to the problem of names and binders, which employ seemingly different tools, namely: atoms and atom abstractions; well-scoped de Bruijn indices; higher-order abstract syntax. We believe that this distinction can be superficial. In fact, our work presents strong connections with all three schools of thought. Perhaps more important are the following questions:

Can α-equivalent terms be distinguished? Except for bare de Bruijn indices and plain nominal terms all the systems we have presented prevent α-equivalent terms to be distinguished. These systems offer an adequate encoding of "nominal" terms and guarantees that the identity of a bound name cannot be observed.

What hygiene properties are enforced by the system? The base hygiene property is prevent from α-equivalent terms to be distinguished. We now focus on more hygiene properties. In FreshML, Fresh OCaml, Cαml, Locally Nameless, Locally Named, and Binders Unbound generating names is an observable effect. Said differently functions can turn α-equivalent inputs into not α-equivalent outputs. Nominal System T repairs this problem by dynamically detecting an atom that attempts to escape and by turning it into a harmless "anonymous atom". Systems based on well-scoped de Bruijn indices enforce the invariant that every name is within range, that is, every name refers to some binding site. However, this alone does not imply that indices are correctly adjusted where needed, or that comparisons between names are allowed only when they make sense. In systems based on higher-order abstract syntax, and in Pure FreshML, name manipulation is hygienic by design: this is built in the syntax and semantics of the programming language.

In the present document, hygiene is built in at a very low level. We provide a small number of abstract types, such as `Binder` and `Name`, together with a small number of operations. These operations are restricted on purpose: for instance, comparing two binders for equality is disallowed; comparing two names for equality is permitted only if they inhabit a common world. Through logical relations and parametricity, we are able to explore the consequences of these restrictions and to find out which hygiene properties can be expected of a well-typed program. In particular, we guarantee that two α-equivalent terms cannot be distinguished and that functions turn α-equivalent inputs into α-equivalent outputs. These results hold because *our* definition of α-equivalence *is* the logical relation, and that every well-typed program inhabits this relation. Moreover, our definition of α-equivalence corresponds to the "standard" notion of α-equivalence, at least for all nominal signatures like the type `Tm`.

Is there a type for names that can be freely used in types? The answer to this question is positive for most "first order" systems we have studied: de Bruijn indices, well-scoped de Bruijn indices, Locally Nameless, Locally Named, FreshML, Fresh OCaml, Cαml, Pure FreshML, Nominal System T, Binders Unbound, and in the present work.

In systems based on LF and higher-order abstract syntax the type for names is not freely used. Instead, names inhabit all types issued from the data layer.

Does the system have different data and computation layers? Elphin, Delphin, Beluga, Beluga$^{\mu}$ clearly have two different layers. Since Cαml does not support function while OCaml does we conclude that Cαml has two different layers as well. In de Bruijn indices, well-scoped de Bruijn indices, Locally Nameless, Locally Named, FreshML, Fresh OCaml, Pure FreshML, Nominal System T, Binders Unbound and the present work there is no separation. However very few systems handle the mix of computational functions and data types with binders. Systems based on a generator or generic programming such as most Locally Nameless and Locally Named implementations, Cαml, and Binders Unbound disallow types with computational functions. In Pure FreshML, there is no support for first class functions at all. In de Bruijn indices, well-scoped de Bruijn indices, FreshML and Fresh OCaml data and computation can be mixed since little operations are generically provided. However in FreshML and Fresh OCaml the swap operation *is* generically provided and supports functions. In Licata and Harper's work [2009] data and computation can be mixed. However they are both part of a "universe" which is then interpreted into computation types of the host language.

Is there a set of types where substitution can be defined? If so what are the restrictions? Some systems let the user program substitution functions. Some systems try to provide them for free either at all types or have restrictions. When a system has a specific "data layer" for types, the goal is to have substitution for free on these types once injected into the computation layer.

In Elphin, Delphin, and Beluga, substitution for data types comes for free but the data layer is restricted and cannot embed computational functions.

In Beluga$^{\mu}$ there is substitution has to be defined for data types in the computation layer and comes for free on the data layer.

In FreshML, Fresh OCaml, Nominal System T, Pure FreshML this is up to the user to program substitution functions. In Cαml (systems based on nominal signatures), Locally Nameless, Locally Named programmers have to define substitution functions but generators may be provided for usual cases. In Binders Unbound, substitution is generically provided for a family of types not including functions. In Licata and Harper's work their is a generic definition for the substitution which is conditioned by criterion on the type.

In our system, substitution has to be programmed by the user and no particular set of types is prescribed for this task. Then as we have shown earlier a wide range of signatures can be embedded in our systems and generic programming could be applied on those.

How does the system keep track of the context or world in which a name makes sense? In Pure FreshML and in Nominal System T, there is effectively just one world, within which every name makes sense; static or dynamic checks guarantee that no confusion can arise. In Elphin, Delphin, or in Licata and Harper's system, the meaning of types is relative to a "current context", and a number of modalities are provided to discard the current context, extend it with one new name, etc. In Beluga and Beluga$^{\mu}$, contexts are explicit: a data layer type, once annotated with a context, becomes a computation layer type. In the present work, worlds are explicit, and are built into algebraic data type definitions by the programmer. There is no notion of a "current world": multiple worlds can co-exist, and mechanisms are offered for transporting names (or whole data structures) from one world to another.

None of de Bruijn indices, Locally Nameless, Locally Named, FreshML, Fresh OCaml, Cαml, or Binders Unbound keep track of the context in which a name makes sense.

Which high-level operations are involved in the semantics of the programming language? The semantics of the "nominal" systems (such as FreshML, Fresh OCaml, Cαml, Pure FreshML, Nominal System T)

involves renaming. In Locally Nameless, Locally Named, and Binders Unbound the open/close operations are costly and necessary to use to go under binders. The semantics of Elphin, Delphin, Beluga, and Beluga$^{\mu}$ involve higher-order matching. In the present work, as well as in Licata and Harper's work, no costly operations are built into the semantics; high-level operations, such as our import and export operations, are programmed explicitly or obtained via generic programming.

Moving across representations It is arguably desirable to be able to offer several choices of representation within a single system, and to be able to migrate from one representation to another. Our system supports multiple representation styles, and combinations thereof (section 6.4). Furthermore, our implementation of normalization by evaluation (section 3.4) illustrates how to move back and forth between "syntactic" name abstractions and "semantic" name abstractions in the style of higher-order abstract syntax.

Robert Atkey and co-authors [Atkey, 2009, Atkey et al., 2009] investigate how to move back and forth between higher-order abstract syntax and de Bruijn indices. The translation out of higher-order abstract syntax produces well-scoped de Bruijn indices, but the proof of this fact is meta-theoretic. Atkey uses Kripke logical relations to argue that the current world at the time of application of a certain function must be larger than the world at the time of construction of this function. This seems somewhat related with our use of bounded polymorphism in the definition of semantic name abstractions (section 3.4). An exact connection remains to be investigated.

Chapter 7

Conclusion

7.1 Contributions

Our contributions are practical and fundamental. On the practical side, we have built a library (in AGDA) for programming with names and binders. The library comprises a safe programming interface and a sound implementation, which has been mechanically proved against a strong specification. The interface supports different styles for binders: nominal, de Bruijn indices, de Bruijn levels, and combinations of them.

Our interface is not only safe but expressive as well. Explicit worlds enable both multiple kinds of variables and precise control over the scoping rules. Name abstraction is *not* a primitive type nor an abstract type but concretely built out of more atomic types. While expressive and safe, the system is economical on the number of introduced types.

On a fundamental side, we re-use the logical relations defined for AGDA in [Bernardy et al., 2010], and build a logical relation argument which provides a strong specification of α-equivalence by building a unified model. This is the very same model that is used for all styles: nominal, de Bruijn indices, de Bruijn levels and the combinations. The framework of logical relations provides a systematic way of defining α-equivalence on complex data types. Moreover free-theorems arise for world-polymorphic functions defined by the user, yielding interesting properties about user code for free.

This work also establishes numerous connections between various systems of the three schools of thought (nominal, de Bruijn, HOAS). We built our library with a “nominal” insight and lightly integrated de Bruijn indices based on well-scoped de Bruijn indices approach. We then built a normalization by evaluation algorithm which shows how “syntactic” and “semantic” abstraction can be combined.

This unified approach does not stop here. De Bruijn levels, Locally Nameless, Locally Named, and combinations of them can be safely built on top of our abstract interface and programmed precisely. Through the use of generic traversals and traversal kits, we can share most of the usual definitions we have on terms.

Moreover our types are very flexible, as shown by the embedding of nominal signatures, Cαml types, and BINDERS UNBOUND types. Not only flexible, our types automatically come with a definition for α-equivalence which is guaranteed to be preserved by the well-typed programs using our interface.

7.2 Future work

As future work we would like to make our library lighter to use. To do so we would like to infer most of the inclusion witnesses needed to move from a world to another.

While hand-written generic traversals already avoid code duplication, we would like to generically define this traversal for a fixed universe of types.

Studying how this system could support the presence of side-effects would be an interesting direction as well. This could be done either by considering a language design with our features and side effects or by studying how our types compose with particular effects as monads.

Finally we would like to explore how convenient this technique is, to not only program but reason about programs. Since the library is based on abstract types, this would require exporting properties about the functions of the interface. We can notice that this direction is promising since a simplified version of our approach has been successfully used by Cave and Pientka to prove properties of Beluga$^{\mu}$.

We would also like to better study the connection with systems based on LF and higher-order abstract syntax. A direction could be to combine the universe from Licata and Harper but using our library instead of well-scoped de Bruijn indices.

We also need to deal with a proper extraction of our library. We want to erase: worlds, membership proofs, world inclusion witnesses, etc. A real implementation should rely on machine integers; and useless traversal functions (like `coerceTm`) should be optimized away.

We focused on names and binders to guarantee that the binding structure of well-scoped terms is preserved by well-typed programs. However safe and expressive meta-programming requires that not only scopes but types are preserved. We worked on initial ideas in this direction but we left this to future work.

Bibliography

Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993. URL http://www.cs.nott.ac.uk/~txa/publ/tlca93.pdf.

Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. URL http://www.cs.nott.ac.uk/~txa/publ/csl99.pdf.

Robert Atkey. Syntax for free: representing syntax with binding using parametricity. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, July 2009.

Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In *Haskell symposium*, pages 37–48, September 2009. URL http://personal.cis.strath.ac.uk/~raa/unembedding.pdf.

Brian Aydemir, Arthur Chargraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 3–15, January 2008. URL http://www.cis.upenn.edu/~bcpierce/papers/binders.pdf.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science. Springer, August 2005.

Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.*, 23(2-3): 287–311, 1994.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 345–356, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863592. URL http://doi.acm.org/10.1145/1863543.1863592.

Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999. URL http://dx.doi.org/10.1017/S0956796899003366.

Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 2012. URL http://www.cs.mcgill.ca/~bpientka/papers/recursive-types.pdf. To appear.

Arthur Charguraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011. ISSN 0168-7433. URL http://www.chargueraud.org/research/2009/ln/main.pdf. 10.1007/s10817-011-9225-2.

Nicolaas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'c: a language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 131–144, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237765. URL http://doi.acm.org/10.1145/237721.237765.

Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In *ACM International Conference on Functional Programming (ICFP)*, 2008. URL http://www-etud.iro.umontreal.ca/~guillelj/icfp08.pdf.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, pages 194–204, 1993.

Graham Hutton. *Programming in Haskell.* Cambridge University Press, January 2007.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. *The Objective Caml system*, October 2005. URL http://caml.inria.fr/.

Daniel R. Licata and Robert Harper. A universe of binding and computation. In *ACM International Conference on Functional Programming (ICFP)*, pages 123–134, September 2009. URL http://www.cs.cmu.edu/~drl/pubs/lh09unibind/lh09unibind.pdf.

Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14:69–111, January 2004. ISSN 0956-7968. doi: 10.1017/S0956796803004829. URL http://www.cs.ru.nl/~james/RESEARCH/view-final2004.pdf.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 371–382, January 2008. URL http://www.cs.mcgill.ca/~bpientka/papers/hoasfun-short.pdf.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

Andrew M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006. URL http://www.cl.cam.ac.uk/~amp12/papers/alpsri/alpsri.pdf.

Andrew M. Pitts. Nominal System T. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 159–170, January 2010. URL http://www.cl.cam.ac.uk/~amp12/papers/nomst/nomst-popl.pdf.

Randy Pollack, Masahiko Sato, and Wilmer Ricciotti. A canonical locally named representation of binding. 2011. URL http://homepages.inf.ed.ac.uk/rpollack/export/PollackSatoRicciottiJAR.pdf. To appear.

Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 93–107. Springer, March 2008.

Adam Poswolsky and Carsten Schürmann. System description: Delphin – A functional programming language for deductive systems. *Electronic*

Notes in Theoretical Computer Science, 228:113–120, 2009. URL http://www.itu.dk/~carsten/papers/lfmtp-08.pdf.

François Pottier. An overview of Cαml. In *ACM Workshop on ML*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 27–52, March 2006. URL http://gallium.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf.

François Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 356–365, July 2007. URL http://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml.pdf.

Nicolas Pouillard. Nompa (Agda code), 2011a. http://tiny.nicolaspouillard.fr/NomPa.agda.

Nicolas Pouillard. Nameless, painless. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11. ACM, September 2011b. URL http://nicolaspouillard.fr/publis/nameless-painless.pdf.

Nicolas Pouillard and François Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 217–228, New York, NY, USA, September 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863575. URL http://gallium.inria.fr/~fpottier/publis/pouillard-pottier-fresh-look.pdf.

John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983. URL ftp://ftp.cs.cmu.edu/user/jcr/typesabpara.pdf.

Masahiko Sato and Randy Pollack. External and internal syntax of the lambda-calculus. *J. Symb. Comput.*, 45(5):598–616, 2010. URL http://homepages.inf.ed.ac.uk/rpollack/export/SatoPollack09.pdf.

Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The ∇-calculus: Functional programming with higher-order encodings. In *International Conference on Typed Lambda Calculi and Applications (TLCA)*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, April 2005. URL http://www.itu.dk/~carsten/papers/nabla.pdf.

Mark R. Shinwell and Andrew M. Pitts. Fresh Objective Caml user manual. Technical Report 621, University of Cambridge, February 2005. URL http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-621.pdf.

Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *ACM International Conference on Functional Programming (ICFP)*, pages 263–274, August 2003. URL `http://www.cl.cam.ac.uk/~amp12/papers/frepbm/frepbm.pdf`.

Walid Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute, November 1999. URL `http://www.cs.rice.edu/~taha/publications/thesis/thesis.pdf`.

Christian Urban, Andrew Pitts, and Murdoch Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004. URL `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=D14EB8679102796E01FBD20257BB5F34?doi=10.1.1.3.7275&rep=rep1&type=pdf`.

Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989. URL `http://homepages.inf.ed.ac.uk/wadler/papers/free/free.ps.gz`.

Stephanie Weirich, Brent Yorgey, and Tim Sheard. Binders unbound. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11. ACM, September 2011. URL `http://www.cis.upenn.edu/~byorgey/papers/binders-unbound.pdf`.

List of Figures

Index