Complexity and Implementation of Nominal Algorithms

Christophe François Olivier Calvès

Department of Computer Science

King's College London

A thesis submitted for the degree of

Doctor of Philosophy

I would like to dedicate this thesis to my loving grandparents and
the teachers who believed in me ...

Acknowledgements

I want to express the deepest gratitude to my supervisor, Maribel Fernández, for all the support, the constant help and the dedication she put into teaching me how to be a good researcher. I wish to be one day, for my students, as nice as she has been for me.

I wish to thank particularly Olivier Danvy for having invited me at BRICS in Aarhus. It was a wonderful and very rewarding experience, as much on the research level as the personal level. Without him the part of this thesis about continuations would have been much lighter. I thank Mathieu Boespflug and Zoé Drey for the great time it was searching and living with them in Aarhus. I wish the stay could have continued more, but reaching the end of the PhD, time was delimited.

I thank James Cheney, Murdoch J. Gabbay, Andrzej Filinski, Aad Mathijssen, Dale Miller, Andrew Pitts and Christian Urban for the precious discussions we had. The experience their shared with me helped directing my efforts in the right direction.

I thank Agi Kurucz for having been my second supervisor and the nice time it was to be her teaching assistant.

I thank the *Engineering and Physical Sciences Research Council* for the grant that enabled me to live three years studying a subject I like.

Last but not least, I thank particularly *Clémentine* for her *p*atience, l*o*ve and the *u*nlimit*e*d motiv*a*tion she give*s* to me.

And many thanks to *Maribel*, *Zoé* and *Clémentine* for the proofreading.

Abstract

Nominal terms generalise first-order terms by introducing abstraction and name swapping constructs. It aims to be a natural and simple way to represent systems that involve binders. α-equivalence, unification, matching and rewriting can be generalized to nominal terms.

This thesis shows that first-order and nominal theories are strongly related. In particular, we extend first-order unification algorithms, such as Patterson and Wegman's linear one and Martelli and Montanari's almost linear one, to nominal unification. We then present a modular algorithm to solve nominal α-equivalence and matching problems. The approach relies on name management abstraction and stream manipulation. We show how choosing an appropriate name management strategy leads, in some cases, to linearity in time. The matching algorithm is then adapted to nominal rewriting, improving in some cases the time complexity.

Handling name management, freshness constraints and specific reduction strategies simultaneously makes nominal algorithms intricate. This thesis presents a high level and modular approach to the implementation of nominal algorithms. Each nominal aspect is implemented as a monadic layer. We show how the CPS hierarchy can be used to separate the signature of each layer from its actual implementation thus enabling to use different interpretation functions. We also present a rewriting framework based on the previous nominal rewriting algorithm and a term navigation monadic layer based on a zipper.

For each of the algorithms, an implementation is available either in the Haskell language or in the Objective Caml.

Contents

Acknowledgements **3**

Abstract **4**

1 Introduction **10**

I Generalities 14

2 Presentation of Haskell and Continuations **15**

2.1 Haskell Syntax 16

2.2 Monads 19

2.3 Continuations 19

3 First-Class Monadic Signatures **21**

3.1 A simple example 21

3.2 A specialized continuation monad 24

3.3 A more complex example 25

3.4 General morphism 27

3.5 Monadic Tower and the CPS Hierarchy 30

3.5.1 Lifting operations 32

3.5.2 Well behaved Error Monad 33

3.6 Conclusion 35

4 Nominal Theory **36**

4.1 Graphs and Trees 36

4.2 Nominal Terms 37

4.2.1 Standard Nominal Term 40
4.2.2 Compact Syntax 41
4.2.3 Reduced Compact Syntax 43
4.2.4 From Standard Terms To Compact Terms 45
4.2.5 From Nominal to First-Order 48
4.3 Nominal Problems 49
4.3.1 Morphisms 54
4.4 Complexity and Implementation of Permutations and Sets 57
4.4.1 Implementation 58
4.4.2 Optimisation 60
4.4.3 Abstraction of sets and permutations 61
4.4.4 Complexity 62

II Unification 63

5 Nominal Unification Terms and Problems as Directed Acyclic Graphs 65

6 Quadratic Unification Algorithm 69
6.1 Nominal Unification as a Relation on Term Nodes 69
6.2 A Quadratic Nominal Unification Algorithm 73

7 A Simple and Efficient Nominal Unification Algorithm 80
7.1 A Nominal Union-Find Algorithm 81
7.2 An Almost Quadratic Nominal Unification Algorithm *(AQNU)* . . 86
7.2.1 Complexity 91

8 Nominal Unification via Graph Rewriting 93
8.1 A polynomial Algorithm via graph rewriting 96
8.1.1 A graph rewriting algorithm to solve $\approx_\alpha$ constraints 96
8.1.1.1 Normalisation of the graph 96
8.1.1.2 Properties 99
8.1.1.3 $\approx_\alpha$-rules 106
8.1.1.4 Computing permutations: Neutralisation 111

8.1.1.5 Putting all together . . . 113
8.1.1.6 An upper bound on the number of iterations . . . 114
8.1.1.7 An upper bound on the number of normalisation steps . . . 116
8.1.1.8 Cost of the rules . . . 118
8.1.2 Freshness constraints . . . 121
8.1.3 Cost . . . 123
8.1.4 Total cost in time for the unification algorithm . . . 124

III α-equivalence, Matching and Rewriting 127

9 A modular algorithm 129

9.1 Environments . . . 129
9.2 Core algorithm . . . 131
9.3 Checking the validity of α-equivalence constraints . . . 138
9.4 Solving Matching Problems . . . 139

10 Implementation 141

10.1 The Freshness Layer . . . 142
10.2 Only one environment: The environment layer . . . 143
10.3 Combining the phases: Streams . . . 146
10.4 α-equivalence and Matching . . . 149
10.5 Complexity . . . 151
10.6 Benchmarks . . . 155

11 A Simple Nominal Rewriting Framework 156

11.1 Nominal rewriting . . . 156
11.2 A nominal rewriting algorithm: two improvements on the matching algorithm . . . 158
11.3 Complexity of nominal rewriting . . . 161
11.3.0.1 Special cases . . . 162
11.4 The Zipper Layer . . . 162

IV Conclusions 164

Related Works 165

Future Works 168

Conclusion 170

Bibliography 176

A Haskell Code 177
- A.1 Base definitions . . . 177
- A.2 Fist-Class Monadic Signatures . . . 180
- A.3 Error Handling Layer . . . 185
- A.4 Store Layer . . . 188
- A.5 Nominal Terms . . . 193
- A.6 Sets and Permutation Layer . . . 198
- A.7 Nominal Union-Find Algorithm . . . 210
- A.8 Unification . . . 217
- A.9 Environment Layer . . . 228
- A.10 α-equivalence and Matching Algorithms . . . 235
- A.11 Zipper Layer . . . 246

B Objective CAML Code 250
- B.1 Quadratic Nominal Unification . . . 250

List of Algorithms

1 First-Order Linear Unification *(FLU): Part 1* 74
2 First-Order Linear Unification *(FLU): Part 2* 75
3 Quadratic Nominal Unification *(QNU): Part 1* 76
4 Quadratic Nominal Unification *(QNU): Part 2* 77
5 Quadratic Nominal Unification *(QNU): Part 3* 78

6 Tarjan's union-find algorithm *(TUF)* 82
7 Nominal Union-Find Algorithm *(NUF): Part 1* 84
8 Nominal Union-Find Algorithm *(NUF): Part 2* 85
9 Unification and normalizing functions 88
10 Equation storing function 90

11 Interpretation of Freshness when Solving an α-equivalence or matching problem 143
12 Phase 2 and 4 145
13 Streamed version of Phase 1 147
14 Streamed version of Phase 3 148
15 The core algorithm 149
16 α-equivalence Interpretation of *core* 150
17 Matching Interpretation of *core* 150

18 Interpretation of Freshness for matching terms in context 160

Chapter 1

Introduction

This thesis is the result of my three years of research as a PhD student at the *Department of Computer Science* at *King's College London*. The subject was to study nominal algorithms, both from a theoretical point of view by studying their complexity and from a practical one by studying their implementations.

The notion of a binder is ubiquitous in computer science. Programs, logic formulas, and process calculi are some examples of systems that involve binding. Program transformations and optimisations, for instance, are defined as operations on programs, and therefore work uniformly on α-equivalence classes. To formally define a transformation rule acting on programs, we need to be able to distinguish between free and bound variables, and between meta-variables of the transformation rule and variables of the object language. We also need to be able to test for α-equivalence, and we need a notion of matching that takes into account α-equivalence.

In *The Implementation of Functional Programming Languages* [41] *Peyton-Jones* shows how manipulating terms up to α-equivalence is a big issue in the implementation of functional programming language.

Nominal techniques were introduced to represent in a simple and natural way systems that include binders [24, 42, 48]. The nominal approach to the representation of systems with binders is characterised by the distinction, at the syntactical level, between *atoms* (or object-level variables), which can be abstracted (we use the notation $[a]t$, where a is an atom and t is a term), and *meta-variables* (or just variables), which behave like first-order variables but may be

decorated with atom permutations. Permutations are generated using swappings (e.g., $(a\ b) \cdot t$ means swap a and b everywhere in t). For instance, $(a\ b)\cdot\lambda[a]a = \lambda[b]b$, and $(a\ b)\cdot\lambda[a]X = \lambda[b](a\ b)\cdot X$ (we will introduce the notation formally in chapter 4.2). As shown in this example, permutations suspend on variables. The idea is that when a substitution is applied to X in $(a\ b)\cdot X$, the permutation will be applied to the term that instantiates X.

As atoms are just names, we want to identify $[a]a$ and $[b]b$ and even $[a]X$ and $[b]X$ with some conditions. Permutations of atoms are one of the main ingredients in the definition of α-equivalence for nominal terms, they are the translation tables from one term to another. Obviously such translations are subject to conditions, $[a]X$ and $[b]X$ can only be identified if neither a nor b are unabstracted in X. The constant management of such translations and conditions is what makes finding and implementing good nominal algorithms not an easy task.

As we will see in this thesis, we managed to show that unification of nominal terms can be done in quadratic time and that α-equivalence and matching problems can even sometimes be done in linear time and space. Implementing these algorithms was not an easy task because of the amount of subtle atom management details. We found an elegant and easy way of programming with nominal terms, handling as many details as possible automatically and efficiently.

Such an implementation was possible thanks to the visit the author made at the *BRICS* [1] PhD school with Olivier Danvy. This is where the second big aspect of this thesis was developed: continuations. Chapter 3 is a result of the thought developed during the visit. The whole *Haskell* implementation is based on it. It enabled the implementation to be clearer, more flexible and manageable.

The main contributions of this thesis are:

- A study of relations between nominal theory and first-order theory. We present a morphism between nominal theory and first-order theory. This morphism enabled us to extend algorithms and properties in first-order theory to nominal theory.

- A quadratic nominal unification algorithm, the most efficient unification algorithm yet.

- An almost quadratic nominal unification algorithm: while being almost as efficient as the one above, it is also very easy to use, to maintain and to extend, which makes it often the best choice in practice.

- A polynomial nominal algorithm based on graph reduction: it gives interesting results and techniques for rewriting up to α-equivalence.

- An efficient, very modular algorithm for nominal α-equivalence and matching: its complexity is linear on ground problems, log-linear on linear problems, which is very frequent in practice.

- An implementation of monadic classes which makes them first-class citizen of the language and makes the instantiation explicit: it makes monadic towers much simple by giving them a concrete representation. It can also bring monadic classes to some languages which do not natively support them.

- A nominal framework made of monadic layers handling different nominal aspects (permutations, sets of atoms, freshness contexts, environments, ...) to ease the development of program manipulating nominal terms. With is programming with nominal terms is almost as easy as programming with first-order terms.

The thesis is structured as follows:

The first part presents general results. Chapter 2 presents briefly the *Haskell* programming, language, its syntax, monads in computer science and continuations. Chapter 3 introduces nominal theory: nominal terms with several syntaxes and how they are related, nominal problems and implementation of permutations and sets.

The second part is about solving nominal unification problems. Chapter 5 shows how to represent nominal terms as directed acyclic graphs. This representation is used to present, in chapter 6 a quadratic algorithm for nominal unification based on a linear first-order algorithm. Chapter 7 introduces a more practical and still efficient algorithm. Finally chapter 8 presents a polynomial algorithm based on graph rewriting.

The third part is about solving α-equivalence and matching problems. Chapter 9 presents, in an abstract way, a very modular algorithm to solve α-equivalence and matching problems. Chapter 10 describes its implementation and proves its complexity. Finally, chapter 11 presents a simple nominal rewriting algorithm based on this algorithm and a zipper.

The last part of the thesis concludes and shows the actual implementation of the algorithms written in *Haskell* and *Objective Caml*. The code has been put in the appendices for two reasons: because it is the most precise description of the algorithms presented in the thesis, and as a real world example of the implementation techniques developed in the thesis. The code can also be found at [5].

Part I

Generalities

Chapter 2

Presentation of Haskell and Continuations

Most of the implementations presented in this thesis are written in the *Haskell* programming language. This section presents briefly the main aspects and the syntax of the language to enable the reader to understand the examples. For a detailed description, please refer to the *Haskell 98 Revised Report* [3]. The implementations use many extensions not present in the report but described in the *GHC* documentation [2]. To learn about the language, the *Haskell* web site [4] is a good entry point.

Haskell is defined in the *Haskell 98 Revised Report* as

> Haskell is a general purpose, **purely functional** programming language incorporating many recent innovations in programming language design. Haskell provides **higher-order functions**, **non-strict semantics**, **static polymorphic typing**, user-defined algebraic datatypes, **pattern-matching**, list comprehensions, a module system, a **monadic** I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages.

"Purely functional" means that the value of any expression in *Haskell* only depends on the expression itself and not the context. As a consequence, values are

persistent and when applied to the same arguments, a function always returns the same value. "Non-strict semantics" means that it is possible that an expression returns a value (not $\perp$) even if some of its subexpressions evaluate to $\perp$. For example, let us consider the following *Haskell* code:

```
let loop () = loop ()
    f    x  = 0
in f (loop ())
```

where loop is a recursive function taking () as argument and looping on itself and f is a function ignoring its argument and returning 0. loop () never terminates but f (loop ()) still returns 0.

2.1 Haskell Syntax

Definitions Variables are defined by:

```
var = expression
```

where var is the variable name and expression an expression. Variables are immutable in *Haskell*, so variables can be seen as identifiers for their expressions. In *Haskell*, definitions are recursive, which means that var can appear in expression. For example, the following code defines an infinite list of 0:

```
list = 0 : list
```

Functions In *Haskell*, a function is an expression of the form:

```
\arg1 ... argn -> expression
```

where $n > 0$ and arg1 to argn are the n arguments of the function. The character \ represents λ in λ-calculus. Because functions are expressions, they can be assigned to variables as any expression. For example the function f defined above could have been defined by:

```
f = \x -> 0
```

For the sake of simplicity, a simpler way to define functions is:

```
f arg1 ... argn = expression
```

where f is the name of the function.

If an argument is not used in the body of the function, it can be replaced by _.

Type declarations In *Haskell*, declaring that the variable variable has the type **type** is written variable :: **type** . For example, list and f could have been defined by:

```
1 list :: [Integer]
2 list = 1 : list
3
4 f :: a -> Integer
5 f x = 0
```

which means list is a list of integers and f is a function whose argument can be of any type and returns an integer.

Pattern-Matching *Haskell* supports pattern-matching. It can be used in several ways:

- by a **case** expression:

  ```
  1     case expression where
  2       pattern1 -> expression1
  3       ...
  4       patternn -> expressionn
  ```

  where expression, expression1, . . . , expressionn are expressions and pattern1 to patternn are patterns. The value of the **case** expression is the value of the expression of the first pattern (from pattern1 to patternn) to match the value of expression.

- in a function definition:

  ```
  1     f pattern1 = expression1
  2     ...
  3     f patternn = expressionn
  ```

  which corresponds to

```
f x = case x of
        pattern1  -> expression1
        ...
        patternn  -> expressionn
```

User-defined algebraic datatypes Users can defined their own algebraic datatypes. The syntax is:

```
data DataTypeName args =
    Constructor1 args1
  | ...
  | Constructorp argsn
```

where DataTypeName is the name of the data type, args a list of type variables, Constructor1 to Constructorn n constructors where args1 to argsn are the lists of the types of the arguments of their corresponding constructor.

For example, the following code defines a parametrized binary tree data type:

```
data Tree leaftype =
    Empty
  | Leaf leaftype
  | Node (Tree leaftype) (Tree leaftype)
```

Type classes Type classes are the *Haskell* way to implement ad-hoc polymorphism. Let us consider the equality function (==), which takes two arguments and returns whether or not they are equal. In *Haskell* this function has the type (==) :: (**Eq** a) =>a -> a -> **Bool**. (**Eq** a) => is a type constraint, it means that the type a has to be an instance of the class **Eq** which is defined as:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

which means that to be an instance of **Eq**, a has to provide the two functions (==) and (/=), both of type a -> a -> **Bool**.

2.2 Monads

Monads are a general way to represent hidden computation [36]. In *Haskell*, among other things, they provide a simple way to implement side effects. A monad in *Haskell* is a datatype which is an instance of the **Monad** class (simplified version):

```
1 class Monad m where
2   return :: a                  -> m a
3   (>>=)  :: m a -> (a -> m b) -> m b
```

The idea is that you can inject a value into a monad (**return**) and bind a monad with a function ((>>=)) but you cannot, in general, retrieve a value from a monad. For example, the datatype of lists is a monad:

```
1 instance Monad [] where
2   return x = [x]
3   m >>= f  = concat (map f m)
```

return x returns the list containing only the element x and (>>=) maps every element of the list m by the function f of type a -> [b] and returns the list of all the generated elements.

Input/Output operations are implemented in *Haskell* via an abstract monad **IO**. On the contrary to most programming languages, input/output operations have to be executed only in the **IO** monad and nowhere else.

2.3 Continuations

Continuations were introduced by Landin [32] to be the functional counterpart of jumps in programming languages (GOTO). They have known since a big development. In this thesis we will not use any of the many control operators that have been introduced since Landin's J operator but plain and simple code in **continuation passing style** in the form of a slight variation of the usual *Haskell* continuation monad (simplified version):

```
1 data Cont r a = Cont (a -> r) -> r
```

which means that a code in continuation passing style which takes a function as argument called the **continuation** and returns value. The continuation represent ”the rest of the computation“. For example, the following code just gives the value 5 to the rest of the computation:

```
give5 = Cont (\k -> k 5)
```

The following code, if the second argument is 0 does not execute the rest of the computation, instead it returns directly a string saying there was an error:

```
div x y = Cont $ \k -> if y == 0
                          then "division by zero"
                          else k (x / y)
```

Code in continuation passing style can decide whether or not they want to continue the computation and even how many times to do it. For a more detailed description of continuation passing style (CPS) and how to how to transform any code into CPS, please refer to [15, 16, 46].

Chapter 3

First-Class Monadic Signatures

A **monadic signature** is an abstract monad along with a set of operations on it. The usual way of doing it is, in *Haskell*, to define a class. In *Objective CAML*, we would declare a *module signature* even if it is not the same thing. Classes are not first-class citizens in either *Haskell* or *Objective CAML*. We present a way to do so in *Haskell*. The implementation relies on the use of existential quantification in datatypes, rank n polymorphism and higher kinded type constructors. It might be possible to do the same in *Objective Caml*, but not as easy as with *Haskell*. That is why we present it in *Haskell*.

The idea is similar to **defunctionalization** [17, 46]. The abstract monad operations are represented by a datatype, the abstract monad being implemented as a concrete continuation monad. As in defunctionalization, concrete instantiations of the monad are implemented by an interpretation function.

3.1 A simple example

In this example we consider a class with logging capabilities. It contains a single operator write taking the string to log and returning the number of char written. The usual way of doing it in Haskell is the following:

```
1 class (Monad m) => Log m where
2   write :: String -> m Int
```

The constraint (**Monad** m) => ensures that m is also a monad. Let us consider a piece of code using write

```haskell
test :: (Log m) => m Int
test = do x <- write "First  write\n"
          y <- write "Second write\n"
          return (x + y)
```

Indeed the constraint (Log m) => ensure that m is an instance of Log. Two instances of Log could be: the state monad with a string state (State **String**), write s would append s to the state and the IO monad (**IO**), write s would print s on the standard output:

```haskell
instance Log (State String) where
  write s = modify (\l -> l ++ s) >> return (length s)

instance Log IO where
  write s = putStr s >> return (length s)
```

and the monad m in test can be instantiated as any of them:

```haskell
testState :: State String Int
testState = test

testIO :: IO Int
testIO = test
```

Let us define writeState and writeIO as

```haskell
writeState :: String -> State String Int
writeState s = write

writeIO    :: String -> IO Int
writeIO    s = write
```

We present now another way to implement and instantiate the class Log without class constraints, based on continuations. The Log class is implemented as the datatype :

```haskell
data DLog r = DWrite String (Int -> r)
```

and the abstract previous monad m satisfying the class Log is implemented as the concrete continuation monad.

The idea is, when an operation of the class is called, to stop the computation and return the constructor corresponding to the invoked call, with its arguments and the captured continuation. DWrite would be something like

```
1 dwrite string = Cont $ \continuation ->
2                     DWrite string continuation
```

Unfortunately this is not that simple. Returning DWrite string continuation requires that r is of type DLog r. Furthermore, the rest of the computation may contains other calls, so other DWrite string' continuation' and finally, if it exists, the final value of the computation. To make all this work finely we need to introduce a data type for such a sequence of calls possibly ended by a final value:

```
1 data SCall datatype finalvalue =
2      End finalvalue
3    | Call (datatype (SCall datatype finalvalue))
```

when datatype is DLog, it gives (in pseudo Haskell) :

```
1 data SCall DLog finalvalue =
2       End finalvalue
3     | Call (DWrite String (Int -> SCall DLog finalvalue))
```

what is what we need.

dwrite is then implemented as

```
1 dwrite :: String -> Cont (SCall Log finalvalue) Int
2 dwrite string = Cont (\cont -> Call (DWrite string cont))
```

Instantiation is made by an interpretation function for DWrite string continuation. This function actually writes string and resume the continuation with the returning value. For example, the interpretation functions for the two previous instances (State **String** and **IO**) are:

```
1 interpretState  :: DLog a -> State String a
2 interpretState  (DWrite s k) = do  r <- writeState s
3                                    return (k r)
4
5 interpretIO     :: DLog a -> IO a
6 interpretIO     (DWrite s k) = do  r <- writeIO s
7                                    return (k r)
```

the final building block is an instantiation function. It takes two arguments: an interpretation function (such as interpretState and interpretIO) and an abstract computation and interprets the stream of calls with the interpretation function, thus instantiating the computation:

```
instantiate :: (Monad m) =>
     (theclass (SCall theclass a) -> m (SCall theclass a))
     -> Cont (SCall theclass a) a
     -> m a

instantiate interpret t = aux (runCont t End)
  where aux (End    r) = return r
        aux (Call   c) = do n <- interpret c
                            aux n
```

For example test is simply

```
dtest :: Cont (SCall DLog e) Int
dtest = do x <- dwrite "First  write\n"
           y <- dwrite "Second write\n"
           return (x + y)
```

and instantiated with

```
dtestState :: State String a
dtestState = instantiate interpretState dtest

dtestIO :: IO a
dtestIO    = instantiate interpretIO dtest
```

3.2 A specialized continuation monad

We introduce a variation of the continuation monad, specialized to our needs. We have seen in the previous section that the continuation was of type

$$a \to \text{SCall datatype final}$$

final can also be universally quantified.

$$\forall final \quad (a \to SCall\ datatype\ final) \to SCall\ datatype\ final$$

which gives in *Haskell*

```
newtype SCont datatype a = SCont { unSCont ::
    forall r . (a -> SCall datatype r) -> SCall datatype r) }
```

SCont theclass is a monad:

```
instance Monad (SCont theclass) where
  return a           = SCont $ \k -> k a
  (SCont m) >>= f    = SCont $ \k -> m (\x -> unSCont (f x) k)
```

and

$$\begin{array}{lcr} return\ a >>= f & \Leftrightarrow & f a \\ m >>= return & \Leftrightarrow & m \\ (m >>= f) >>= g & \Leftrightarrow m >>= (\lambda x \to f x >>= g) & \end{array}$$

This a particular case of the general continuation monad:

```
newtype LContT m a = LContT { unLContT ::
     forall r . (a -> m r) -> m r }
```

which is just the usual ContT continuation monad transformer (see section 3.5) with universally quantified return type.

Jaskelioff calls in [28] this monad the **codensity monad**. Jaskelioff's codensity monad seems to come from considerations in category theory. For the scope of this thesis, this is just a continuation monad.

3.3 A more complex example

Let us consider a monadic class for exception handling:

```
class Monad m => MonadError m error where
  throwError :: error -> m a
  catchError :: m a -> (error -> m a) -> m a
```

This is not the real *Haskell* MonadError defined in the *Monad transformer library* but a simplified version of it without error messages. The semantic of throwError and catchError is not important here as our only concern is correctly implementing the signature and instantiation. As before we want to implement the class MonadError as a data type DMonadError and throwError and catchError as

```
dthrowError error =
      SCont $ \k -> Call (DThrowError error k)

dcatchError v h =
      SCont $ \k -> Call (DCatchError v h k)
```

error is a parameter of MonadError, it has to be also a parameter of DMonadError. a is a universally quantified type variable in throwError and catchError, it has to be also universally quantified in DThrowError and DCatchError. Furthermore, because the abstract monad m satisfying MonadError will be implemented as the concrete SCont (DMonadError **error**) monad. The arguments of dcatchError have to be of type SCont (DMonadError **error**) a and **error** −> SCont (DMonadError **error**) a instead of m a and **error** −> m a. The datatype implementing MonadError is then:

```
1 data DMonadError error r =
2     forall a . DThrowError (a -> r)
3   | forall a . DCatchError (SCont (DMonadError error) a)
4                            (error -> SCont (DMonadError error) a)
5                            (a -> r)
```

In practice it is easier to define DMonadError as

```
1 data DMonadError error mc r =
2     forall a . DThrowError (a -> r)
3   | forall a . DCatchError (mc a)
4                            (error -> mc a)
5                            (a -> r)
```

and modify SCall to instantiate mc to SCont theclass:

```
1 data SCall datatype finalvalue =
2    End finalvalue
3  | Call (theclass (SCont datatype)
4                   (SCall datatype finalvalue)
5         )
```

As expected dthrowError and dcatchError are

```
1 dthrowError :: error -> SCont (DMonadError error) a
2 dthrowError error =
3       SCont $ \k -> Call (DThrowError  error  k)
4
5
6 dcatchError :: SCont (DMonadError error) a
7      -> (error -> SCont (DMonadError error) a)
8      -> SCont (DMonadError error) a
9 dcatchError v h  =  SCont $ \k -> Call (DCatchError v h  k)
```

The interpretation function follows the same principles as before but we can not directly give the arguments stored by dcatchError to catchError because

dcatchError arguments are of types SCont (DMonadError **error**) a and (**error** −> SCont (DMonadError **error**) a whereas catchError accepts arguments of types m a and **error** −> m a where m is an abstract monad satisfying the MonadError class. So we have to map arguments from SCont (DMonadErorr **error**) to m what is exactly what instantiation is. The interpretation function is then:

```
1 interpretError :: (MonadError error m) =>
2         DMonadError error
3                       (SCont (DMonadError error))
4                       (SCall (DMonadError error) a)
5     -> m (SCall (DMonadError error) a)
6
7 interpretError (DThrowError error k) =
8     do v <- throwError error
9        return $ k v
10
11 interpretError (DCatchError m h   k) =
12     let m'    = instantiate interpretError    m
13         h' e  = instantiate interpretError $  h e
14     in do v <- catchError m' h'
15           return $ k v
```

Obviously the instantiation function from SCont (DMonadError **error**) to m is

```
1 instantiateError :: (MonadError error m) =>
2            SCont (DMonadError error) a -> m a
3
4 instantiateError = instantiate interpretError
```

3.4 General morphism

On a general manner we can see all previous monadic operations as

```
1 op :: ∀β̄.  F m → m y
```

where F is a functor from the category of monads to a category C.

In the case of write, C is the category of strings (**String**). For throwError, C is the categories of values of type **error**. For catchError, C is the category of values of type

(m a , **error** −> m a)

. write (resp. throwError) takes the same arguments of dwrite (resp. dthrowError) because in this case

(Log m) =>F m = F (SCont DLog) (resp.
(MonadError **error** m) =>F m = F (SCont (DMonadErorr **error**))

. But in the case of catchError,

$$\text{(MonadError error m)} \Rightarrow \mathtt{F}\ m \neq \mathtt{F}\ \text{(SCont (DMonadErorr error))}$$

so to interpret DCatchError we had to map the arguments from

SCont (DMonadError **error**)

to m. The piece of code

```
1 let m'     = instantiate interpretError    m
2     h' e   = instantiate interpretError $  h e
3 in ...
```

is actually equivalent to

```
1 let (m',h') = F (instantiate interpretError) (m , h) in ...
```

because instantiate interpretError is a function of type

(MonadError **error** m) =>SCont (DMonadError **error** m) a =>m a

so F (instantiate interpretError) maps arguments of dcatchError to arguments of catchError.

Let us formalize the transformation and consider the following class:

```
1 class Monad m ⇒ TheClass m ᾱ where;
2    op_1 :: ∀β̄_1 . F_1 m → m y_1;
3    ...;
4    op_p :: ∀β̄_p . F_p m → m y_p
```

where (**Monad** m) => indicates that m has to be a monad. $\mathtt{op}_i$ are the p operations that m has to implement. $\overline{\alpha}$ and $\overline{\beta}_i$ are lists of distinct type variables.

1 $\mathsf{op}_i :: \forall (m :: * \to *)\ \overline{\alpha}\ \overline{\beta}_i\ .\ (\mathsf{TheClass}\ moveralpha) \Rightarrow \mathtt{F}_i\ m \to m\ y_i$

$\overline{\alpha}$ are class parameters common to every $\mathtt{op}_i$ whereas $\overline{\beta}_i$ are not parameters of the class but specific to every $\mathtt{op}_i$. Such a class is called a **monadic class signature**.

The complete type of each $\mathtt{op}_i$ is

The class TheClass is implemented as the algebraic data type with existential quantification TheDataType, called a **monadic signature datatype**:

1 **data** $\mathsf{DataType}\ \overline{\alpha}$ n r =;
2 $\quad \forall \overline{\beta}_1\ .\ \mathsf{Op}_1\ (\mathtt{F}_1\ \mathsf{n})\ (y_1 \to r)$;
3 $\quad | \ldots$;
4 $\quad | \forall \overline{\beta}_p\ .\ \mathsf{Op}_p\ (\mathtt{F}_p\ \mathsf{n})\ (y_p \to r)$

Every Op_i is the constructor corresponding to $\mathtt{op}_i$. It has type

1 $\mathsf{Op}_i :: \forall \overline{\alpha}\ \mathsf{n}\ \mathsf{r}\ \overline{\beta}_i\ .\ \mathtt{F}_i\ \mathsf{n} \to (y_i \to r) \to \mathsf{DataType}\ \overline{\alpha}\ \mathsf{n}\ \mathsf{r}$

The general form of SCall and SCont are

```
1 data SCall datatype e =
2     End e
3   | Call (theclass (SCont datatype)
4                    (SCall datatype e)
5          )
6
7 newtype SCont datatype a = SCont { unSCont ::
8     forall e . (a ->  SCall datatype e) -> SCall datatype e
9   }
```

For the correctness of the transformation, using directly the constructors End and Call will be forbidden. The following function call will be used to implement monadic calls:

```
1 call :: (forall r . (a -> r) -> datatype (SCont datatype) r)
2             -> SCont datatype a
3 call a = SCont (\k -> Call (a k))
```

every op_i is then implemented as:

```
1 dop_i :: ∀ᾱ β̄_i . F_i (SCont datatype) → SCont datatype y_i;
2 dop_i t = call (Op_i t)
```

```
1 interpret (Op_i t) = do   r ← op_i (F_i (instantiate interpret) t);
2                           return (k r)
```

and interpretation functions as:

The instantiation function becomes:

```
1 instantiate :: (Monad m) =>
2      (datatype (SCont datatype) (SCall datatype a)
3              -> m (SCall datatype a)
4      )
5    -> SCont thecall a -> m a
6 instantiate interpret t = aux (unSCont t End)
7   where  aux (End    r)  = return r
8          aux (Call   c)  = do  n <- interpret c
9                                aux n
```

3.5 Monadic Tower and the CPS Hierarchy

Monad transformers [29] are a way to build a new monad on top of another one. It enables us to combine monads and thus their effects instead of having to define a monolithic monad. For example, let m be a monad. To add a global state to m, we can simply use the StateT monad transformer defined in the *GHC library* as:

```
1 newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
```

StateT s m is the monad with as a global state of type s and two operations get and put to fetch the state or put one. Because it has been made from m, it has the behavior or m.

We can use another monad transformer on it: for example, to give StateT s m the capabilities of the list monad, we can use the ListT transformer:

```
1 newtype ListT m a = ListT { runListT :: m [a] }
```

ListT (StateT s m) is then a monad with the behavior of m, a global state and multiple computation branch capabilities. Monad transformers provide a function

to lift any computation in a monad m (m a) to a computation in the transformed monad (t m a) where t is the transformer. For example, for StateT, the lifting function is:

```
1 lift :: (Monad m) => m a -> StateT s m a
2 lift m = StateT $ \s -> do
3                 a <- m
4                 return (a, s
```

Stacking transformers this way gives what is called a **monadic tower**. Each transformer in the tower is called a **layer**. The lowest layer is the initial monad and the uppermost layer is the transformer at the top of the stack.

The transformation to first-class monadic signatures can be extended very easily to monad transformers. The implementation can be found in appendix A.2 and practical examples of monadic towers made of first-class monadic signatures are all around the thesis and the implementation.

Let us consider an example of tower made of monadic classes and its transformed version using first-class monadic signatures. Let us take Log as the lowest layer, then add a monad transformer implementing a state monad and finally one implementing exception handling:

```
1 exampleoftower  :: (  MonadState [Char] (t (t1 m)),
2                       MonadTrans t,
3                       MonadError [Char] (t1 m),
4                       MonadTrans t1,
5                       Log m) =>
6                       t (t1 m) [Char]
7 exampleoftower = do s <- get
8                     if null s
9                      then lift $ throwError "empty␣state"
10                     else lift $ lift $ write s
```

t and t1 are monad transformers, t1 implements a MonadError, t implements a MonadState and m has to satisfy Log. The same example can be expressed with out approach:

```
1 dexampleoftower :: SContT (DState        [Char])
2                  ( SContT (DMonadError  [Char])
3                  ( SCont   DLog                ))  [Char]
4 dexampleoftower = do s <- dgetT
```

```
5                           if null s
6                            then lift $ dthrowErrorT "empty_state"
7                            else lift $ lift $ dwrite s
8                           return "Ok"
```

We believe it to be much easier to read and in practice and because first-class monadic signatures use explicit instantiation, it also makes the life of the typing inference algorithm better.

Remark 1 *First-class monadic signatures being implemented via continuations. Towers made of First-class monadic signatures are actually in the **CPS hierarchy** [9, 18]*

3.5.1 Lifting operations

To catch the error thrown in the previous example, we would naively write:

```
1 catchError exampleoftower (\ _ -> return "Not_OK")
```

Unfortunately it does not work because exampleoftower is not of type (t1 m) [**Char**] but t (t1 m) [**Char**]. It is even clearer with dexampleoftower: dcatchError expects a SContT (DMonadError **error**) DOTS but dexampleoftower is a SContT (DState [**Char**]) DOTS. Fortunately there exists a technique to lift monadic operators.

Let us consider a monadic operation op over a monad m:

$$op :: \forall \overline{\beta}\ .\ m\ a \rightarrow m\ a$$

and a monad transformer t. We want to lift op to t m. op does not accept arguments of type t m a but it does accept arguments of type m (t m a). The return value is then also of type m (t m a). By lifting it we get t m (t m a). We only have to join it to get a value of type t m a.

The lifting operation is then for op:

```
1 liftedOp t = join $ lift $ op (return t)
```

On a general manner, for a monadic operation op:

$$op :: \forall \overline{\beta}\ .\ F\ m \rightarrow m\ y$$

the lifted operation is

```
1 liftedOp t = join $ lift $ op (F return t)
```

This technique has been used by Filinski [23] to implement its monadic layers. The composition `join . lift` corresponds to glue function.

For catchError we get:

```
1 liftedCatchError :: (Monad (t m), MonadTrans t, MonadError e m) =>
2      t m a -> (e -> t m a) -> t m a
3 liftedCatchError t h = join $ lift $ catchError (return t) (return . h)
4
5
6 liftedDCatchError :: (Monad m) =>
7             SContT    theclass
8          (  SContT (  DMonadError error) m) a
9      -> (error ->     SContT theclass
10                   (  SContT (DMonadError error) m) a)
11     ->       SContT theclass
12         (  SContT (DMonadError error) m) a
13 liftedDCatchError t h = join $ liftSContT $
14                              dcatchErrorT (return t) (return . h)
```

what is exactly what we want. Unfortunately, even it the types are correct, this transformation is not correct for every monad. It is not for the **Maybe**, **Either error** and ErrorT **error** m monads provides in the *Monad Transformer Library*. Jaskelioff presents in [27, 28] a general lifting way. We present in the next section a way to get the monad right right using backtracking. It seems that Jaskelioff's approach is the categorical general counterpart of the backtracking ad-hoc solution.

3.5.2 Well behaved Error Monad

The usual error monad is as follows

```
1 data Either errror value = Left error
2                          | Right value
```

and its implementation is

```
1 instance Monad (Either error) where
2  return v = Right v
3  m >>= f  = case m of
```

```
4                 Left error  -> Left error
5                 Right value -> f value
```

Let us introduce a very simple monad transformer, the identity transformer:

```
1 newtype IdentityT m a = IdentityT { runIdentityT :: m a }
2  deriving (Monad)
3
4 liftIdentityT :: m a -> IdentityT m a
5 liftIdentityT = IdentityT
6
7 instance MonadTrans IdentityT where
8  lift = liftIdentityT
```

Now let us try to lift catchError to IdentityT (**Either error**)

```
1 liftedCatchError t h
2       = join $ lift $ catchError (return t) (return . h)
3       = join $ lift $ catchError (Right t) (return . h)
4       = join $ lift $ Right t
5       = join $ IdentityT (Right t)
6       = IdentityT (Right t) >>= (\x -> x)
7       = t
```

so whatever t is, liftedCatchError will never catch the exception. Let us assume that t is an exception. **Right** t is not one, so catchError (**Right** t) (**return** . h) is right returning **Right** t. The problem is the computation should backtrack when t is reached so that h can be called. Because continuations are very well suited for backtracking, we add a continuation layer to the monad: the well behaved Error **error** monad is thus defined as LContT (**Either error**).

throwError is straightforward:

```
1 throwError :: error -> LContT (Either error) a
2 throwError e = lift $ Left e
```

catchError m h runs m and if and exception is raised, backtracks on h. The backtracking is done by running m and h with the current continuation where catchError is called:

```
1 catchError ::    LContT (Either error) a
2              -> (error -> LContT (Either error) a)
3              -> LContT (Either error) a
4 tryErrorT  m h = LContT $ \k ->
```

```
5      case unLContT m k of
6        Left   e -> unLContT (h e) k
7        Right v -> Right v
```

Now the operations of the exception handling layer can be lifted to any layer.

3.6 Conclusion

We have seen in this chapter how to implement monadic signatures via the CPS Hierarchy and and how a continuation monad can transform a problematic monad into a well behaved one. The *Haskell* code provided in appendices relies a lot on first-class monadic signatures. One of the goals of doing so was to prove that, using it, even towers with lots of layers can still be readable and manageable. Another positive aspect of the approach is efficiency: using the continuation monad to implement a given monad can be faster if there are few calls to the operations of the monad.

Chapter 4

Nominal Theory

This chapter introduces nominal terms and problems. We present several syntaxes for nominal terms. Each of them has its own benefits. We show how theses syntaxes are related and that they are all different facets of the same objects: terms with names.

4.1 Graphs and Trees

Definition 1 *A* ***directed graph*** *is a pair* (S_n, S_e) *where* S_n *is a set of elements called* ***nodes*** *and* $S_e \subseteq S_n \times S_n$ *a set of edges. A edge* e *is a pair of nodes* (o, d) *written* $o \rightarrow_e d$*.* o *is the origin and* d *the destination of the edge.* e *is said to be an incoming edge for* d *and an outcoming edge for* o*.*

An ***undirected graph*** *is a graph whose edges* (o, d) *and* (d, o) *are identified. Its edges have neither origin nor destinations but two extremities (without order).*

A ***root*** *of a directed graph is a node without incoming edge. A* ***leaf*** *is a node without outcoming edge.*

Examples of graphs are given in section 5

Definition 2 *A* ***path*** *in the graph is a sequence*

$$(n_1 \rightarrow_e n_2), (n_2 \rightarrow_e n_3), \ldots, (n_{l-1} \rightarrow_e n_l)$$

such that the origin (resp. destination) of an edge is the destination (resp. origin) of the previous (resp. following) one in the sequence. For example

$$(n_1 \rightarrow_e n_2), (n_2 \rightarrow_e n_3), (n_3 \rightarrow_e n_4)$$

is a path but

$$(n_1 \to_e n_2), (n_3 \to_e n_4), (n_2 \to_e n_3)$$

is not. n_1 *is called the origin of the path and* n_l *its destination.*

Definition 3 *A* ***directed acyclic graph*** *is a directed graph without cyclic path.*

A ***tree*** *is a directed acyclic graph with exactly one root and every other node has exactly one incoming edge.*

Size

Definition 4 (Size) *The* size *of an object* e *is written* $\mid e \mid$*.*

Definition 5 *The size of a tuple* $(e_1, \ldots, e_n)$*, a sequence* $e_1, \ldots, e_n$*, a list* $[e_1, \ldots, e_n]$ *or a set* S*, when considered as mathematical objects, is defined as*

- $\mid (e_1, \ldots, e_n) \mid = \mid e_1 \mid + \cdots + \mid e_n \mid$
- $\mid e_1, \ldots, e_n \mid = \mid e_1 \mid + \cdots + \mid e_n \mid$
- $\mid [e_1, \ldots, e_n] \mid = \mid e_1 \mid + \cdots + \mid e_n \mid$
- $\mid S \mid = \sum_{x \in S} \mid x \mid$

When considered as concrete objects (in the implementation), their size is the size of their representation.

Definition 6 *The size of a graph* $d = (S_n, S_e)$ *is defined as*

$$\mid d \mid = \mid S_n \mid + \mid S_e \mid$$

4.2 Nominal Terms

In this section, we present several syntaxes of nominal terms. Let Σ be a denumerable set of **function symbols** f, g, $\ldots$; $\mathcal{X}$ a denumerable set of **variables** $X, Y, \ldots$; and $\mathcal{A}$ a denumerable set of **atoms** $a, b, \ldots$ We assume that Σ, $\mathcal{X}$, and $\mathcal{A}$ are pairwise disjoint. Σ, $\mathcal{X}$, and $\mathcal{A}$ are **defined for the whole thesis**: unless stated otherwise, all atoms (resp. variables or functions) will be considered as elements of $\mathcal{A}$ (resp. $\mathcal{X}$ or Σ).

Definition 7 (Size) *The size of an atom a, a variable X or a function symbol f is defined as*

$$| a |=| X |=| f |= 1$$

In the intended applications, variables will be used to denote meta-level variables (unknowns), and atoms will be used to represent object level variables.

A **swapping** is a pair of (not necessarily distinct) atoms, written $(a\ b)$. **Permutations** π are lists of swappings, generated by the grammar:

$$\pi ::= \mathsf{Id} \mid (a\ b) \circ \pi$$

where Id is the **identity permutation**. We write π^{-1} for the permutation obtained by reversing the list of swappings in π. We denote by $\pi \circ \pi'$ the permutation containing all the swappings in π followed by those in π'. Set set of permutations is written Π.

Definition 8 (Size) *The size of a swapping $(a\ b)$ is defined as*

$$| (a\ b) |= 2$$

Because permutations are sequences of swappings, their size is the size of sequences.

Definition 9 (Terms) *In all the syntaxes presented here, nominal terms, denoted $s, t, u, \ldots$, are trees made of atoms, variables, function symbols, abstractions $[a]t$, tuples and permutations. $A(t)$ (resp $V(t)$) denotes the set of elements of $\mathcal{A}$ (resp. $\mathcal{X}$) that occur in t. A term t is **ground** if $V(t) = \emptyset$. For example, $\lambda[a]a$ is a ground term, and $V(\lambda[a](X, X)) = \{X\}$.*

A pair of a permutation π and a variable X is called a **suspended variable**, written $\pi{\cdot}X$; we say that π is **suspended** on X. The application of a permutation π on a term t is denoted $\pi{\cdot}t$.

Definition 10 (Size) *The size of an abstraction $[a]t$ or a suspension $\pi{\cdot}t$ is defined as*

- $| [a]t |= 3+ | a | + | t |$

- $\mid \pi{\cdot}t \mid = 1 + \mid \pi \mid + \mid t \mid$

The definition of the size will depend on the syntax but will always be the size of the graph representing them.

A **substitution** over the set of terms $\mathcal{T}$ is generated by the grammar:

$$\sigma ::= \mathsf{Id} \mid [X \mapsto t]\sigma$$

where $t \in \mathcal{T}$. $\mathcal{S}_\mathcal{T}$ denotes the set of substitutions over $\mathcal{T}$. The composition of permutation, written $\sigma \circ \sigma'$, is defined as:

$$\begin{array}{lcl} Id \circ \sigma' & = & \sigma' \\ ([X \mapsto t]\sigma) \circ \sigma' & = & [X \mapsto t](\sigma \circ \sigma') \end{array}$$

Definition 11 (Size) *The size of $[X \mapsto t]$ is defined as*

$$\mid [X \mapsto t] \mid = 3 + \mid X \mid + \mid t \mid$$

Because substitutions are sequences, their size is the size of sequences.

Substitutions can be applied to terms:

Definition 12 *Let $\mathcal{T}$ be a term set. A **substitution application function** over $\mathcal{T}$ is a function from $\mathcal{S}_\mathcal{T} \times \mathcal{T}$ to $\mathcal{T}$ such that*

$$\begin{array}{llcl} \forall t \in \mathcal{T} & t \smile Id & = & t \\ \forall t \in \mathcal{T}, \sigma, \sigma' \in \mathcal{S}_\mathcal{T} & t \smile (\sigma \circ \sigma') & = & (t \smile \sigma) \smile \sigma' \end{array}$$

where $t \smile \sigma$ denotes the application of σ on t. When there is no ambiguity, $t \smile \sigma$ will be written $t\sigma$.

Definition 13 *Let $\sigma = [X_1 \mapsto t_1] \ldots [X_n \mapsto t_n]$. σ is called a standard (resp. compact, reduced, encoded, ...) substitution if $(t_i)_{i=1}^n$ are terms of the standard (resp. compact, reduced, encoded, ...) syntax.*

Definition 14 *For any function f on $\mathcal{T}$, f can be extended to $\mathcal{S}_\mathcal{T}$ by*

$$f([X_1 \mapsto t_1] \ldots [X_n \mapsto t_n]) = [X_1 \mapsto f(t_1)] \ldots [X_n \mapsto f(t_n)]$$

Permutations will act top-down and accumulate on variables whereas substitutions act on variables.

Definition 15 *We call a* ***nominal structure*** *the triple* $(\mathcal{T}, \cdot, \lrcorner)$. $\mathcal{T}$ *denotes the set of terms,* $\cdot$ *a function from* $\Pi \times \mathcal{T}$ *to* $\mathcal{T}$ *representing the application of a permutation on a term, written* $\pi{\cdot}t$. *And* $\lrcorner$ *a substitution application function over* $\mathcal{T}$.

Definition 16 *A morphism from the nominal structure* $(\mathcal{T}_1, \cdot_1, \lrcorner_1)$ *to* $(\mathcal{T}_2, \cdot_2, \lrcorner_2)$ *is a function* φ *from* $\mathcal{T}_1$ *to* $\mathcal{T}_2$ *such that*

$$\forall t \in \mathcal{T}_1, \pi \in \Pi \quad \varphi(\pi \cdot_1 t) = \pi \cdot_2 \varphi(t)$$

and

$$\forall t \in \mathcal{T}_1, \sigma \in \mathcal{S}_{\mathcal{T}_1} \quad \varphi(t \lrcorner_1 \sigma) = \varphi(t) \lrcorner_2 \varphi(\sigma)$$

where $\varphi([X_1 \mapsto t_1] \ldots [X_n \mapsto t_n]) = [X_1 \mapsto \varphi(t_1)] \ldots [X_n \mapsto \varphi(t_n)]$

4.2.1 Standard Nominal Term

Standard Nominal terms, or just **standard terms** for short, are trees built from **atoms**, **suspended variables**, **tuples**, **abstractions**, and **function applications**. More precisely, standard terms are generated by the grammar:

$$s, t ::= a \mid \pi{\cdot}X \mid (s_1, \ldots, s_n) \mid [a]s \mid f\ t$$

We follow standard notational conventions, omitting brackets and abbreviating $\mathsf{Id}{\cdot}X$ as X when there is no ambiguity. The set of standard nominal terms is written $\mathcal{T}_s$.

For example: $(a, X, f\ c)$, $[a]f((a\ b){\cdot}X, Y, [c](c, d))$, and $[a](c\ d)(d\ e){\cdot}X$ are standard nominal terms but $[X]a$ is not.

A() and V() are defined on standard terms inductively by:

$$\begin{array}{lcl}
\mathrm{A}(a) & = & \{a\} \\
\mathrm{A}((a_1\ a_2)\ldots(a_{n-1}\ a_n){\cdot}X) & = & \{a_1, \ldots, a_n\} \\
\mathrm{A}(f\ t) & = & \mathrm{A}(t) \\
\mathrm{A}((t_1, \ldots, t_n)) & = & \bigcup_{i=1}^{n} \mathrm{A}(t_i) \\
\mathrm{A}([a]t) & = & \{a\} \cup \mathrm{A}(t)
\end{array}$$

and

$$\begin{array}{lcl} \mathrm{V}(a) & = & \emptyset \\ \mathrm{V}((a_1\ a_2)\dots(a_{n-1}\ a_n)\cdot X) & = & \{X\} \\ \mathrm{V}(f\ t) & = & \mathrm{V}(t) \\ \mathrm{V}((t_1,\dots,t_n)) & = & \bigcup_{i=1}^{n}\mathrm{V}(t_i) \\ \mathrm{V}([a]t) & = & \mathrm{V}(t) \end{array}$$

Application of permutation π on standard term t is defined by induction:

$$\mathsf{Id}\cdot_s t = t \text{ and } ((a\ b)\circ\pi)\cdot_s t = (a\ b)\cdot_s(\pi\cdot_s t),$$

where

$$\begin{array}{rcl} (a\ b)\cdot_s a & = & b \\ (a\ b)\cdot_s b & = & a \\ (a\ b)\cdot_s c & = & c \quad \text{if } c\notin\{a,b\} \\ (a\ b)\cdot_s(\pi\cdot X) & = & ((a\ b)\circ\pi)\cdot X \\ (a\ b)\cdot_s(f\ t) & = & f(a\ b)\cdot_s t \\ (a\ b)\cdot_s[n]t & = & [(a\ b)\cdot_s n](a\ b)\cdot_s t \\ (a\ b)\cdot_s(t_1,\dots,t_n) & = & ((a\ b)\cdot_s t_1,\dots,(a\ b)\cdot_s t_n) \end{array}$$

We define the instantiation of a standard term t by a substitution σ by induction:

$$t\ \mathsf{Id} = t \text{ and } t[X\mapsto s]\sigma = (t[X\mapsto s])\sigma$$

where

$$\begin{array}{rcl} a[X\mapsto s] & = & a \\ ([a]t)[X\mapsto s] & = & [a](t[X\mapsto s]) \\ (\pi\cdot X)[X\mapsto s] & = & \pi\cdot_s s \\ (\pi\cdot Y)[X\mapsto s] & = & \pi\cdot Y \\ (t_1,\dots,t_n)[X\mapsto s] & = & (t_1[X\mapsto s],\dots,t_n[X\mapsto s]) \\ (ft)[X\mapsto s] & = & f(t[X\mapsto s]) \end{array}$$

Definition 17 *$\mathcal{T}_s$ with $\cdot_s$ and the above substitution application function forms a nominal structure called the* ***standard nominal structure***

4.2.2 Compact Syntax

We present here another way to define nominal terms. **Compact nominal terms** are terms generated by the following grammar:

$$s,t ::= a \mid X \mid f \mid (t_1,t_2) \mid [a]s \mid \pi\cdot t$$

This grammar is called the **compact syntax**. Variables, atoms and constants are called **leaves**. $\pi\cdot t$ is called a **suspended term**. Leaves, abstraction and pairs

are called **non suspended terms**. The set of compact nominal terms is written $\mathcal{T}_c$.

For example: (a, X), $[a](e\ f){\cdot}(f, ((a\ b){\cdot}X, (Y, [c](c, d))))$, and $[a](c\ d)(d\ e){\cdot}(X, c)$ are compact nominal terms but $f\ a$ and (a, b, X) are not.

Definition 18 *Let t be a compact nominal term then $n_\pi(t)$ denotes the number of $\cdot$ nodes in t the number of subterms of the form $\pi{\cdot}u$)*

A() and V() are defined on compact terms inductively by:

$$\begin{array}{lcl} \mathrm{A}(a) & = & \{a\} \\ \mathrm{A}(X) & = & \emptyset \\ \mathrm{A}(f) & = & \emptyset \\ \mathrm{A}((a_1\ a_2)\ldots(a_{n-1}\ a_n){\cdot}t) & = & \{a_1, \ldots, a_n\} \cup \mathrm{A}(t) \\ \mathrm{A}((t_1, t_2)) & = & \mathrm{A}(t_1) \cup \mathrm{A}(t_2) \\ \mathrm{A}([a]t) & = & \{a\} \cup \mathrm{A}(t) \end{array}$$

and

$$\begin{array}{lcl} \mathrm{V}(a) & = & \emptyset \\ \mathrm{V}(f) & = & \emptyset \\ \mathrm{V}(X) & = & \{X\} \\ \mathrm{V}((a_1\ a_2)\ldots(a_{n-1}\ a_n){\cdot}t) & = & \mathrm{V}(X) \\ \mathrm{V}((t_1, t_2)) & = & \mathrm{V}(t_1) \cup \mathrm{V}(t_2) \\ \mathrm{V}([a]t) & = & \mathrm{V}(t) \end{array}$$

In the standard syntax, $\pi \cdot_s t$ is not a term but denotes the application of π on t as defined above but in the compact syntax $\pi{\cdot}t$ is the compact term made of the pair of the permutation π and the compact term t. The application of a permutation π on a term t, written $\pi \cdot_c t$ is simply the syntactic operation $\pi \cdot_c t = \pi{\cdot}t$.

Substitution application is defined inductively by:

$$t \smile_c \mathsf{Id} = t \text{ and } t \smile_c [X \mapsto s]\sigma = (t \smile_c [X \mapsto s]) \smile_c \sigma$$

where

$$\begin{array}{rcl} a \smile_c [X \mapsto s] & = & a \\ (t_1, t_2) \smile_c [X \mapsto s] & = & (t_1 \smile_c [X \mapsto s], t_2 \smile_c [X \mapsto s]) \\ ([a]t) \smile_c [X \mapsto s] & = & [a](t \smile_c [X \mapsto s]) \\ f \smile_c [X \mapsto s] & = & f \\ (\pi{\cdot}t) \smile_c [X \mapsto s] & = & \pi \cdot_c (t \smile_c [X \mapsto s]) \\ X \smile_c [X \mapsto s] & = & \pi{\cdot}s \end{array}$$

Definition 19 *$\mathcal{T}_c$ with permutation application and substitution application as above forms a nominal structure called the* ***compact nominal structure****.*

4.2.3 Reduced Compact Syntax

Let *RedudeCompactTerms* be the following reduction system on compact nominal terms:

$$\begin{array}{lll} t & \Rightarrow & Id{\cdot}t \quad \text{(applied only once, before any other rule!)} \\ \pi{\cdot}a & \Rightarrow & \pi(a) \\ \pi{\cdot}f & \Rightarrow & f \\ \pi{\cdot}(t_1, t_2) & \Rightarrow & (\pi{\cdot}t_1, \pi{\cdot}t_2) \\ \pi{\cdot}[a]t & \Rightarrow & [\pi(a)](\pi{\cdot}t) \\ \pi{\cdot}(\pi'{\cdot}t) & \Rightarrow & (\pi \circ \pi'){\cdot}t \end{array}$$

Proposition 1 *This system always terminates and is confluent*

Proof Let t be a compact term. Let w be the function associating to each subterm position in the tree of t a natural integer. w is defined by the following equations:

$$\begin{array}{lll} w(a) & = & w(f) = w(X) = 0 \\ w((u_1, u_2)) & = & 1 + w(u_1) + w(u_2) \\ w([a]u) & = & 1 + w(u) \\ w(\pi{\cdot}u) & = & w(u) \end{array}$$

Let us consider the measure $m(t) = (\sum_{u \in \mathcal{P}} w(u), \mid \mathcal{P} \mid)$ where $\mathcal{P}$ is the set of subterms of t of the form $\pi{\cdot}..$ with the usual lexical ordering on pairs of integers.

All the rules make the measure to decrease:

- $w(\pi{\cdot}a) = w(\pi(a)) = 0$ and the number of suspended subterm decreases
- $w(\pi{\cdot}f) = w(f) = 0$ and the number of suspended subterm decreases
- $w(\pi{\cdot}(t_1, t2)) = 1 + w((\pi{\cdot}t_1, \pi{\cdot}t_2))$
- $w(\pi{\cdot}[a]t) = 1 + w([\pi(a)](\pi{\cdot}t))$

So the system terminates.

The system is also locally confluent. The critical pairs are joinable:

$$\begin{array}{lcl}
\pi\cdot(\pi'\cdot a) & \stackrel{*}{\Longrightarrow} & (\pi\circ\pi')(a) \\
\pi\cdot(\pi'\cdot f) & \stackrel{*}{\Longrightarrow} & f \\
\pi\cdot(\pi'\cdot(t_1,t_2)) & \stackrel{*}{\Longrightarrow} & ((\pi\circ\pi')\cdot t_1,(\pi\circ\pi')\cdot t_2)) \\
\pi\cdot(\pi'\cdot[a]t) & \stackrel{*}{\Longrightarrow} & [((\pi\circ\pi')(a)]((\pi\circ\pi')\cdot t) \\
\pi\cdot(\pi'\cdot(\pi''\cdot t)) & \stackrel{*}{\Longrightarrow} & (\pi\circ\pi'\circ\pi'')\cdot t
\end{array}$$

So the system is confluent.

$*$

Normal forms are terms generated by the grammar

$$s,t ::= a \mid \pi\cdot X \mid f \mid (r_1,r_2) \mid [a]s$$

and called **reduced compact terms**. The set of reduced compact terms is written $\mathcal{T}_r$.

For example: $(a, Id\cdot X)$, $[a](a,b)$ are reduced compact nominal terms but $(a\ b)\cdot a$, $[e](c\ d)\cdot(X,a)$ is not.

When reduction rules are only applied on the head of the term, head normal forms are called **head reduced compact nominal terms** and correspond to the grammar:

$$s ::= a \mid \pi\cdot X \mid f \mid (t_1,t_2) \mid [a]t$$

where t, t_1 and t_2 are not head reduced compact terms but ordinary nominal compact terms. The set of head reduced compact nominal terms is written $\mathcal{T}_h$.

For example: $[a](a\ b)(a,X)$ and $(f,(a\ b)\cdot a)$ are nominal terms but $(a\ b)\cdot(f,X)$ is not.

Let t be a term, its normal form (resp. head normal form) is written $\overline{t}^r$ (resp. $\overline{t}^h$).

Reduced terms and head reduced terms being compact terms, the definitions of A() and V() for compact term hold for reduced and head reduced terms.

Remark 2 *Reduced compact nominal terms are standard nominal terms*

Definition 20 *The application of a permutation on reduced compact nominal terms is defined as* $\pi\ \cdot_r\ t = \overline{\pi\cdot t}^r$. *Substitution application is defined as* $t \smile_r \sigma = \overline{t \smile_c \sigma}^r$. $(\mathcal{T}_r, \cdot_r, \smile_r)$ *forms a nominal structure called the* ***reduced nominal***

structure. *$\varphi(t) = \overline{t}^r$ is a morphism from the compact nominal structure to the reduced nominal structure.*

Proof Let t be a compact nominal structure and π a permutation:
$\varphi(\pi \cdot_c t) = \overline{\pi \cdot_c t}^r = \overline{\pi \cdot t}^r = \pi \cdot_r \varphi(t)$.
Let $t, s \in \mathcal{T}_c, X \in \mathcal{X}$:

$$\begin{array}{rcl} \varphi(t \smile_r [X \mapsto s]) & = & \overline{t \smile_r [X \mapsto s]}^r \\ & = & \overline{\overline{t \smile_c [X \mapsto s]}^r}^r \\ & = & \overline{\overline{t}^r \smile_c \overline{[X \mapsto s]}^r}^r \\ & = & \overline{\varphi(t)\varphi([X \mapsto s])}^r \\ & = & \varphi(t) \smile_r \varphi([X \mapsto s]) \end{array}$$

*

4.2.4 From Standard Terms To Compact Terms

Let t be a standard nominal term. t can be encoded as a compact term with 3 new function symbols O, C and F not in Σ: O means "Open Tuple", C "Close Tuple" and F "Function".

The encoding is defined by:

$$\begin{array}{lcl} \mu(a) & = & a \\ \mu(\pi \cdot X) & = & \pi \cdot X \\ \mu([a]u) & = & [a]\mu(u) \\ \mu(f\ u) & = & (F, (f, \mu(u))) \\ \mu((t_1, \ldots, t_n)) & = & (O, (\mu(t_1), (\ldots, (\mu(t_n), C)))) \end{array}$$

where $(O, (t_1, (\ldots, (t_n, C))))$ is the usual encoding of list by nested pairs. For example $\mu(()) = (O, C)$, $\mu((t)) = (O, (t, C))$, $\mu((t_1, t_2)) = (O, (t_1, (t_2, C)))$ and so on.

Terms in the image of μ are called **compact encoded standard nominal terms**, or **encoded terms** for short. They are a particular case of reduced terms (and so of compact terms) generated by the following grammar called the **compact encoded standard nominal syntax**, or **encoded syntax** for short:

$$s, t ::= a \mid \pi \cdot X \mid [a]s \mid (F, (f, t)) \mid (O, (t_1, (\ldots, (t_n, C))))$$

where $f \notin \{F, O, C\}$. The set of encoded terms is written $\mathcal{T}_e$.

For example $\mu((f\ X, a, Y)) = (O, ((F, (f, (Id{\cdot}X))), (a, (Id{\cdot}Y, C))))$

Encoded terms being compact terms, the definition of A() and V() for compact term hold for encoded terms.

The inverse encoding is defined on encoded terms by:

$$\begin{array}{lcl}
\omega(a) & = & a \\
\omega(\pi{\cdot}X) & = & \pi{\cdot}X \\
\omega([a]t) & = & [a]\omega(t) \\
\omega((F, (f, t)) & = & f\ \omega(t) \\
\omega((O, (t_1, (\ldots, (t_n, C))))) & = & (\omega(t_1), \ldots, \omega(t_n))
\end{array}$$

Proposition 2

$$\forall t \in \mathcal{T}_s \quad \omega(\mu(t)) = t$$

$$\forall t \in \mathcal{T}_e \quad \mu(\omega(t)) = t$$

Proof

$$\begin{array}{lcl}
\omega(\mu(a)) & = & \omega(a) = a \\
\omega(\mu(\pi{\cdot}X)) & = & \omega(\pi{\cdot}X) \\
 & = & \pi{\cdot}X \\
\omega(\mu([a]t)) & = & \omega([a]\mu(t)) \\
 & = & [a]\omega(\mu(t)) \\
\omega(\mu(f\ t)) & = & \omega((F, (f, \mu(t)))) \\
 & = & \omega((F, (f, \mu(t)))) \\
 & = & f\ \omega(\mu(t)) \\
\omega(\mu((t_1, \ldots, t_n))) & = & \omega((O, (\mu(t_1), (\ldots, (\mu(t_n), C))))) \\
 & = & (\omega(\mu(t_1)), \ldots, \omega(\mu(t_n)))
\end{array}$$

$$\begin{array}{lcl}
\mu(\omega(a)) & = & \mu(a) = a \\
\mu(\omega(\pi{\cdot}X)) & = & \mu(\pi{\cdot}X) \\
 & = & \pi{\cdot}X \\
\mu(\omega([a]t)) & = & \mu([a]\omega(t)) \\
 & = & [a]\mu\omega((t)) \\
\mu(\omega((F, (f, t)))) & = & \mu(f\ \omega(t)) \\
 & = & (F, (f, \mu(\omega(t)))) \\
 & = & f\ \omega(\mu(t)) \\
\mu(\omega((O, (t_1, (\ldots, (t_n, C)))))) & = & \omega((O, (\mu(t_1), (\ldots, (\mu(t_n), C))))) \\
 & = & (\omega(\mu(t_1)), \ldots, \omega(\mu(t_n)))
\end{array}$$

*

Proposition 3 *μ is a morphism from the standard nominal structure to the reduced nominal structure.*

Proof Let t be a standard nominal term, π a permutation:
$\mu(\pi \cdot_s t) = \pi \cdot_r \mu(t)$

Indeed:

$$
\begin{array}{lcl}
\mu(\pi{\cdot}a) & = & \pi(a) \\
 & = & \overline{\pi{\cdot}\mu(a)}^r \\
\mu(\pi{\cdot}(\pi'{\cdot}X)) & = & \mu((\pi \circ \pi'){\cdot}X) \\
 & = & (\pi \circ \pi'){\cdot}X \\
 & = & \overline{\pi{\cdot}(\pi'{\cdot}X)}^r \\
 & = & \overline{\pi{\cdot}(\mu(\pi'{\cdot}X))}^r \\
\mu(\pi{\cdot}[a]t) & = & \mu([\pi(a)]\pi{\cdot}t) \\
 & = & [\pi(a)]\mu(\pi{\cdot}t) \\
 & = & [\pi(a)]\overline{\pi{\cdot}\mu(t)}^r \\
 & = & \overline{[\pi(a)]\pi{\cdot}\mu(t)}^r \\
 & = & \overline{\pi{\cdot}[a]\mu(t)}^r \\
 & = & \overline{\pi{\cdot}\mu([a]t)}^r \\
 & = & \pi{\cdot}\mu([a]t) \\
\mu(\pi{\cdot}(f\ t))) & = & (F,(f,\mu(\pi{\cdot}t))) \\
 & = & (F,(f,\overline{\pi{\cdot}\mu(t)}^r)) \\
 & = & \overline{\pi{\cdot}(F,(f,\mu(t)))}^r \\
 & = & \overline{\pi{\cdot}\mu(f\ t)}^r \\
\mu(\pi{\cdot}(t_1,\ldots,t_n))) & = & (O,(\mu(\pi{\cdot}t_1),(\ldots,(\mu(\pi{\cdot}t_n),C)))) \\
 & = & \overline{(O,(\pi{\cdot}\mu(t_1),(\ldots,(\pi{\cdot}\mu(t_n),C))))}^r \\
 & = & \overline{\pi{\cdot}(O,(\mu(t_1),(\ldots,(\mu(t_n),C))))}^r \\
 & = & \overline{\pi{\cdot}\mu((t_1,\ldots,t_n))}^r
\end{array}
$$

Let σ be a substitution over $\mathcal{T}_s$:
$\mu(t\sigma) = \mu(t) \smile_r \mu(\sigma)$ by induction on the structure of t and σ. $*$

Proposition 4 *ω is a morphism from the reduced nominal structure to the standard nominal structure.*

Proof The proof is the same as for μ. $*$

Proposition 5 *μ/ω is an isomorphism between the standard nominal structure and $(\mathcal{T}_e, \cdot_r, \smile_r)$*

4.2.5 From Nominal to First-Order

Nominal terms (either standard or compact) not containing atoms ($\mathrm{A}(t) = \emptyset$) are actually first-order terms. In the rest of this section we use compact terms but the definition and properties can easily be transposed any of the above syntax.

Definition 21 *The set of first-order terms, written $\mathcal{T}_f$, is defined as*

$$\{t \mid t \in \mathcal{T}_c \wedge A(t) = \emptyset\}$$

.

When applying a permutation on a standard or reduced nominal term, permutations accumulate on variables but in first-order syntax, variables represent first-order terms so $\pi{\cdot}X = X$. Thus we define the permutation application on first-order terms as

$$\forall t \in \mathcal{T}_f, \pi \in \Pi \quad \pi \cdot_f t = t$$

Proposition 6 *$\mathcal{T}_f$ with $\cdot_f$ as permutation application function and the standard substitution application function forms a nominal structure called the **first-order structure**.*

We present here a morphism F from compact terms to first-order terms. Let t be a compact nominal term over Σ, $\mathcal{X}$ and $\mathcal{A}$. F associate to t a first-order term with two new function symbols $\bullet$ and $[]$ of arity respectively 0 and 1. $\bullet$ will represent atoms and $[]$ abstraction. F "forgets" the nominal details of t:

$$\begin{array}{lcl}
F(a) & = & \bullet \\
F(X) & = & X \\
F(f) & = & f \\
F(\pi{\cdot}t) & = & F(t) \\
F((t_1, t_2)) & = & (F(t_1), F(t_2)) \\
F([a]t) & = & [] \;\; F(t)
\end{array}$$

For example: $F(([a](a, ((a\ b){\cdot}X, Y)), f)) = ([](\bullet, (X, Y)), f)$

Proposition 7 *F is a morphism from the compact nominal structure to the first-order structure.*

Proof Let t be a compact term and π a permutation, $F(\pi \cdot_c t) = F(t) = \pi \cdot_f F(t)$.

We check inductively on the structure of t that $F(t[X \mapsto s]) = F(t)[X \mapsto F(s)]$ $*$

Proposition 8 *Let t be a compact nominal term: $\mid t \mid \leq max_{\pi \in t} \mid \pi \mid \times (\mid F(t) \mid + n_\pi(t))$ where $\pi \in t$ ranges over the permutations appearing in t.*

Proof Because the only difference in size between first-order terms and nominal terms are permutation sizes. $*$

4.3 Nominal Problems

The predicate # specifies a **freshness** relation between atoms and terms, and $\approx_\alpha$ denotes **alpha-equivalence**. **Constraints** have the form $a\#t$ or $s \approx_\alpha t$.

Definition 22 (Size) *The size of constraints is defined as follows:*

- $\mid a\#t \mid = 3 + \mid a \mid + \mid t \mid$
- $\mid s \approx_\alpha t \mid = 3 + \mid s \mid + \mid t \mid$

A set Pr of constraints is called an **alpha-equivalence problem**. Intuitively, $a\#t$ means that if a occurs in t then it must do so under an abstractor $[a]$-. For example, $a\#b$, and $a\#[a]a$ but not $a\#a$. We sometimes write $a, b\#s$ instead of $a\#s, b\#s$, or write $A\#s$, where A is a set of atoms, to mean that all atoms in A are fresh for s. In the absence of variables, $a\#s$ and $s \approx_\alpha t$ are structural properties (to check $a\#s$ we just check that every a in s occurs under an abstractor), but in the presence of variables both predicates may depend on assumptions $a\#X$ about what will get substituted for the variables. A set of assumptions $A \# X$ is called a *freshness context* and written Δ.

Formally, we define # and $\approx_\alpha$ under the freshness context Δ inductively, by a system of axioms and rules, using # in the definition of $\approx_\alpha$ (see below). We write $ds(\pi, \pi')\#X$ as an abbreviation for $\{n\#X \mid n \in ds(\pi, \pi')\}$, where $ds(\pi, \pi') = \{n \mid \pi \cdot n \neq \pi' \cdot n\}$ is the set of atoms where π and π' differ (i.e., their difference set). Below a, b are any pair of distinct atoms.

With the standard syntax:

$$\frac{}{\Delta \vdash a\#b}(\#_{ab}) \qquad \frac{\Delta \vdash a\#s}{\Delta \vdash a\#fs}(\#_f) \qquad \frac{\Delta \vdash a\#s_1 \ \cdots \ \Delta \vdash a\#s_n}{\Delta \vdash a\#(s_1,\dots,s_n)}(\#_{tup})$$

$$\frac{}{\Delta \vdash a\#[a]s}(\#_{absa}) \qquad \frac{\Delta \vdash a\#s}{\Delta \vdash a\#[b]s}(\#_{absb}) \qquad \frac{\Delta \vdash \pi^{-1}\cdot a\#X}{\Delta \vdash a\#\pi\cdot X}(\#_X)$$

$$\frac{a \# X \in \Delta}{\Delta \vdash a\#X}(\#_\Delta)$$

$$\frac{}{\Delta \vdash a \approx_\alpha a}(\approx_{\alpha a}) \qquad \frac{\Delta \vdash ds(\pi,\pi')\#X}{\Delta \vdash \pi\cdot X \approx_\alpha \pi'\cdot X}(\approx_{\alpha X})$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1 \ \cdots \ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\dots,s_n) \approx_\alpha (t_1,\dots,t_n)}(\approx_{\alpha tup}) \qquad \frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash fs \approx_\alpha ft}(\approx_{\alpha f})$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash [a]s \approx_\alpha [a]t}(\approx_{\alpha absa}) \qquad \frac{\Delta \vdash s \approx_\alpha (a\ b)\cdot t \quad a\#t}{\Delta \vdash [a]s \approx_\alpha [b]t}(\approx_{\alpha absb})$$

With the compact syntax:

$$\frac{}{\Delta \vdash a\#b}(\#_{ab}) \qquad \frac{}{\Delta \vdash a\#f}(\#_f) \qquad \frac{\Delta \vdash a\#s_1 \ \Delta \vdash a\#s_2}{\Delta \vdash a\#(s_1,s_2)}(\#_{pair})$$

$$\frac{}{\Delta \vdash a\#[a]s}(\#_{absa}) \qquad \frac{\Delta \vdash a\#s}{\Delta \vdash a\#[b]s}(\#_{absb}) \qquad \frac{\Delta \vdash \pi^{-1}\cdot a\#t}{\Delta \vdash a\#\pi\cdot t}(\#_\pi)$$

$$\frac{a \# X \in \Delta}{\Delta \vdash a\#X}(\#_\Delta)$$

$$\frac{}{\Delta \vdash a \approx_\alpha a}(\approx_{\alpha a}) \qquad \frac{}{\Delta \vdash f \approx_\alpha f}(\approx_{\alpha f}) \qquad \frac{}{\Delta \vdash X \approx_\alpha X}(\approx_{\alpha X})$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1 \ \Delta \vdash s_2 \approx_\alpha t_2}{\Delta \vdash (s_1,s_2) \approx_\alpha (t_1,t_2)}(\approx_{\alpha tup}) \qquad \frac{\Delta \vdash ds(\pi,\pi')\#t}{\Delta \vdash \pi\cdot t \approx_\alpha \pi'\cdot t}(\approx_{\alpha\pi})$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash [a]s \approx_\alpha [a]t}(\approx_{\alpha absa}) \qquad \frac{\Delta \vdash s \approx_\alpha (a\ b)\cdot t \quad a\#t}{\Delta \vdash [a]s \approx_\alpha [b]t}(\approx_{\alpha absb})$$

We remark that $\approx_\alpha$ is indeed an equivalence relation (see [51] for more details). We write $a \# t$ and $t \approx_\alpha s$) when there is no ambiguity on Δ (usually when Δ is empty or when Δ).

Below we define reduction relations (also called rewriting relations) on standard terms or problems. If $\Longrightarrow$ is a reduction relation, we will denote by $\stackrel{*}{\Longrightarrow}$ its transitive and reflexive closure. Irreducible terms are called **normal forms**.

The following set of simplification rules from [51], acting on problems in the standard syntax, where a, b denote any pair of distinct atoms, can be used to *check* the validity of α-equality constraints.

$$
\begin{array}{rcll}
a\#b, Pr & \Longrightarrow & Pr & \\
a\#fs, Pr & \Longrightarrow & a\#s, Pr & \\
a\#(s_1,\ldots,s_n), Pr & \Longrightarrow & a\#s_1,\ldots,a\#s_n, Pr & \\
a\#[b]s, Pr & \Longrightarrow & a\#s, Pr & \\
a\#[a]s, Pr & \Longrightarrow & Pr & \\
a\#\pi\cdot X, Pr & \Longrightarrow & \pi^{-1}\cdot a\#X, Pr & \qquad \pi \neq \mathsf{Id}
\end{array}
$$

$$
\begin{array}{rcl}
a \approx_\alpha a, Pr & \Longrightarrow & Pr \\
(l_1,\ldots,l_n) \approx_\alpha (s_1,\ldots,s_n), Pr & \Longrightarrow & l_1 \approx_\alpha s_1,\ldots,l_n \approx_\alpha s_n, Pr \\
fl \approx_\alpha fs, Pr & \Longrightarrow & l \approx_\alpha s, Pr \\
[a]l \approx_\alpha [a]s, Pr & \Longrightarrow & l \approx_\alpha s, Pr \\
[a]l \approx_\alpha [b]s, Pr & \Longrightarrow & l \approx_\alpha (a\ b)\cdot s, a\#s, Pr \\
\pi\cdot X \approx_\alpha \pi'\cdot X, Pr & \Longrightarrow & ds(\pi,\pi')\#X, Pr
\end{array}
$$

This reduction relation can be adapted to problems in the compact syntax by adding the rules

$$
a\#\pi\cdot t, Pr \;\Longrightarrow\; \pi^{-1}\cdot a\#t, Pr \quad \pi \neq \mathsf{Id}
$$

$$
\begin{array}{rcl}
\pi\cdot t \approx_\alpha u, Pr & \Longrightarrow & \overline{\pi\cdot t}^h \approx_\alpha u\, Pr \\
t \approx_\alpha \pi\cdot u, Pr & \Longrightarrow & t \approx_\alpha \overline{\pi\cdot u}^h\, Pr
\end{array}
$$

Definition 23 *Let Pr be a problem and Δ a freshness context. Pr is said **valid** in the context Δ, written $\Delta \vdash Pr$ if and only if for any constraint $a \# t$ and $s \approx_\alpha u$ in Pr, $\Delta \vdash a \# t$ and $\Delta \vdash s \approx_\alpha u$. Otherwise it is **not valid**.*

The above rules generate a reduction relation on problems: $Pr \Longrightarrow Pr'$ if Pr' is obtained from Pr by applying a simplification rule. To check constraints we proceed as follows: Given a problem Pr, we apply the rules until we get an irreducible problem, i.e., a normal form. If only a set Δ of constraints of the form $a\#X$ are left, then the original problem is **valid** in the context Δ (i.e., $\Delta \vdash Pr$).

Definition 24 *Let Δ be a freshness context and σ a substitution. $\Delta\sigma$ is the problem defined as*

$$\Delta\sigma = \{a \# X\sigma \mid a \# X \in \Delta\}$$

.

Definition 25 *A* ***nominal α-equivalence problem****, or* ***α-equivalence problem*** *for short, is, given a problem Pr, to know whether there exists a freshness context Δ such that $\Delta \vdash Pr$. Such a Δ is called the solution of the nominal α-equivalence problem.*

For example, the problem $[a]X \approx_\alpha [b]X$ reduces to the set of constraints $a\#X, b\#X$; therefore $a\#X, b\#X \vdash [a]X \approx_\alpha [b]X$, as mentioned in the introduction. A problem such as $X \approx_\alpha a$ is not valid since it is irreducible; however, in this case X can be made equal to a by *instantiation* (i.e., applying a substitution) and we say that this constraint can be **solved**.

Definition 26 *A* ***nominal unification problem****, or* ***unification problem*** *for short, is, given a problem Pr to know whether it exists a freshness context Δ and a substitution σ such that for any constraint $a \# t$ and $s \approx_\alpha u$ in Pr, $\Delta \vdash a \# t\sigma$ and $\Delta \vdash s\sigma \approx_\alpha u\sigma.(\Delta, \sigma)$ is called a solution of the nominal unification problem.*

A most general **solution** to a nominal unification problem Pr is a pair (Δ, σ) of a freshness context and a substitution, obtained from the simplification rules above enriched with two **instantiating** rule labelled with substitutions:

$$\begin{array}{llll} \pi{\cdot}X \approx_\alpha u, Pr & \overset{X \mapsto \pi^{-1}{\cdot}u}{\Longrightarrow} & Pr[X \mapsto \pi^{-1}{\cdot}u] & (X \notin \mathrm{V}(()u)) \\ u \approx_\alpha \pi{\cdot}X, Pr & \overset{X \mapsto \pi^{-1}{\cdot}u}{\Longrightarrow} & Pr[X \mapsto \pi^{-1}{\cdot}u] & (X \notin \mathrm{V}(()u)) \end{array}$$

If we impose the restriction that in a constraint $s \approx_\alpha t$ the variables in t cannot be instantiated and the variables in left-hand sides are disjoint from the variables in right-hand sides, then we obtain a nominal **matching** problem. If we also require s to be linear (i.e., each variable occurs at most once in s), we obtain a **linear** nominal matching problem.

Definition 27 *A* ***nominal matching problem****, or* ***matching problem*** *for short, is, given a problem Pr where variables appearing on the left-hand side of $\approx_\alpha$ are distinct from variables appearing on the right-hand side of $\approx_\alpha$:*

$$\{X \mid X \in V(s_i) \quad s_i \approx_\alpha t_i \in Pr\} \cap \{Y \mid Y \in V(t_i) \quad s_i \approx_\alpha t_i \in Pr\} = \emptyset$$

to know whether it exists a freshness context Δ and a substitution σ such that for any constraint $a \# t$ and $s \approx_\alpha u$ in Pr, $\Delta \vdash a \# t\sigma$ and $\Delta \vdash s \approx_\alpha u\sigma.(\Delta, \sigma)$ is called a solution of the nominal matching problem.

A most general **solution** to a nominal matching problem Pr is a pair (Δ, σ) of a freshness context and a substitution, obtained from the simplification rules above enriched with an **instantiating** rule labelled with substitutions:

$$\pi{\cdot}X \approx_\alpha u, Pr \quad \overset{X \mapsto \pi^{-1} \cdot u}{\Longrightarrow} \quad Pr[X \mapsto \pi^{-1}{\cdot}u]$$

Note that there is no need to do an occur-check because left-hand side variables are distinct from right-hand side variables in a matching problem.

As shown in [51], these rules define sound and complete nominal unification and nominal matching algorithm, where the most general solution to the input problem is obtained by computing its normal form Δ and by composing the substitutions used in the instantiation rules. We give an example and refer to [51] for more details.

Example 1 *Since there is a reduction sequence:*

$$[a]X \approx_\alpha [b]b \implies X \approx_\alpha a,\ a\#b \overset{X \mapsto a}{\Longrightarrow} a\#b \implies \emptyset$$

the most general solution of $[a]X \approx_\alpha [b]b$ is $(\emptyset, [X \mapsto a])$.

Although in the case of first order terms we can, without loss of generality, restrict ourselves to matching problems with ground right-hand sides (it is sufficient to consider all variables as constants), this is not the case for nominal problems: when terms contain variables we may need to compute difference sets of permutations.

4.3.1 Morphisms

Definition 28 *A* ***nominal α-equivalence structure*** *is a 5-tuple* $(\mathcal{T}, \cdot, \llcorner, \approx_\alpha, \#)$ *where* $(\mathcal{T}, \cdot, \llcorner)$ *is a nominal structure,* $\approx_\alpha$ *an equivalence relation on terms and* $\#$ *a relation between atoms and terms such that:*

$$
\begin{array}{lll}
\Delta \vdash a \# s \wedge \Delta \vdash s \approx_\alpha t & \Rightarrow & \Delta \vdash a \# t \\
\forall \pi \in \Pi \quad \Delta \vdash s \approx_\alpha t & \Rightarrow & \Delta \vdash \pi{\cdot}s \approx_\alpha \pi{\cdot}t \\
\forall \pi \in \Pi \quad \Delta \vdash a \# t & \Rightarrow & \Delta \vdash \pi(a) \# \pi{\cdot}t \\
\forall \sigma \in \mathcal{S}_\mathcal{T} \quad \Delta \vdash s \approx_\alpha t \wedge \Delta' \vdash \Delta\sigma & \Rightarrow & \Delta'\sigma \vdash s \llcorner \sigma \approx_\alpha t\sigma \\
\forall \sigma \in \mathcal{S}_\mathcal{T} \quad \Delta \vdash a \# t \wedge \Delta' \vdash \Delta\sigma & \Rightarrow & \Delta' \vdash a \# t \llcorner \sigma
\end{array}
$$

Proposition 9 *All the nominal structures defined in the previous section, with their corresponding α-equivalence and freshness relations, are nominal α-equivalence structures. We write the α-equivalence and freshness relation on:*

- *standard terms* $\approx_{\alpha s}$ *and* $\#_s$
- *compact terms* $\approx_{\alpha c}$ *and* $\#_c$
- *first-order terms* $\approx_{\alpha f}$ *and* $\#_f$

Proof All equations are properties of the α-equivalence and freshness relations on standard terms. Please refer to [51] for more details. Proofs on standard terms can be easily adapted to compact terms. First-order syntax, ignoring all naming details, all the above equations are straightforward. $*$

Definition 29 *A morphism φ from the nominal α-equivalence structure* $(\mathcal{T}_1, \cdot_1, \llcorner_1, \approx_{\alpha 1}, \#_1)$ *to* $(\mathcal{T}_2, \cdot_2, \llcorner_2, \approx_{\alpha 2}, \#_2)$ *is a morphism from* $(\mathcal{T}_1, \cdot_1, \llcorner_1)$ *to* $(\mathcal{T}_2, \cdot_2, \llcorner_2)$ *such that* $\Delta \vdash a \#_1 t \Rightarrow \Delta \vdash a \#_2 \varphi(t)$ *and* $\Delta \vdash s \approx_{\alpha 1} t \Rightarrow \Delta \vdash \varphi(s) \approx_{\alpha 2} \varphi(t)$

Proposition 10 *All the morphisms between nominal structures defined in the previous section are morphism between nominal α-equivalence structures.*

Proof Let $\varphi_{c \mapsto r}$ be the morphism from compact terms to reduced terms. We need to prove that $\Delta \vdash a \#_c t \Rightarrow \Delta \vdash a \#_c \varphi_{c \mapsto r}(t)$ and $\Delta \vdash s \approx_{\alpha c} t \Rightarrow \Delta \vdash$

$\varphi_{c\mapsto r}(s) \approx_{\alpha c} \varphi_{c\mapsto r}(t)$. All the rules of the system $RedudeCompactTerms$ preserve α-equivalence:

$$\begin{array}{ll} t & \approx_\alpha Id{\cdot}t \\ \pi{\cdot}a & \approx_\alpha \pi(a) \\ \pi{\cdot}f & \approx_\alpha f \\ \pi{\cdot}(t_1, t2) & \approx_\alpha (\pi{\cdot}t_1, \pi{\cdot}t_2) \\ \pi{\cdot}[a]t & \approx_\alpha [\pi(a)](\pi{\cdot}t) \\ \pi{\cdot}(\pi'{\cdot}t) & \approx_\alpha (\pi \circ \pi'){\cdot}t \end{array}$$

so $t \approx_\alpha \overline{t}^r$.

Let $\varphi_{s\mapsto e}$ the morphism from standard terms to encoded terms. Let t_1 and t_2 two encoded terms, $t_1 \#_s t_2 \Leftrightarrow t_1 \#_c t_2$ and $a \#_s t_1 \Leftrightarrow a \#_s t_1$. Indeed, on encoded terms, the α-equivalence and freshness axioms in the standard (resp. compact) syntax can all be proved in the compact (resp. standard) syntax.

Let F the morphism from one of the nominal syntax to its corresponding first-order structure. We need to prove that for any term t that $\Delta \vdash a \# t \Rightarrow \Delta \vdash a \# F(t)$ and $\Delta \vdash s \approx_\alpha t \Rightarrow \Delta \vdash F(s) \approx_\alpha F(t)$. $\Delta \vdash a \# t \Rightarrow \Delta \vdash a \# F(t)$ is trivial because an atom is always fresh in a term without atoms. Note that in first-order syntax a variable X can only be substituted by a first-order term, so $a \# X$ is always true. The latter equation is proved by induction on the structure of s and t:

- If s is an atom a , t as to be the same atom a and $F(a)$ is the constant $\bullet$.
- If s is a constant or a variable, then $F(s) = s$ to the equation is trivial.
- If s is a pair (s_1, s_2) then t has to be a pair $(t_1, t_2$ with $s_1 \approx_\alpha t_1$ and $s_2 \approx_\alpha t_2$, by induction hypothesis $F(s_1) \approx_\alpha F(t_2)$ and $F(s_2) \approx_\alpha F(t_2)$ and $F(s) = (F(s_1), F(s_2))$ so $F(s) = F(t)$.
- If s is an abstraction $[a]s'$ then t has to be an abstraction $[b]t'$ and $s \approx_\alpha t$ implies $s' \approx_\alpha (a\ b){\cdot}t' \quad a \# [b]t'$ by induction hypothesis $F(s') \approx_\alpha F(t')$ and of course $a \# F(t)$, $F(t') = F((a\ b)t')$.

*

Definition 30 *Let φ a morphism from the nominal α-equivalence structure $(\mathcal{T}_1, \cdot_1, \smile_1, \approx_{\alpha 1}, \#_1)$ to $(\mathcal{T}_2, \cdot_2, \smile_2, \approx_{\alpha 2}, \#_2)$. It can be extended to:*

- *freshness constraints by* $\varphi(a \#_1 t) = a \#_2 \varphi(t)$
- α*-equivalence constraints by* $\varphi(s \approx_{\alpha 1} t) = \varphi(s) \approx_{\alpha 2} \varphi(t)$
- *a problem* Pr *by applying it to all of the constraints in* Pr.

Proposition 11 *Let* φ *be a morphism from the nominal* α*-equivalence structures* $(\mathcal{T}_1, \cdot_1, \smile_1, \approx_{\alpha 1}, \#_1)$ *to* $(\mathcal{T}_2, \cdot_2, \smile_2, \approx_{\alpha 2}, \#_2)$ *and* Pr *a problem whose terms are in* $\mathcal{T}_1$. *Then*

- *if* Δ *is a solution of the* α*-equivalence problem* Pr, *then* Δ *is solution of the* α*-equivalence problem* $\varphi(Pr)$.
- *if* (Δ, σ) *is a solution of the unification problem* Pr, *then* $(\Delta, \varphi(\sigma))$ *is solution of the unification problem* $\varphi(Pr)$.
- *if* (Δ, σ) *is a solution of the matching problem* Pr, *then* $(\Delta, \varphi(\sigma))$ *is solution of the matching problem* $\varphi(Pr)$.

Proof Let Δ be a solution of the α-equivalence problem Pr. For any constraint $a \# t$ and $s \approx_\alpha u$ in Pr, $\Delta \vdash a \# t$ and $\Delta \vdash s \approx_\alpha u$. φ is a morphism so $\Delta \vdash a \#_1 t \Rightarrow a \#_2 \varphi(t)$ and $\Delta \vdash s \approx_{\alpha 1} t \Rightarrow \varphi(s) \approx_{\alpha 2} \varphi(t)$. $\Delta \vdash a \#_2 \varphi(t) = \varphi(\Delta \vdash a \#_1 t)$ and $\Delta \vdash \varphi(s) \approx_{\alpha 2} \varphi(t) = \varphi(\Delta \vdash s \approx_{\alpha 1} t)$ Which means Δ is solution of $\varphi(Pr)$

Let (Δ, σ) be a solution of the unification problem Pr. For any constraint $a \#_1 t$ and $s \approx_{\alpha 1} u$ in Pr, $\Delta \vdash a \#_1 t\sigma$ and $\Delta \vdash s\sigma \approx_{\alpha 1} u\sigma$. $\varphi(\Delta \vdash a \#_1 t\sigma) = \Delta \vdash a \#_2 \varphi(t)\varphi(\sigma)$ and $\varphi(\Delta \vdash s\sigma \approx_{\alpha 1} t\sigma) = \Delta \vdash \varphi(s)\varphi(\sigma) \approx_{\alpha 2} \varphi(t)\varphi(\sigma)$. So $(\Delta, \varphi(\sigma))$ is solution of the unification problem $\varphi(Pr)$

Let (Δ, σ) be a solution of the matching problem Pr. For any constraint $a \#_1 t$ and $s \approx_{\alpha 1} u$ in Pr, $\Delta \vdash a \#_1 t\sigma$ and $\Delta \vdash s\sigma \approx_{\alpha 1} u\sigma$. $\varphi(\Delta \vdash a \#_1 t\sigma) = \Delta \vdash a \#_2 \varphi(t)\varphi(\sigma)$ and $\varphi(\Delta \vdash s \approx_{\alpha 1} t\sigma) = \Delta \vdash \varphi(s) \approx_{\alpha 2} \varphi(t)\varphi(\sigma)$. So $(\Delta, \varphi(\sigma))$ is solution of the matching problem $\varphi(Pr)$ $*$

This result is interesting because it relates solutions of problems in different syntaxes. For example it says that if the first-order problem corresponding to a nominal problem does not have any solution, then neither have the nominal problem. Because first-order unification and matching problems can be solved

in linear time and space, it provides a fast way to decide if some problems are solvable. It also says that solutions of nominal problem and first-order problems have the same structure, which means the same function symbol, abstractions, variables, tuples at the same position. From an implementation point of view, it says that an algorithm for the compact syntax can be used to solve a problem in the standard syntax because μ and ω provide an isomorphism between the encoded and standard structures. Note that the complexity of the translation has to be taken in account in the complexity of the algorithm.

So for the rest of the thesis, we will use the standard or compact syntax depending on which one is the more adapted to the situation.

Definition 31 *In the compact syntax, any α-equivalence, unification or matching problem can be encoded in linear time and space into a single constraint $s \approx_\alpha t$ such that the set of solutions is preserved.*

Proof Indeed the following equalities form a rewriting system whose normal forms are problems of the form $\{s \approx_\alpha t\}$.

$$\begin{array}{lcl} s_1 \approx_\alpha t_1 \quad s_2 \approx_\alpha t_2 & \Leftrightarrow & (s_1, s_2) \approx_\alpha (t_1, t_2) \\ \{a_1, \ldots, a_n\} \# t & \Leftrightarrow & (a_1\ a_2)(a_2\ a_3)\ldots(a_{n-1}\ a_n)(a_n\ c)[c]t \approx_\alpha [c]t \end{array}$$

where c is a fresh atom (not already present in the problem). The system terminates in linear time in the number of $\approx_\alpha$ and # present in the problem because one $\approx_\alpha$ or one # is consumed every time but none is created. $*$

4.4 Complexity and Implementation of Permutations and Sets

The complexity of the algorithms presented in this thesis depends a lot on the implementations of permutation and sets of atoms. Because we will have to use different implementations of permutations and sets of atoms even for the same algorithm, we need to abstract their actual implementation to make the algorithm to be able to use several implementations.

4.4.1 Implementation

This section describes the implementation and abstraction of permutations and sets. It has been implemented in the *Haskell* module A.6. The implementation in the *Objective Caml* B.1 does not need all the mechanisms presented here, it only needs the array implementation.

The operations on permutations and sets of atoms used in the algorithms will be :

- $a \in A$: membership test
- $A \cup \{a\}$: add an atom to a set
- $A \setminus \{a\}$: remove an atom from a set
- $A \cup A'$: compute the union of two sets
- $A \cap A'$: compute the intersection of two sets
- $\pi{\cdot}a$: compute the image of an atom by a permutation
- π^{-1}: compute the inverse of a permutation
- $\pi \circ \pi'$: compose two permutations
- $supp(\pi)$: compute the support of a permutation
- $\pi(A) = \{\pi(a) \mid a \in A\}$: compute the image of a set by a permutation.

Operations in this list are called the **basic operations** on permutations and sets of atoms.

Definition 32 *We call **updating** the action of adding/removing an atom to a set or to compose a permutation by another one.*

Definition 33 (Implementation as arrays) *A and π can be implemented respectively as mutable arrays arr_A and arr_π indexed by atoms. Let a be an atom, arr_A is defined as $arr_A[a] = (a \in_? A)$ and $arr_\pi[a] = \pi(a)$.*

Mutable arrays have constant access time but computing $A\{a\}$ requires either to modify the array in place by $arr_A[a] := \top$ or to create another array arr' such that

$$arr'[b] = \begin{cases} \top & \text{if } b = a \\ arr_A[b] & \text{otherwise} \end{cases}$$

Modifying arr_A in place to get arr' takes a constant time but modifies all the other occurrences of A which become A'. To keep A, we need to create a new array arr' but creating the array takes is linear in time and space in the size of arr_A.

Definition 34 *The size of an array arr is defined as* $\mid arr \mid = l \times n$ *where* l *is the length of arr (the size of its domain) and* n *the size taken,* ***in the array****, by an element. If the elements are pointers to other structures, the size an element is the size of the pointer and not the size of the structure pointed.*

Another implementation is using **persistent sets and maps**. Persistent sets (resp. maps) are *size balanced binary trees* whose nodes are elements in the sets (resp. a pair of a key and a value). They correspond to the *Haskell* module Data.Set (resp. Data.Map) and are described in [8, 39].

Persistent maps represent partial functions. Their domain of definition is a set of keys for which they associate a value. Let pm be a persistent map. For a key k in the domain of definition of pm, the value associated to k is $pm[k]$.

Definition 35 (Size) *Because such data are trees, their size is the size on graphs.*

Definition 36 (Implementation as persistent data) *A can be implemented directly as the corresponding persistent set* ps_A π *can be implemented as the finite map* pm_π *whose domain of definition is the support of* π *and*

$$\forall a \in supp(\pi) \qquad pm_\pi[a] = \pi(a)$$

Membership tests and value look up take logarithmic time. Computing $A \backslash \{a\}$ (resp. $\pi \circ (a\ b)$) returns a new set (resp. map) and takes logarithmic time too. The original set (resp. map) remains unmodified. Thus using persistent data, sets permutations can be passed away without risks of side effects, so no need to copy them.

4.4.2 Optimisation

The type of sets is actually

```
1 data Set set  =  SetEmpty
2               |  SetNative set
```

where set represents an abstract implementation of set like arrays, persistent sets, ... Adding the value SetEmpty enable optimisations:

- there is no need to create a whole array, which takes a linear time and space in the size of A_0, for the empty set
- $A \cup \emptyset$ or $A \cap \emptyset$ can be computed in constant time

The type of permutation is:

```
1 data Perm set perm =
2       PermId
3     | PermSwap    Atm   Atm
4     | PermNative  perm  perm  (Set set)
```

where set is as before and perm represents an abstract implementation of permutations like arrays or persistent data. PermId represents the id permutation and PermSwap a b the swapping $(a\ b)$. PermNative p i s represents any permutation π where p is the native implementation of π, i the native implementation of π^{-1} and s its support. It also enable lots of optimisations:

- the identity permutation or a swapping do not depend on the native implementation
- composing by PermId is done in constant time
- in the case of arrays, composing in place by PermSwap a b can be done in constant time whereas it would take a linear time in the size of A_0 otherwise
- getting the inverse of a permutation and its support is done in constant time because it is already there.

When permutations are composed, we only need to update the inverse and the support and not to recompute them. For example, let π' be a permutation

$$(\pi \circ (a\ b))^{-1} = (a\ b) \circ \pi^{-1}$$

and $supp(\pi \circ (a\ b))$ is computed by adding or removing a and b whether $(\pi \circ (\underline{a}))a = a$ and $(\pi \circ (\underline{a}))a = a$.

4.4.3 Abstraction of sets and permutations

To abstract the native implementation of sets we add another layer in the monadic tower for set handling. This layer is implemented by the first-class monadic signature SetCall set where set represents the native type of sets. SetCall set provides the following calls:

- SetIsIn checks if an atom is in a set
- SetIsSubset checks if the first set is subset of the second
- SetSize returns the size of a set
- SetSet add or remove an atom from a set, of the first argument is $\top$ the modification can be done in place otherwise a new set has be returned
- SetEmptyNative returns a native empty set
- SetToList returns the list of the elements of a set
- SetUnion/SetInter compute the union/intersection of two sets. The first argument tells if the modification can be done in place and if yes, on which set.

The native implementation of permutation is also abstracted. A new layer is added to the monadic tower for permutation handling. It is implemented by the first-class monadic signature data type PermCall perm where perm represent the native type of permutations. The following calls are provided:

- PermImage returns the image of an atom by a permutation

- PermSwapRight composes a permutation by a swapping on the right side
- PermCompose composes two permutations. The first argument tells if the composition can be done in place and if yes on which permutation.
- PermSupp computes the support of a permutation
- PermIdNative the native identity permutation
- PermUnshare makes a fresh copy of a permutation

4.4.4 Complexity

Because it will happen that the same algorithm uses different implementations of permutations and sets of atoms we need to have a way to abstract the actual implementation also in complexity proofs.

Definition 37 *For a given implementation of permutations and sets of atoms we define $\mathcal{C}_\mathcal{A}$ as an upper bound of the time complexity of the basic operations on permutations and sets of atoms (*cf. *section 4.4.1).*

In practice it is impossible to handle the infinite set of atoms $\mathcal{A}$. Instead we take a big enough set such as the set A_0 of atoms in the problem. $\mathcal{C}_\mathcal{A}$, in every implementation considered, depends on the number of atoms (i.e. the size of A_O, written $\mid A_0 \mid$).

Definition 38 *$\mathcal{C}_\mathcal{A}$ for the implementation of permutations and sets of atoms as arrays is $\mid A_0 \mid$ where A_0 is the set of index of the arrays.*

Proof Every basic operation time complexity is bounded by $\mathcal{C}_\mathcal{A}$ $*$

Definition 39 *$\mathcal{C}_\mathcal{A}$ for the implementation of permutations and sets of atoms as persistent data is $log(\mid A_0 \mid) \times \mid A_0 \mid$ where A_0 is the set of atoms used in the implementation.*

Proof Every basic operation time complexity is bounded by $\mathcal{C}_\mathcal{A}$ $*$

The interpretation of SetCall and PermCall as persistent data is given in the appendix A.6.

Part II

Unification

An implementation of the unification algorithm using trees to represent terms, is exponential. For instance, the problem:

$$\begin{array}{rcl} X_1 & \approx_\alpha & (X_2, X_2) \\ Y_1 & \approx_\alpha & (Y_2, Y_2) \\ & \dots & \\ X_{n-1} & \approx_\alpha & (X_n, X_n) \\ Y_{n-1} & \approx_\alpha & (Y_n, Y_n) \\ X_n & \approx_\alpha & a \\ Y_n & \approx_\alpha & a \\ X_1 & \approx_\alpha & Y_1 \end{array}$$

which does not involve abstraction, requires a number of reduction steps that is exponential in the size of the problem because the solution size, without subterm sharing, is exponential in the size of the problem.

For first-order unification problems, there are linear-time unification algorithms that represent terms as directed acyclic graphs. However, it is well-known that these algorithms cannot be generalised to higher-order problems (i.e. unification of λ-terms modulo α and β); higher-order unification is undecidable. In the case of unification modulo α, no linear algorithm is known, but we will show that using a representation of problems as directed acyclic graphs we can obtain a quadratic algorithm which is the most efficient implementation so far.

We will start by presenting in chapter 5 a representation of nominal terms and problems as directed acyclic graphs. Then we will present 3 nominal unification algorithms in chapters 6, 7 and 8. The first one is the most efficient (it is quadratic in time and space) but not the easiest to to maintain and extend. The second one is a almost as efficient as the first one and easier to use in practice. The last one presents a polynomial algorithm based on graph rewriting.

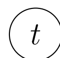
Chapter 5

Nominal Unification Terms and Problems as Directed Acyclic Graphs

This chapter presents an encoding of compact nominal terms as directed acyclic graphs. We use compact terms because we will deal with permutations in a lazy way: permutations will suspend always, unless performing the permutation is necessary to trigger a simplification rule in a problem.

We will define a recursive function to transform terms, and more generally problems, into graphs. Nodes are represented by circles, annotated by a constructor. The term-constructors are: abstraction [], pairs (), constants f, permutations π, permutation application $\bullet$, variables X, atoms a and sets of atoms A. In addition we use two constructors for freshness and α-equality: $\#$, $\approx_\alpha$, and a node Pr which is the root of the problem. The constructors $[], \#, \approx_\alpha, \bullet, ()$ are binary, f is unary, X, π, a are 0-ary, Pr has variable arity, equal to the the number of constraints in the problem, respectively.

We will often overload the notation, by using a node annotated with a term t, as in the following diagram:

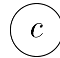


to denote the root node of the subgraph representing t. In the same way, we

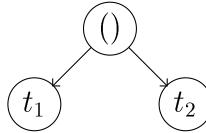
sometimes annotate a node with a constraint P_i to denote the root of the subgraph representing the constraint.

Definition 40 *The graph representation of a term, and more generally a problem, is inductively defined by:*

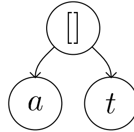
- *c where c is an atom a, a constant f or a variable X:*



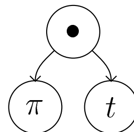
- *(t_1, t_2):*



- *$[a]t$:*

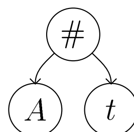


- *$\pi{\cdot}t$:*

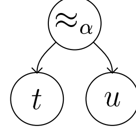


  In particular, the graph representing $\pi{\cdot}X$ has two leaves labelled by π and X.

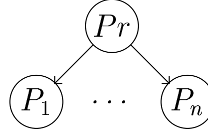
- *$A \# t$, where A is a set of atoms:*



- *$t \approx_\alpha u$:*



- *$Pr = P_1, \ldots, P_n$, where each P_i is a constraint:*



In this case, we fix an arbitrary ordering on the constraints $P_1, \ldots, P_n$, to obtain a unique representation.

After transforming a problem into a graph as above, we identify the leaves that are annotated with the same atom, constant or variable but not leaves that are annotated by the same permutation.

Having a root node Pr for a problem is not essential, but it will be useful when we define garbage collection on graphs.

The graph representation of a nominal problem, as defined above, is a **nominal graph** (see Definition 41 below). Nominal graphs are directed acyclic graphs, they can be seen as termgraphs [43], that is, trees where subterms may be shared.

We will also include $\perp$ nodes to represent unsolvable constraints, and σ nodes to represent solutions (substitutions).

Remark 3 *Definition 40 specifies a function that transforms a nominal problem into a graph. This function is linear in the size of the problem.*

Definition 41 (Nominal graph) *A nominal graph is a set N of nodes labelled with constructors (i.e., term constructors, and $\#$, $\approx_\alpha$, $\perp$, σ, Pr), together with a set $E \subseteq N \times N$ of edges, such that if n is a node decorated with a symbol of arity m, then there are m outcoming edges for n (i.e., it has m children). If*

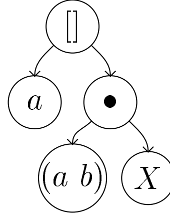
$e = (n_1, n_2) \in E$, written $n_1 \to n_2$, we say that e is an incoming edge for n_2, and outcoming for n_1.

*The edges connected to each node are ordered, and we will call the points of attachment **ports**.*

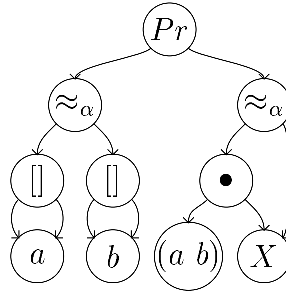
A root is a node without incoming edges.

Different nodes may have the same decoration, however, leaves with the same label are identified (they are the same node) except for permutations.

Example 2 *The nominal graph representing the term $[a](a\ b)\cdot X$ is:*



and the nominal graph representing the problem $[a]a \approx_\alpha [b]b, (a\ b)\cdot X \approx_\alpha X$ is:



Note that we have fixed an arbitrary order for the constraints above. The root of a graph representing a problem is labelled by Pr, and its children have roots labelled by $\approx_\alpha$ or $\#$. We will say that the root of the i-th constraint is the i-th root of the problem.

We presented here an encoding of compact nominal terms as directed acyclic graphs. The same technique can be used to define an encoding of standard, reduced, head reduced, encoded and first-order terms as directed acyclic graphs.

Chapter 6

Extending Paterson and Wegman's First-Order Unification Algorithm

In this chapter we present an adaptation of *Paterson* and *Wegman* 's First-Order Linear Unification Algorithm *(FLU)* [40] to nominal unification. We will show that, the first-order and nominal algorithms have the same structure. Because any unification problem can be encoded in linear time and space to a single constraint $s \approx_\alpha t$, in the rest of this chapter, we only consider unification problems of the form $\{s \approx_\alpha t\}$. The set of atoms in the problem, written A_0 is defined as $A_0 = \mathrm{A}(s) \cup \mathrm{A}(t)$. In the rest of the chapter, we consider a nominal graph d representing the problem $s \approx_\alpha t$.

Section 6.1 gives a theoretical characterization of nominal unification as a relation on the nodes of d. It will be used to prove the correction of the algorithm presented in section 6.2. A concrete implementation of the algorithm in *Objective CAML* [6] is in appendix B.1. Please refer to it for a complete description of the actual implementation of the algorithm.

6.1 Nominal Unification as a Relation on Term Nodes

The *FLU* algorithm computes the solution, if it exists, by computing a relation on the nodes of the directed acyclic graph representing the problem. To prove the

correctness of their algorithm Paterson and Wegman developed in [40] the concept of **valid** equivalence relation. To extend their algorithm to nominal terms, we need to start by extending their notion of (valid) equivalence relation to nominal directed acyclic graphs.

Definition 42 *A nominal relation R is a pair of a 3-ary relation $\mathbb{R}_{?\approx?,}$ between nodes and permutations, written $s \, \mathbb{R}_{?\approx?,\pi} \, t$, and a binary relation $\mathbb{F}$ between atoms and nodes, written $a \, \mathbb{F} \, s$. Two nodes s and t are related by R, written $s \, R \, t$ if and only if it exists a permutation π such that $s \, \mathbb{R}_{?\approx?,\pi} \, t$.*

A nominal relation is a nominal equivalence relation if:

- $s \, \mathbb{R}_{?\approx?,id} \, s$ *for any* s *(Reflexivity)*
- $s \, \mathbb{R}_{?\approx?,\pi} \, t \Rightarrow t \, \mathbb{R}_{?\approx?,\pi^{-1}} \, s$ *(Symmetry)*
- $s \, \mathbb{R}_{?\approx?,\pi_1} \, u \qquad u \, \mathbb{R}_{?\approx?,\pi_2} \, t \Rightarrow s \, \mathbb{R}_{?\approx?,\pi_1 \circ \pi_2} \, t$ *(Transitivity)*
- $s \, \mathbb{R}_{?\approx?,\pi} \, s \Rightarrow (a \, \mathbb{F} \, s)_{a \in supp(\pi)}$ *(Disagreement set)*
- $a \, \mathbb{F} \, s \quad s \, \mathbb{R}_{?\approx?,\pi} \, t \Rightarrow \pi^{-1}(a) \, \mathbb{F} \, t$ *(Congruence)*

Definition 43 *Let $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ be a nominal relation. Let us define the function fc over nodes as*

$$fc(s) = \{a \mid a \, \mathbb{F} \, s\}$$

Proposition 12 *If $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ is a nominal equivalence relation then*

$$s \, \mathbb{R}_{?\approx?,\pi} \, t \Rightarrow fc(s) = \pi(fc(t))$$

Proof Let a be in $fc(s)$, $a \, \mathbb{F} \, s$ and $s \, \mathbb{R}_{?\approx?,\pi} \, t$ so $\pi(a) \, \mathbb{F} \, t$, $\pi^{-1}(a) \in fc(t)$ and $a \in \pi(fc(t))$. $s \, \mathbb{R}_{?\approx?,\pi} \, t \Rightarrow t \, \mathbb{R}_{?\approx?,\pi^{-1}} \, s$ gives $fc(t) \subseteq \pi^{-1}(fc(s))$ $*$

If two first-order terms do not have, on their root node, the same function symbol and the same arity, then they can not be unified. Paterson and Wegman's algorithm uses it to detect when a problem does not have any solution. We extend this notion of compatibility to nominal graphs:

Definition 44 ($\pi, \mathbb{F}$-compatibility) *Two nodes s and t are said $\pi, \mathbb{F}$-**compatible** if and only if at least one of the following properties hold:*

- *$s = \pi' \cdot s'$ and s' and t are $(\pi'^{-1} \circ \pi), \mathbb{F}$-compatible*
- *$t = \pi' \cdot t'$ and s and t' are $(\pi \circ \pi'), \mathbb{F}$-compatible*
- *s and t are two atoms, $s = \pi \cdot t$, $s \notin fc(s)$ and $t \notin fc(t)$*
- *s and t are two constants and $s = t$*
- *s and t are two pairs*
- *s and t are two abstractions*
- *s is a variable*
- *t is a variable*

When there is no ambiguity on $\mathbb{F}$, $\pi, \mathbb{F}$-compatibility will be called π-compatibility

Definition 45 *Two nodes s and t, representing first-order terms, are said **compatible** if and only if they are $Id, \emptyset$-compatible.*

Any directed acyclic graph can provide a partial ordering on the nodes of the graph:

Definition 46 *The partial order derived from the directed acyclic graph d is defined as: $n_1 >_d n_2$ if and only if there exists a path from the node n_1 to the node n_2 in d ($n_1 \neq n_2$).*

Now, we can extend the notion of *valid* equivalence relation to nominal relations. Paterson and Wegman define a valid equivalence relation as:

> An equivalence relation on the nodes of a dag is valid if it has the following properties:
>
> (i) if two function nodes are equivalent then their corresponding sons are equivalent in pairs,
>
> (ii) each equivalence class is homogeneous, that is it does not contain two nodes with distinct function symbols,
>
> (iii) the equivalence classes may be partially ordered by the partial order on the dag.

We wanted our definition as close as possible to Paterson and Wegman's. The first rule is essentially a propagation rule, the second one a compatibility rule on the nodes of a class and the third a rule to check there is no cycle in the graph. So here is the definition of a valid nominal equivalence relation:

Definition 47 *A nominal equivalence relation* $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ *on the nodes of a nominal dag is **valid** if:*

- *Propagation:*

$$
\begin{array}{lcl}
(s_1, s_2)\ \mathbb{R}_{?\approx?,\pi}\ (t_1, t_2) & \Rightarrow & s_1\ \mathbb{R}_{?\approx?,\pi}\ t_1 \quad s_2\ \mathbb{R}_{?\approx?,\pi}\ t_2 \\
[a]s\ \mathbb{R}_{?\approx?,\pi}\ [b]t & \Rightarrow & s\ \mathbb{R}_{?\approx?,(a\ \ \pi(b))\circ\pi}\ t \\
 & & (\pi(b)\ \mathbb{F}\ s \qquad \textit{if } a \neq \pi(b)) \\
a\ \mathbb{F}\ (s_1, s_2) & \Rightarrow & a\ \mathbb{F}\ s_1 \quad a\ \mathbb{F}\ s_2 \\
a\ \mathbb{F}\ [b]s & \Rightarrow & a\ \mathbb{F}\ s\ (\textit{if } a \neq b) \\
a\ \mathbb{F}\ [a]s & &
\end{array}
$$

- *Compatibility: each nominal class is* homogeneous, *that is for any s and t such that $s\ \mathbb{R}_{?\approx?,\pi}\ t$, (s, t) is $\pi, \mathbb{F}$-compatibles.*

- *Occurrence check: the partial order derived from the directed acyclic graph, quotiented by the equivalence relation, is a partial order on the equivalence classes.*

The last remaining thing we need to extend is the notion of "minimal relation"
:

Definition 48 *Let us extend the standard partial ordering over relations to nominal relations.* $(\mathbb{R}_{?\approx?,_1}, \mathbb{F}_1) \leq (\mathbb{R}_{?\approx?,_2}, \mathbb{F}_2)$ *if and only if*

$$\forall s, t, \pi \quad s\ \mathbb{R}_{?\approx?,_{1\pi}}\ t \Rightarrow \exists \pi' \quad s\ \mathbb{R}_{?\approx?,_{2\pi'}}\ t \quad \{a\ \mathbb{F}_1\ t\}_{\pi(a)\neq\pi'(a)} \quad dom(\pi) \subseteq dom(\pi')$$

and

$$\forall a, s \quad a\ \mathbb{F}_1\ s \Rightarrow a\ \mathbb{F}_2\ s$$

Now we can state the same property as in [40] between valid relations and solutions of unification problems:

Proposition 13 *Two nominal terms s and t are unifiable if and only if there exist a valid nominal equivalence relation on the graph representing $s \approx_\alpha t$ such that $s \ \mathbb{R}_{?\approx?,id} \ t$. If such a relation is minimal, then it defines a most general unifier.*

Proof If s and t are unifiable then let (σ, Δ) be a solution of the unification problem. Let $\mathbb{R}_{?\approx?,}$ be the relation containing $s \ \mathbb{R}_{?\approx?,\pi} \ t$ if and only if $s\sigma \approx_\alpha t\sigma$ under Δ. Let $\mathbb{F}$ be the relation containing $a \ \mathbb{F} \ s$ if and only if $a \# s$ under Δ.

If such a relation exists, σ can be derived from the dag by $\sigma(X) = \pi{\cdot}t$ if $X \ \mathbb{R}_{?\approx?,\pi} \ t$ and t not a variable, or $\sigma(X) = X$ is there is no such t. And $\Delta = \{a \# X \mid a \ \mathbb{F} \ X \quad \sigma(X) = X\}$ $*$

6.2 A Quadratic Nominal Unification Algorithm

The Paterson and Wegman First-Order Linear Unification Algorithm is as described by the algorithm 1 and 2. It is linear in time and space in the size of the directed acyclic graph.

The nominal version of *(FLU)* is described in algorithms 3 , 4 and 5. Like in *FLU*, the algorithm proceeds by computing a valid relation. There are two kinds of edges in the algorithm: the edges of d and undirected edges $s \ {}_{\pi^{-1}}\!\leftrightarrow_\pi t$ which represent $s \approx_\alpha \pi{\cdot}t$ (also $s \ \mathbb{R}_{?\approx?,\pi} \ t$). `pointer()` is a partially defined function whose domain is empty at the beginning of the algorithm. `pointer(`s`)` represents the class of s with the corresponding permutation.

In the rest of this chapter, we will use the similarity between *FLU* and *QNU* to prove the correctness and the complexity of *QNU*.

Proposition 14 *Applying the morphism F from compact terms to first-order terms to every operation in* QNU *gives exactly* FLU*:*

$$F(\text{QNU}) = \text{FLU}$$

Proposition 15 QNU *is correct.*

Proof Like *FLU*, *QNU* computes a valid nominal equivalence relation. So if the algorithm terminates without raising an error, then the computed relation is valid. Furthermore the relation is minimal. $*$

```
1  unify-flu(s,t) = ;
2      create-edge-flu(s,t) ;
3      while There is a non-variable node r do
4      |   finish-flu(r)
5      end
6      ;
7      while There is a variable node r do
8      |   finish-flu(r)
9      end
10     ;
11     print Unified;
12 create-edge-flu(s,t) = ;
13   create the undirected edge s ↔ t;
14 propagate-unification-edges-flu(r,s) = ;
15   switch (r, s) do
16   |   case ((r1, r2), (s1, s2))
17   |   |   create-edge-flu(r1, s1) ;
18   |   |   create-edge-flu(r2, s2)
19   |   case ([]r', []s')
20   |   |   create-edge-flu(r', s')
21   |   otherwise
22   |   |   do nothing
23   |   endsw
24   endsw
25 doFathers-flu(s) = ;
26   while there is a father t of s do
27   |   finish-flu(t)
28   end
```

Algorithm 1: First-Order Linear Unification *(FLU): Part 1*

```
1  finish-flu(r) = ;
2    Create a new empty stack stack;
3    push(r) on stack;
4    while stack is non empty do
5        s := pop(stack) ;
6        if pointer(s) is defined then
7            r' := pointer(s) ;
8            if r' ≠ r then
9                error(FAIL-LOOP)
10           end
11       else
12           pointer(s) := r
13       end
14       ;
15       if r and s are not compatible then
16           error(FAIL-CLASH)
17       end
18       ;
19       doFathers-flu(s);
20       if s is a variable node then
21           σ(s) := r
22       else
23           propagate-unification-edges-flu(r,s)
24       end
25       ;
26       while there is a undirected edge s ↔ t do
27           push(t) on stack;
28           delete the edge s ↔ t
29       end
30       if r and s are not the same node then
31           delete s
32       end
33   end
34   delete r ;
```

Algorithm 2: First-Order Linear Unification *(FLU): Part 2*

```
1  unify-qnu(s,t) = ;
2      create-edge-qnu(s, Id, t) ;
3      while There is a non-variable and non-suspended node r do
4          finish-qnu(r)
5      end
6      ;
7      while There is a variable node r do
8          finish-qnu(r)
9      end
10     ;
11     print Unified;
12 create-edge-qnu(s,π,t) = ;
13   switch (s, t) do
14       case (π'·s', t)
15           create-edge-qnu(s,π'^{-1} ∘ π,t)
16       case (s, π'·t')
17           create-edge-qnu(s,π ∘ π',t)
18       otherwise
19           create the undirected edge s _{π^{-1}}↔_π t
20       endsw
21   endsw
```

Algorithm 3: Quadratic Nominal Unification *(QNU): Part 1*

```
1  propagate-unification-edges-qnu(r,π,s) = ;
2    switch (r, s) do
3      case ((r1, r2), (s1, s2))
4        create-edge-qnu(r1, π, s1) ;
5        create-edge-qnu(r2, π, s2)
6      case ([a]r′, [b]s′)
7        create-edge-qnu(r′, (a π(b)) ∘ π, s′) ;
8        if  a ≠ π(b) then
9          add π⁻¹(a) to fc(s′)
10       end
11     otherwise
12       do nothing
13     endsw
14   endsw
15 doFathers-qnu(s) = ;
16   while there is a father t of s do
17     if t is a suspended node then
18       doFathers-qnu(t)
19     else
20       finish-qnu(t)
21     end
22   end
```

Algorithm 4: Quadratic Nominal Unification *(QNU): Part 2*

```
1  finish-qnu(r) = ;
2    Create a new empty stack stack;
3    push((id,r)) on stack;
4    while stack is non empty do
5        (π,s) := pop(stack) ;
6        if pointer(s) is defined then
7            (π',r') := pointer(s) ;
8            if r' ≠ r then
9                error(FAIL-LOOP)
10           end
11       else
12           pointer(s) := (π⁻¹,r)
13       end
14       ;
15       if r and s are not π-compatible then
16           error(FAIL-CLASH)
17       end
18       ;
19       doFathers-qnu(s);
20       if s is a variable node then
21           σ(s) := π⁻¹·r
22       else
23           propagate-unification-edges-qnu(r,π,s)
24       end
25       ;
26       while there is a nominal undirected edge s π''⁻¹↔π'' t do
27           push((π ∘ π'', t)) on stack;
28           delete the edge s π''⁻¹↔π'' t
29       end
30       if r and s are the same node then
31           add supp(π) to fc(r)
32       else
33           add π⁻¹(fc(s)) to fc(r) ;
34           delete s
35       end
36   end
37   propagate fc(r) to its children ;
38   delete r ;
```


Algorithm 5: Quadratic Nominal Unification *(QNU): Part 3*

Proposition 16 *Let d be a nominal directed acyclic graph representing the problem $s \approx_\alpha t$. The complexity* `unify-qnu(`s,t`)` *is at most $\mathcal{C}_\mathcal{A} \times (\mid F(d) \mid + n_\pi(d))$ where $\mid F(d) \mid$ is the size of the first-order directed acyclic graph corresponding to d and $n_\pi(d)$ the number of permutation application $(\cdot)$ nodes in d.*

Proof Let $F(d)$ be the direct acyclic graph obtained by applying F to every node in d the same way it is done with terms. Let us compare the execution of `unify-flu(`$F(s), F(t)$`)` and `unify-qnu(`s,t`)`. The only possible difference in the execution of the two functions is line 14 of algorithm 2 and line 14 of algorithm 4 when checking (π)-compatibly. It may happen that two terms r and s are not π-compatibly but $F(r)$ and $F(s)$ are compatible. In that case, *QNU* will halt on the error FAIL-CLASH while *FLU* continues its execution. Otherwise, the execution of `unify-flu(`$F(s), F(t)$`)` and `unify-qnu(`s,t`)` pass through the same steps. Most of the operation which take n time in *FLU* take $\mathcal{C}_\mathcal{A} \times n$ in *QNU*. The only one which does not, is the edge creation. When creating an edge $s \xrightarrow{\pi}_e t$, s and t may be suspended terms but we want s and t to be head reduced terms so we reduce them before creating the edge. Fortunately, this extra cost is bounded in the whole algorithm by $\mathcal{C}_\mathcal{A} \times \mid F(d) \mid$. $\ast$

Implementation and complexity Atoms, constants and variables are implemented as integers. Permutations and sets are implemented as arrays indexed by atoms. The actual implementation of permutations and sets is in appendix B.1. Composing and inverting permutations and permuting and computing the union of sets takes a linear time in the number of atoms in the problem. Accessing the image of an atom by permutation, or performing a membership test takes a constant time. So with this implementation of permutations and sets of atoms $\mathcal{C}_\mathcal{A} = \mid A_0 \mid$ where A_0 is the set of atoms appearing in the problem.

Remark 4 *When $\mid d \mid$ is close $\mid A_0 \mid \times (\mid F(d) \mid + n_\pi(d))$ and with mutable arrays as permutations and sets implementation, the unification algorithm becomes linear in time and space.*

Chapter 7

A Simple and Efficient Nominal Unification Algorithm

QNU is a good algorithm and provides a concrete proof that nominal unification can be done in quadratic time. But *QNU* relies a lot on very ad-hoc imperative techniques which make it difficult to extend and in practice manipulations on directed acyclic graph are much harder than on trees. In practice it is often better to have a simple good algorithm, easy to maintain and extend, than a more efficient, but complex and hard to maintain and extend, one.

In this chapter we present a modular algorithm *(AQNU)*, whose complexity is almost quadratic in time, to solve nominal unification. It has been inspired by the *Martelli* and *Montanari* efficient first-order unification algorithm [34]. Martelli and Montanari use the *good but not linear set union Tarjan's algorithm* [50], more well known as the *Tarjan*'s union-find algorithm, to maintain equivalence classes. To maintain nominal equivalence classes, we need to adapt *Tarjan*'s union-find algorithm to nominal equivalence relations.

Section 7.1 presents such a nominal union-find algorithm and section 7.2 uses it to build a simple and efficient nominal unification algorithm. Concrete implementations in *Haskell* of the nominal union-find and the nominal unification algorithm are respectively in appendix A.7 and A.8.

7.1 A Nominal Union-Find Algorithm

The **disjoint set union problem** is to maintain a collection of disjoint sets $S_1, \ldots, S_n$ with two operations on them:

- `find(`x`)` to find the set S_i containing the element x.
- `union(`S_i, S_j`)` unite S_i and S_j into a single set: The sets S_i and S_j are removed from the collection and replaced by the set $S_i \cup S_j$.

Definition 49 *A **union-find** algorithm is an algorithm implementing the collection and the two functions of a disjoint set union problem.*

Tarjan's union-find algorithm 6 represents each set as a tree. The collection is called a **forest**, written $\mathbb{F}$. Edges are written $x \rightarrow_e y$ where x is the origin and y the destination of the edge. Each set is identified by the element at the root node of its tree called the representative of the set. `find(`x`)` returns the root node of tree containing x and `union(`x,y`)` merges the trees whose root nodes are `find(`x`)` and `find(`y`)`. To be efficient, the merging strategy of `union()` is to minimize the depth of the trees and every time `find(`x`)` is computed, the path from x to the root node is rewritten so that x points directly to it.

The collection of trees are implemented as an array arr_{uf} indexed by the elements of $\cup_{i=1}^{n} S_i$ so that :

$$arr_{uf}[x] = \begin{cases} \text{Indirect } y & \text{where } y \text{ is the father of } x \\ \text{Direct } r & \text{if } x \text{ is a root node} \end{cases}$$

where r is an integer called the **rank** of x. The rank represents a bound of the depth of the tree.

Tarjan's union-find algorithm is well suited to maintain first-order equivalence classes. But in nominal unification, two variables x and y in the same equivalence class do not directly represent α-equivalent terms but there exists a permutation π such that x and $\pi \cdot y$ do. We need to adapt Tarjan's algorithm to take account of these permutations and the freshness constraints they generate. We annotate every edge $x \rightarrow_e y$ by a permutation π, written $x \xrightarrow{\pi}_e y$, arr_{uf} becomes:

$$arr_{uf}[x] = \begin{cases} \text{Indirect } \pi \cdot y & \text{where } y \text{ is the father of } x \\ \text{Direct } (r, d) & \text{if } x \text{ is a root node} \end{cases}$$

```
1  find(x) = ;
2    if arr_uf[x] = Indirect y then
3      r = find(y) ;
4      arr_uf[x] := Indirect r ;
5      return r
6    else
7      return x
8    end
9  union(x,y) = ;
10   z_x = find(x) ;
11   z_y = find(y) ;
12   if z_x ≠ z_y then
13     Direct r_x = arr_uf[z_x] ;
14     Direct r_y = arr_uf[z_y] ;
15     if r_x = r_y then
16       arr_uf[z_y] = Direct (r_y + 1) ;
17       arr_uf[z_x] := Indirect z_y
18     else
19       if r_x < r_y then
20         arr_uf[z_x] := Indirect z_y
21       else
22         arr_uf[z_y] := Indirect z_x
23       end
24     end
25   end
```

Algorithm 6: Tarjan's union-find algorithm *(TUF)*

In the unification algorithm, $arr_{uf}[x] = \text{Indirect } \pi{\cdot}y$ will mean that $x \ \mathbb{R}_{?\approx?,\pi} \ y$. r is still the rank of the tree. The new value d is a data attached to the root of every tree.

Definition 50 *A nominal data set S_{dat} is a tuple of (G, f, p) where G is a group (its internal law is written $+$), f a group action from $\mathbb{P}(\mathcal{A})$ on G where $\mathbb{P}(\mathcal{A}) = \{A \mid A \subseteq \mathcal{A}\}$ and p is a group action of Π on G such that*

$$\forall A \subseteq S_{atms}, \pi \in \Pi, g \in G \quad p(\pi, f(A, g)) = f(\pi(A), p(\pi, g))$$

For the sake of simplicity, in the rest of this section, given a nominal data set S_{dat}, we use $g \in S_{dat}$ for $g \in G$, $A \# g$ for $f(A, g)$ and $\pi{\cdot}g$ for $p(\pi, g)$.

A suspended node is a pair of a permutation π and a node v written $\pi{\cdot}v$. The structure of arr_{uf} defines a relation $\overline{\mathbb{R}}$ over suspended nodes and nominal data. Let $\overline{\mathbb{R}}$ be the relation defined as:

$$id{\cdot}s \ \overline{\mathbb{R}} \ \pi{\cdot}e \Leftrightarrow s \xrightarrow{\pi}_e e \in \mathbb{F}$$
$$id{\cdot}r \ \overline{\mathbb{R}} \ id{\cdot}(\text{data } r) \text{ if } r \text{ is a root node}$$

and $\overline{\mathbb{R}}$ the reflexive, transitive, symmetric and invariant closure of $\overline{\mathbb{R}}$, ie the minimal relation satisfying:

- $\pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_s{\cdot}s$
- $\pi{\cdot}s \ \overline{\mathbb{R}} \ \pi'{\cdot}s \bigwedge id{\cdot}s \ \overline{\mathbb{R}} \ id{\cdot}d \Rightarrow d = ds(\pi, \pi') \# ds$
- $\pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_e{\cdot}e \Rightarrow \pi_e{\cdot}e \ \overline{\mathbb{R}} \ \pi_s{\cdot}s$
- $\pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_e{\cdot}e \bigwedge \pi_e{\cdot}e \ \overline{\mathbb{R}} \ \pi_f{\cdot}f \Rightarrow \pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_f{\cdot}f$
- $\forall \pi \quad \pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_e{\cdot}e \Rightarrow (\pi \circ \pi_s){\cdot}s \ \overline{\mathbb{R}} \ (\pi \circ \pi_e){\cdot}e$

where s, e, and f are nodes or nominal data and d is a nominal data.

The **Nominal Union-Find** algorithms 7 and 8 provide four functions: *find*, *union*, *data* and *putData* such that

- find$(\pi_s{\cdot}s)$ returns $(\pi_r{\cdot}(\text{root } s))$ where $\pi_s{\cdot}s \ \overline{\mathbb{R}} \ \pi_r{\cdot}(\text{root } s)$
- union$(\pi_{s_1}{\cdot}s_1, \pi_{s_2}{\cdot}s_2)$ does not return anything but merge the trees of s_1 and s_2 such that $\pi_{s_1}{\cdot}s_1 \ \overline{\mathbb{R}} \ \pi_{s_2}{\cdot}s_2$

- $data(\pi{\cdot}s)$ returns d such that $\pi{\cdot}s\ \overline{\mathbb{R}}\ d$
- $putData(\pi{\cdot}s, d)$ does not return anything puts stores d in such a way that $\pi{\cdot}s\ \overline{\mathbb{R}}\ d$

where s, s_1 and s_2 are nodes and d is a nominal data.

```
1  find(π_s·s) = ;
2    if arr_uf[s] = π_e·e then
3      π_r·r = find(π_e·e) ;
4      arr_uf[s] := π_r·r ;
5      return (π_s ∘ π_r)·r
6    else
7      return π_s·s
8    end
9  data(π_s·s) = ;
10   π_z·z = find(π_s·s) ;
11   Direct (r, d) = arr_uf[z];
12   return π_z·d
13 putData(π_s·s,d) = ;
14   π_z·z = find(π_s·s) ;
15   Direct (r, d') = arr_uf[z] ;
16   arr_uf[z] := Direct (r,π_z^{-1}·d)
```

Algorithm 7: Nominal Union-Find Algorithm *(NUF): Part 1*

Proposition 17 (Complexity) *The complexity of* `find()` *(resp.* `union()`*) in* NUF *is bounded by* $\mathcal{C}_{\mathcal{A}} \times C$ *where* C *is the complexity of* `find()` *(resp.* `union()`*) in* TUF*. So the amortized complexity in time of* `find()` *and* `union()` *is* $\mathcal{C}_{\mathcal{A}} \times ack^{-1}(\mid \cup_{i=1}^{n} S_i \mid)$ *where* $ack^{-1}()$ *is the* ***inverse Ackermann function****.*

Proof `union()` has the same structure and pass through the same steps in *TUF* and *NUF*. And any step in *TUF* that takes n time takes at most $\mathcal{C}_{\mathcal{A}} \times\ n$ in *NUF* $*$

Now that we have an algorithm well suited to maintain nominal relation equivalence classes, we can use it to develop a nominal unification algorithm.

```
1  union(π₁·s₁, π₂·v₂) = ;
2     (π_{z₁}·z₁) = find(π₁·s₁) ;
3     (π_{z₂}·z₂) = find(π₂·s₂) ;
4     Direct (r₁, d₁) = arr_{uf}[z₁] ;
5     Direct (r₂, d₂) = arr_{uf}[z₂] ;
6     if z₁ = z₂ then
7        putData(id·z₂ , ds(π_{z₁}, π_{z₂}) # d₁)
8     else
9        if r₁ = r₂ then
10          arr_{uf}[z₂] := Direct (r₂ + 1, d₂) ;
11          union(π_{z₁}·z₁, π_{z₂}·z₂)
12       else
13          if r₁ > r₂ then
14             union(π_{z₂}·z₂, π_{z₁}·z₁)
15          else
16             π₃ = π_{z₁}^{-1} ∘ π_{z₂} ;
17             arr_{uf}[z₁] := Indirect π₃·z₂ ;
18             putData(id·z₂ ,d₂ + π₃^{-1}·d₁)
19          end
20       end
21    end
```

Algorithm 8: Nominal Union-Find Algorithm *(NUF): Part 2*

7.2 An Almost Quadratic Nominal Unification Algorithm *(AQNU)*

In this section we present a nominal unification algorithm inspired by Martelli and Montanari's almost linear first-order unification algorithm [34]. The algorithm uses the same techniques of using multi-equations and maintaining, for each class of variable, a counter of the number of occurrence of this class still in the problem. Once again the nominal algorithm will have the same structure as the first-order algorithm.

Because any unification problem can be encoded in linear time and space to a single constraint $s \approx_\alpha t$, in the rest of this chapter, we consider that the input problem is $\{s \approx_\alpha t\}$. The set of atoms in the problem, written A_0 is defined as $A_0 = \mathrm{A}(s) \cup \mathrm{A}(t)$.

When we encounter $X \approx_\alpha \pi{\cdot}Y$ in the process of the algorithm, it means that if the problem has a solution then there exists a valid nominal equivalence relation $(\mathbb{R}_{?\approx?,}, \mathbb{F})$ such that $X \ \mathbb{R}_{?\approx?,\pi} \ Y$. X could be replaced by $\pi{\cdot}Y$ and Y by $\pi^{-1}{\cdot}X$, so X and Y can be considered more or less as the same variable. Replacing X by $\pi{\cdot}Y$ (or Y by $\pi^{-1}{\cdot}X$) would be too costly but maintaining the nominal equivalence classes of variables using the above nominal unification algorithm is cheap.

Definition 51 (Equations) *Here equations are $A \# t_1 \ {}_?{\approx}_? \ \ldots \ {}_?{\approx}_? \ t_n$ where A is a set of atoms and $t_1 \ {}_?{\approx}_? \ \ldots \ {}_?{\approx}_? \ t_n$ a multiset of terms.*

Definition 52 *The unification nominal data attached to a variable X in the nominal union-find algorithm are triples (o, A, lt) where o is an integer counting the number of occurrences of variables in the class of X in the problem, A is a set of atoms and lt is a list of terms $[t_1, \ldots, t_n]$ such that*

$$A \# X \ {}_?{\approx}_? \ t_1 \ {}_?{\approx}_? \ \ldots \ {}_?{\approx}_? \ t_n$$

is an equation of the problem. The set of unification nominal data is called S_{udat}

Proposition 18 S_{udat} *with*

- $(o_1, A_1, lt_1) + (o_2, A_2, lt_2) = (o_1 + o_2, A_1 \cup A_2, lt_1 + lt_2)$ *where* $lt_1 + lt_2$ *is the list forms of all the elements of* lt_1 *followed by all the elements of* lt_2
- $(0, \emptyset, [])$ *as neutral element (where* $[]$ *is the empty list)*

form a group. With the two following actions:

- $\forall \pi \in \Pi \quad \pi \cdot (o, A, [t_1, \ldots, t_n]) = (o, \pi(A), [\pi \cdot t_1, \ldots, \pi \cdot t_n])$
- $\forall A \in \mathcal{A} \quad A \# (o, A', lt) = (o, A' \cup A, lt)$

it forms a nominal data set.

The algorithm uses a global stack to store the equations of a nominal unification problem. Let *stack* be this stack and *emptyStack* be the empty stack.

The main function of the algorithm, `unify()`, is described in algorithm 9. It initializes the nominal union-find array, the global stack *stack* while the stack is not empty, pops an equation from it and treats it. When the stack is empty, it performs an occurrence check.

The normalizing function, `normalizeEquation()`, makes sure all terms in the equation are in head reduced normal form and that there is at most only one term that can be a suspended variable.

Definition 53 *At any time* t *in the execution of the problem, the problem* Pr_t *at time* t *is formed of the equations in the global stack, the ones in the nominal unification data and the constraints* $A \# X$ *and* $X \mapsto t$ *(which represents* $X \approx_\alpha t$*) in the output.*

Decomposition function Let $eq = A \# t_1 \ {}_?{\approx}_? \ \ldots \ {}_?{\approx}_? \ t_n$ be a normalized equation. If eq does not contains head variables, then it has to match one of this

```
   Input: Two terms s and t to unify
1  ;
2  unify(s,t) = ;
3    for X ∈ Vars(s) ∪ Vars(t) do
4    |  putData(Id·X, (o_X, ∅, [])) ;
5    |  where o_X is the number of occurrences of X in ∅ # s ?≈? t
6    end
7    stack:= emptyStack ;
8    push(∅ # s ?≈? t) on stack;
9    while stack is not empty do
10   |  eq = pop(stack) ;
11   |  eq' = normalizeEquation(eq) ;
12   |  if eq' has a head variable then
13   |  |  storeEquation(eq')
14   |  else
15   |  |  treatDecompose(eq')
16   |  end
17   end
18 occurCheck()

   Input: an equation A # t_1 ?≈? ... ?≈? t_n
19 ;
20 normalizeEquation(A # t_1 ?≈? ... ?≈? t_n) = ;
21   for 0 ≤ ilen do
22   |  replace t_i by \overline{t_i}^h
23   end
24   while there are t_i = π_i·X_i and t_j = π_j·X_j with i ≠ j do
25   |  union (π_i·X_i , π_j·X_j) ;
26   |  remove t_j from the equation ;
27   |  decrement the counter of occurrence of X_j by one
28   end
29 return the remaining equation
```

Algorithm 9: Unification and normalizing functions

rules:

$$
\begin{array}{ll}
(\mathit{AtomsRule}) & A \# a \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; a \\
 & \Rightarrow \;\; \emptyset \\
 & \text{with } a \text{ and atom and } a \notin A \\
(\mathit{CstRule}) & A \# f \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; f \\
 & \Rightarrow \;\; \emptyset \\
 & \text{with } f \text{ a constant} \\
(\mathit{PairRule}) & A \# (t_{1_1}, t_{1_2}) \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; (t_{n_1}, t_{n_2}) \\
 & \Rightarrow \;\; \{A \# t_{1_1} \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; t_{n_1}, \\
 & \quad\;\; A \# t_{1_2} \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; t_{n_2}\} \\
(\mathit{AbsRule}) & A \# [a_1]t_1 \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; [a_n]t_n \\
 & \Rightarrow \;\; \{(A \cup \{a_1 \ldots a_n\}) \setminus \{a_1\} \# (a_1\ a_1)\cdot t_1 \;{}_?{\approx}_?\; \ldots \;{}_?{\approx}_?\; (a_1\ a_n)\cdot t_n\}
\end{array}
$$

Then resulting equations are put in the stack.

If *eq* does not match any of these rules then the problem has no solution and the program raise an error.

The function taking *eq* as input, putting the resulting equation in the stack or raising an error is called *decompose*

Storing equation If the normalized equation does contain a term $\pi_X \cdot X$, then it can not be decomposed but it can be store in the unification nominal data of X. The function which stores such equations is called `storeEquation()` and described in algorithm 10.

OccurCheck The function *occurCheck* checks that the data attached to every class is $(0, \emptyset, [])$. If not, it means that the problem does contains a loop and so has no solution. Thus, if one of the data is not $(0, \emptyset, [])$ the function raise an error, otherwise returns nothing.

Proposition 19 (Correctness) *If the problem raised an error, then the problem does not have any solution. Otherwise, the output of the algorithm is a most general solution of the problem.*

Proof All the steps of the algorithm transform the problem into an equivalent one. So the set of the solutions of the problem is preserved. $*$

```
Input: A normalized equation eq = A # π·X ?≈? t_1 ?≈? ... ?≈? t_n
1  storeEquation(A # π·X ?≈? t_1 ?≈? ... ?≈? t_n) = ;
2    d = data((π·X)) ;
3    (o, A', [t_{n+1}, ..., t_m]) = d ;
4    which represents the equation A' # π·X ?≈? t_{n+1} ?≈? ... ?≈? t_m;
5    the two equations can be combined into :;
6    A'' # t_1 ?≈? ... ?≈? t_m with A'' = A ∪ A';
7    if o = 1 then
8      this is the only occurrence of X in the problem;
9      then the class of X is a root class and ca be processed;
10     if n = 0 then
11       then equation is the freshness constraints A'' # π·X;
12       output the freshness constraint π^{-1}(A'') # X
13     else
14       output the substitution X ↦ π^{-1}·t_1 ;
15       push(A'' # t_1 ?≈? ... ?≈? t_m) on stack
16     end
17     putData(π·X, (0, ∅, []))
18   else
19     putData(π·X, (o − 1, A'', [t_1, ..., t_m]))
20   end
```

Algorithm 10: Equation storing function

7.2.1 Complexity

Let d be a nominal data (o, A, lt). Representing lt as a list is too costly because of the number of times we need to apply a permutation to its elements and appending two lists. For example, let $l_1, \ldots, l_4$ be 4 lists and let us consider $l_1 + l_2 + l_3 + l_4$. Doing it in one shot is linear in $|l_1| + |l_2| + |l_3| + |l_4|$ but computing $l_5 = l_1 + l_2$, then $l_6 = l_3 + l_4$ and finally do $l_5 + l_6$ is not. So performing lazily these operation is more efficient. In this section we implement a better structure to represent lt with lazy permutations and lazy appending.

This structure is the algebraic datatype:

$Tree\ a$	$=$	Nil	the empty tree
	$\mid$	$Singleton\ a$	the leaf a
	$\mid$	$Node\ (Tree\ a)\ (Tree\ a)$	a node with two children
	$\mid$	$Remap\ \Pi\ (Tree\ a)$	

With this structure we can merge two trees t_1 and t_2 in constant time with $Node\ t_1\ t_2$. And we can lazily apply a permutation to all the elements of a tree in constant time with $Remap\ \pi\ t$. If we have to apply many permutations to a tree, permutations will be composed together and only then the resulting permutation will be applied on the elements of the tree, which means only one tree and term traversal.

When we need to get the list of terms, we flatten the tree, applying permutations and giving back the list of leaves.

Input: A nominal tree t

```
1 flatten(t) = flattenAux(id , t);
2   where;
3     flattenAux(id , t) = switch (π,t) do
4         case (π,Nil) → return [] ;
5         case (π,Singleton a) → return [π·a] ;
6         case (π,Node t1 t2) → Node flattenAux(π,t1) flattenAux(π,t2)
          ;
7         case (π,Remap π' t) → flattenAux((π ∘ π'),t)
8     endsw
```

Proposition 20 *The complexity of the unification algorithm with the lazy structure is at worse* $\mathcal{C}_{\mathcal{A}} \times (\mid F(s) \mid + n_{\pi}(s) + \mid F(t) \mid + n_{\pi}(t)) \times \ ack^{-1}(\mid \mathcal{X} \mid)$ *where* $ack^{-1}()$ *is the* ***inverse Ackermann function****.*

Proof Once again, when we forget all nominal details in the nominal algorithm, we get a first-order unification algorithm. The complexity in time of this algorithm is $(\mid F(s) \mid + \mid F(t) \mid) \times \ \mathrm{ack}^{-1}(\mid \mathcal{X} \mid)$ and traversing a permutation application node takes $\mathcal{C}_{\mathcal{A}}$ time. $\mathrm{ack}^{-1}(\mid \mathcal{X} \mid)$ comes from the nominal union-find algorithm. $*$

Remark 5 *When* $\mid F(s) \mid + \mid F(t) \mid$ *is close to* $\mid F(s) \mid + n_{\pi}(s) + \mid F(t) \mid + n_{\pi}(t)$*, with mutable arrays as permutations ans sets implementation the nominal algorithm becomes linear in time and space.*

Unlike to *QNU*, *AQNU* is convenient to use in practice: its input is simply two nominal terms, it is modular and quite functional. That is why it has been implemented in the *Haskell* library whereas *QNU* has been implemented as an isolated algorithm.

Chapter 8

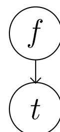
Nominal Unification via Graph Rewriting

In this chapter we present a polynomial nominal unification algorithm based on graph rewriting. The algorithm takes a directed acyclic graph representing the problem as input and performs a series of transformations on it until either raising an error or reaching a solution.

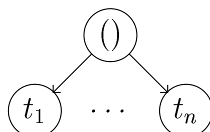
The encoding of nominal terms is a bit different from what was presented in chapter 5. The two differences are in the syntax of functions and tuples:

Definition 54

$f\ t$:



$(t_1, \ldots, t_n)$:

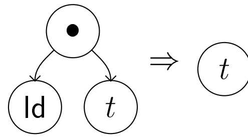


Definition 55 (Graph rewriting rule) *A graph rewriting rule is a pair of nominal graphs, written $l \Rightarrow r$, where leaves may be labelled by (graph) metavariables*

$s, t, \ldots$, and where l has exactly one root. All the metavariables occurring in r must also occur in l, and we will also assume that each metavariable occurs at most once in l.

A graph rewriting rule may have several roots in the right-hand side; rules with multiple roots in the right-hand side will be used in Section 8.1.1.

Example 3 *The following is a graph rewriting rule, using a metavariable t.*



To apply a graph rewriting rule $l \Rightarrow r$ to a graph G, we first create a copy of r and add it to G, and then pointers towards the instance of l in G are redirected towards the copy of r. Occurrences of metavariables in r are replaced by a pointer to the corresponding subgraph in G (thus, they are shared). Other pointers are not affected. If a node is no longer needed (i.e. there is no path from the root node) we garbage-collect it.

We now define graph reduction more precisely:

Definition 56 (Graph reduction) *The one-step reduction relation generated by a graph rewriting rule $l \Rightarrow r$ is defined as a set of pairs $G \Rightarrow G'$ generated as follows:*

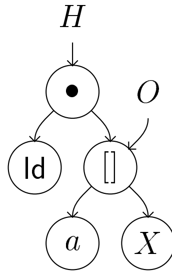
When a subgraph g in G matches the left-hand side l of a rule (that is, g can be obtained by replacing the metavariables in l by graphs; we say that g is a redex*), G' is obtained by:*

1. *Adding to G a copy of the right-hand side r, where the metavariables t in r are replaced by pointers to the corresponding instance of t in g.*

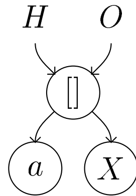
2. *Redirecting all the pointers to the root of g towards the root of r (the pointers are duplicated if r has more than one root, or erased if r is empty); in the special case in which r is just a metavariable t, then pointers to the root of g are updated to point directly to the instance of t in g. The use of a rule with multiple roots in the right-hand side is only permitted if the parent nodes of g have variable arity.*

3. Nodes that have no incoming edges are erased (garbage collected), except for the node Pr (the root of the problem). In particular, the root of g will be garbage-collected since all pointers to it are redirected, but the other nodes in g may still be of use.

Example 4 *The rule in Example 3 can be applied to the graph of $\mathsf{Id}\cdot[a]X$ (below we assume there are incoming edges H and O):*



since we can identify an instance of the left-hand side in this graph (replacing t by the graph representing $[a]X$). After one step of rewriting we obtain the graph representation of $[a]X$:



where the nodes $\bullet$ and Id have been garbage collected, and the pointer H has been updated, so that it points to the instance of t (O is not affected).

The following sections specify a correct and complete graph rewriting algorithm to solve nominal problems. We first consider $\approx_\alpha$ constraints in Section 8.1.1, and show how the graph representing a set of $\approx_\alpha$ constraints can be reduced to $\perp$ if there is no solution, or to a graph representing a substitution and set of freshness constraints. Section 8.1.2 then deals with freshness constraints: we specify an algorithm to transform a set of freshness constraints into an equivalent freshness context. We will show that both algorithms are polynomial in time and space.

8.1 A polynomial Algorithm via graph rewriting

8.1.1 A graph rewriting algorithm to solve $\approx_\alpha$ constraints

We will present an algorithm to solve α-equality constraints using three kinds of graph rewriting rules: **normalisation rules** to simplify the graph representing the problem (eliminating trivial constraints, identity permutations, etc), **$\approx_\alpha$-rules** to solve equality constraints, and **neutralisation of permutations** to propagate lazy permutations in case no $\approx_\alpha$-rule can be applied.

In the sequel, we assume that we start with a graph representing a nominal problem and we only consider graphs that are obtained by reduction of the initial graph. We show below that all the graph rewriting rules we will define transform the graph into a graph representing an equivalent problem.

8.1.1.1 Normalisation of the graph

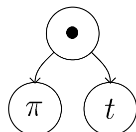
Definition 57 *A skeleton node is a node labelled by a term constructor that is not $\bullet$ or a permutation. A value node is either an atom, a set of atoms, or unit (i.e., a 0-ary tuple).*

Skeleton nodes are therefore value nodes, variables, abstractions, tuple constructors and function nodes. For example, in the graph representing the term $\pi{\cdot}a$, there is only one skeleton node, which is the one labelled by a.

The following function W will be used to distinguish suspended variables on variable nodes from other suspended variables, ignoring permutations.

Definition 58 (Weight) *The function $W : N \to \{0, 1\}$ assigns a weight (0 or 1) to each node n. It is defined by cases on n:*

- *If n is a variable, $W(n) = 0$.*
- *If n is neither a variable nor an $\bullet$, $W(n) = 1$.*
- *If n is $\bullet$ and it is the root of a subgraph as shown below, then $W(n) = W(t)$.*

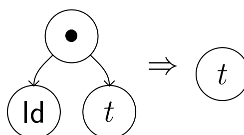


The computation of the weight of a node can be done in polynomial time in the size of the graph, since in the worst case it requires the traversal of a path in the graph (and the graph is acyclic).

To reduce the graph we apply a set of *normalisation rules*, which are partitioned into three groups: simplification, ordering, and compacting.

Simplification rules

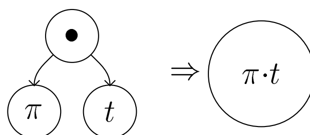
- Rule *Id-Perm* eliminates identity permutations:



  Note that when applying this rule, no node is created since t was already present in the left-hand side. According to Definition 56, all pointers to the • node in the left are redirected towards the node t, and the • node is garbage collected. The node Id will be garbage collected if there are no other pointers to it.

  It is clear that this rule preserves all the solutions of the problem, but notice that in the case in which t is a variable, we are replacing $\mathsf{Id}{\cdot}X$ by X, which strictly speaking is not a term. We could impose a condition that the rule applies only when t is not a variable, but in practice it is better to use the rule also with variables and take this fact into account when we read the problem represented by the graph (see Section 8.1.1.2).

- Rule *Apply-Perm* applies permutations on *value* nodes. We give below a rule scheme which encompasses three kinds of rules (one for each kind of value node, see Definition 57):



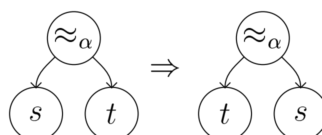
if t is a value node.

Note that, according to Definition 56, when we reduce a graph using *Apply-Perm*, all pointers to the $\bullet$ node are redirected towards the node $\pi{\cdot}t$, and the $\bullet$ node is garbage collected. The node π instead may have other pointers to it (it will only be erased if there are no pointers to it).

This rule preserves the solutions of the problem, since it replaces a suspended permutation by its result.

Ordering rules

- Rule *Order-Unif* puts the heaviest side of an equation on the right, using the notion of weight defined above (see Definition 58):

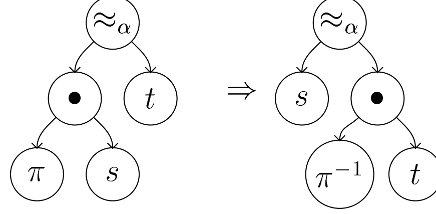


if $W(s) > W(t)$

Only the edges of the graph are affected by this rule, no nodes need to be created or erased.

Clearly, solutions are preserved since $\approx_\alpha$ is commutative.

- Rule *Order-Perm* moves permutations towards the right-hand side of equations:

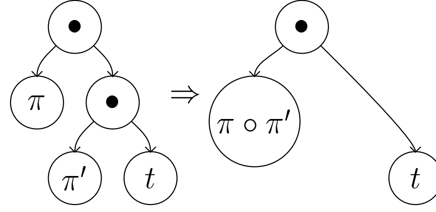


if $W(s) \leq W(t)$ and s is a *skeleton node* .

In other words, this rule transforms a constraint of the form $\pi{\cdot}s \approx_\alpha t$ into $s \approx_\alpha \pi^{-1}{\cdot}t$, which is equivalent (see [20, 51]).

Compacting rules

- Rule *Consec-Perm* composes consecutive permutations:



if t is a *skeleton node* .

If there are more than two consecutive permutations, the condition in this rule ensures that they are composed bottom-up (because t cannot be an application of permutation).

8.1.1.2 Properties

To obtain a graph representation of a nominal unification problem, in the sequel we assume that the translation function described in Definition 40 is used, with a further optimisation: suspended variables of the form $\mathsf{Id}{\cdot}X$ may simply be represented as a node labelled by X.

Proposition 21 (Correctness of normalisation) *The normalisation rules are correct; more precisely: if a graph G representing a unification problem Pr reduces to a graph G', then*

1. *G' is the representation of a problem Pr' (except that a term $Id{\cdot}X$ may be represented simply by a node X); and*

2. *Pr' is equivalent to Pr.*

Proof

1. We consider each normalisation rule in turn. Since the translation function from problems to graphs is inductive, we obtain the problem Pr' associated to G' by replacing in Pr the subproblem corresponding to the left-hand side of the rule by the problem represented by the right-hand side. The only special case is *Id-Perm*, which may replace $\mathsf{Id}{\cdot}X$ by X.

2. The normalisation rules can be justified as follows:

   - *Id-Perm* and *Apply-Perm* are correct by definition of permutation (see Section 4.2).
   - *Order-Unif* is correct because $\approx_\alpha$ is commutative.
   - *Order-Perm* is justified by the equivalence: $\pi{\cdot}s \approx_\alpha t \Leftrightarrow s \approx_\alpha \pi^{-1}{\cdot}t$.
   - *Consec-Perm* is justified by the fact that $\pi{\cdot}(\pi'{\cdot}t)$ and $(\pi\circ\pi'){\cdot}t$ represent the same nominal term.

$*$

To show that the normalisation rules are terminating (i.e., reduction sequences are finite), we introduce the notion of position in the graph.

Definition 59 (Positions) *A position in a graph G representing a unification problem Pr is a string of integers. We use Λ to denote the empty string, which will be associated to the root of the graph, and use the notation $p.i$ to denote the string containing all the elements of p followed by i. The node at position p in G, written $G|_p$, is defined as follows:*

- *$G|_\Lambda$ is the root of the problem, the only node labelled by Pr.*

- *$G|_i$ is the i-th root, that is, the root of the i-th constraint.*

- *$G|_{p.i}$ is the node attached to the i-th port of the node $G|_p$.*

The weight of a position is the weight of the node at that position.

Due to sharing, the same node may be associated to several positions as the following example illustrates.

Example 5 *In the graph G representing the problem*

$$[a]a \approx_\alpha [b]b, (a\ b)\cdot X \approx_\alpha X$$

given in Example 2, the node at position 2 is labelled by $\approx_\alpha$ (that is, $G|_2$ is the root of the second $\approx_\alpha$ constraint), and the node at position 2.2 (i.e., the second child of the node $\approx_\alpha$ at position 2) is labelled by the variable X. Since X is shared, this node is also at position 2.1.2.

All the normalisation rules, except *Order-Unif*, preserve the weight of positions; more precisely:

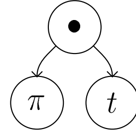
Proposition 22 *Let G be a nominal graph, p a position in G, and G' the graph obtained by applying a normalisation rule $l \Longrightarrow r$ different from* Order-Unif *on G. If $G'|_p$ exists, and p corresponds to a position that exists in both l and r, then $W(G|_p) = W(G'|_p)$.*

Proof By inspection of the rules. The weight of a node at a position p in the left-hand side is the same as the one in position p in the right-hand side. $*$

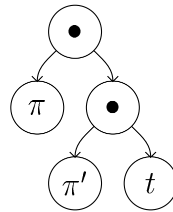
Intuitively, this property says that the graph positions that survive reduction, that is, the positions that are in the left and right-hand sides of the rule, maintain their weight, except when *Order-Unif* is used.

We will now show that the rewrite system defined by the simplification, ordering and compacting rules is terminating. Unfortunately we cannot use a simple interpretation based on sizes, because the rule *Consec-Perm*, although having a left-hand side which is bigger than the right-hand side, may increase the size of the graph. Indeed, this rule creates a node $\pi \circ \pi'$, and the nodes in the left-hand side may be shared, so cannot be erased. We will define the following patterns, which include the left-hand sides of the rules above and some additional graphs that will be useful to define a decreasing measure:

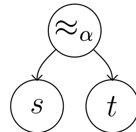
- Pattern *Apply-Perm*:



- Pattern *Consec-Perm*:

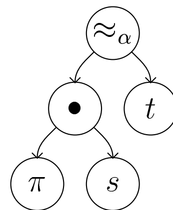


- Pattern *Order-Unif*:



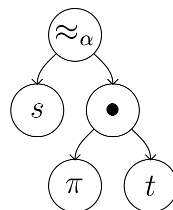
where $W(s) > W(t)$

- Pattern *Order-Perm-Left*:



where $W(s) \leq W(t)$

- Pattern *Order-Perm-Right*:



where $W(s) > W(t)$

We associate to each pattern a number of points:

Apply-Perm	: 1 point
Consec-Perm	: 1 point
Order-Unif	: 1 point
Order-Perm-Left	: 3 points
Order-Perm-Right	: 3 points

and to each node n in a graph the sum of the points of all the patterns that match the subgraph rooted by n (a subgraph may match several patterns). More precisely:

Definition 60 (Reduction Points) *Let G be a graph and n a node in G. Let $RedPts(G, n)$ be the sum of the points of patterns matched by the subgraph with root n in G. $RedPts(G) = \sum_{n \in G} RedPts(G, n)$.*

Proposition 23 *For any node n in a graph G, $RedPts(G, n) \leq 4$.*

Proof Let n be the root of a subgraph that matches a pattern. If n is a $\bullet$ node, it may only match *Apply-Perm* and *Consec-Perm*: 2 points. If n is a $\approx_\alpha$ node it may match *Order-Perm-Left*, or *Order-Unif* and *Order-Perm-Right*, so 4 points at most. Otherwise, n matches no pattern so has zero points. $*$

Proposition 24 *Let G be a graph and G' be the graph obtained by applying a normalisation rule on G. $RedPts(G') < RedPts(G)$.*

Proof Let n be the root of a subgraph that matches a rule; we say that n is the node on which the rule is applied. We consider each rule in turn:

- *Apply-Perm* and *Id-Perm*: n is deleted, and the *value node* $\pi{\cdot}t$ created by *Apply-Perm* has zero points. $RedPts()$ may have changed for some nodes: Let n' be a node of G' and G (so $n' \neq n$). Either n' was not affected by the rewrite step, or it is a node in a position belonging both to the left- and righ-hand sides. *Apply-Perm* and *Id-Perm* preserve the weights of those positions (Proposition 22), and pattern conditions depend only on weight conditions so a condition is satisfied after the application of the rule if and

only if it was satisfied before. If a pattern is matched by n' in G' so it was in G. So $RedPts(G', n) \leq RedPts(G, n)$, hence $RedPts(G') < RedPts(G)$ (because n has points and n is not in G').

- *Order-Unif*: n in G' cannot match the patterns *Apply-Perm* or *Consec-Perm* (because it is $\approx_\alpha$), and *Order-Perm-Right* (because of weight conditions). It matches the pattern *Order-Perm-Left* in G' if and only if it matched *Order-Perm-Right* in G and it matched *Order-Unif* in G. So $RedPts(G', n) = RedPts(G, n) - 1$. The rule neither deletes nor creates any node, the subgraph of the other nodes is the same, and so are the points. Thus, $RedPts(G') = RedPts(G) - 1$.
- *Order-Perm*: n cannot match any pattern in G', but in G it matched the pattern *Order-Perm-Left*. Nodes $\bullet$ and π^{-1} are created, with at most 2 points. The subgraph of the other nodes remains unchanged, so do the points. Hence, $RedPts(G') \leq RedPts(G) - 3 + 2 = RedPts(G) - 1$.
- *Consec-Perm*: n matches only the pattern *Apply-Perm* in G' (not *Consec-Perm* because t is a *skeleton node*) and it matched the patterns *Apply-Perm* and *Consec-Perm* in G. The rule *Consec-Perm* preserves weights, so conditions satisfied in G' are also satisfied in G. Let n' be a parent of n, if n is an $\approx_\alpha$ node and matches patterns *Order-Unif* and *Order-Perm-Left*, or *Order-Perm-Right*, so it was in G. If n' is a $\bullet$ node, and matches *Consec-Perm* or *Apply-Perm*, so it was in G. Otherwise n matches nothing. Other subgraphs are not modified. Hence $RedPts(G') = RedPts(G) - 1$.

$*$

As a corollary, we obtain the termination property for reduction:

Corollary 1 *The graph rewriting system defined by the simplification, ordering and compacting rules is terminating. The normalisation process terminates in at most $RedPts(G)$ steps.*

We can give an upper bound on $RedPts(G)$, that is, a bound on the number of reduction steps needed to compute a normal form (i.e., an irreducible graph):

Proposition 25 *$RedPts(G) \leq 2Nb(G, \bullet) + 4Nb(G, \approx_\alpha)$ where $Nb(G, \bullet)$ is the number of $\bullet$ nodes in G and $Nb(G, \approx_\alpha)$ is the number of $\approx_\alpha$ nodes in G.*

Proof $\bullet$ nodes have at most 2 points, $\approx_\alpha$ nodes at most 4, other nodes have no points. $*$

Corollary 2 *The number of normalisation steps for a graph G representing a nominal problem is at most $2Nb(G, \bullet) + 4Nb(G, \approx_\alpha)$.*

We will now compute a bound on $Nb(G, \bullet)$. This will be needed to evaluate the cost of the unification algorithm. First we introduce some notation:

Definition 61 *We denote by $App(G)$ the set of $\bullet$ nodes in the graph G (so $Nb(G, \bullet) = |App(G)|$), $Ports_{sk}(G)$ (resp. $Ports_{\approx_\alpha}(G)$) denotes the set of all the ports of skeleton nodes (resp. $\approx_\alpha$ nodes) in G, $Ports_{sk,\alpha}(G)$ denotes the set of all the ports of skeleton and $\approx_\alpha$ nodes in G, and $Ports^\bullet_{sk,\alpha}(G)$ denotes the set of all the ports of skeleton nodes and $\approx_\alpha$ nodes in G that are attached to an $\bullet$ node.*

Proposition 26 *Let G be a normal form of a graph representing a problem Pr. Then $Nb(G, \bullet) = |App(G)| \leq |Ports_{sk,\alpha}(G)|$.*

Proof Let $f : Ports^\bullet_{sk,\alpha} \to App(G)$ be the function that for each port in $Ports^\bullet_{sk,\alpha}(G)$ returns the $\bullet$ node attached to it. We can show that f is surjective as follows: Each $\bullet$ node n has at least one parent (otherwise it is garbage collected); let n_p be a parent of n, then n_p is not an $\bullet$ node (otherwise we would have two consecutive applications of permutations, which contradicts the fact that G is in normal form). Thus, n_p is either a skeleton node or a $\approx_\alpha$ node. Let p be the port of n_p to which n is attached. Then p is in $Ports^\bullet_{sk,\alpha}(G)$ and $f(p) = n$. Therefore $|App(G)| = |Im(f)| \leq |Ports^\bullet_{sk,\alpha}(G)| \leq Ports_{sk,\alpha}(G)|$. $*$

Note that in the unification algorithm it is not necessary to reduce the full graph to normal form. In fact, in our implementation we normalise locally, reducing only the subgraphs that are relevant for the application of an $\approx_\alpha$ rule. The local normalisation process terminates too, and it has a smaller cost.

8.1.1.3 $\approx_\alpha$-rules

We will now solve α-equality constraints using $\approx_\alpha$-rules which apply to normalised graphs. Before giving the rules, it is useful to characterise graphs in normal form.

Proposition 27 *The normal forms of graphs representing nominal terms are the graph representation of terms generated by the following grammar:*

$$
\begin{array}{lcll}
T_{nf} & ::= & \pi\cdot(T_{nf.cons}|T_{nf.var}) \mid T_{nf.np} & \pi \neq \mathsf{Id} \\
T_{nf.np} & ::= & T_{nf.cons} \mid T_{nf.val} \mid T_{nf.var} & \\
T_{nf.cons} & ::= & f(T_{nf}) \mid (T^1_{nf}, \ldots, T^n_{nf}) \mid [a]T_{nf} & n \neq 0 \\
T_{nf.val} & ::= & a \mid () & \\
T_{nf.var} & ::= & X & \\
T_{nf.npvar} & ::= & T_{nf.cons} \mid T_{nf.val} &
\end{array}
$$

Here nf stands for normal form, np for non permutation, cons for term-constructor, val for value, var for variable and npvar for not variable-or-permutation.

Proof A normal form is irreducible, so it cannot contain Id, consecutive permutations, or permutations acting on value nodes. $*$

Proposition 28 *The normal forms of $\approx_\alpha$-constraints are generated by:*

$$GR_{\approx_\alpha nf} \quad ::= \quad (T_{nf.var} \approx_\alpha T_{nf}) \mid (T_{nf.npvar} \approx_\alpha (\pi\cdot \mid \epsilon) T_{nf.npvar})$$

Proof By inspection of the rules: A constraint in normal form is irreducible, so its left-hand side cannot be a permutation application. If we have a variable right-hand side (possibly with one permutation) then we must have a variable on the left-hand side. Note that we can have at most one application of a permutation in the right-hand side (otherwise the rule *Consec-Perm* would apply). $*$

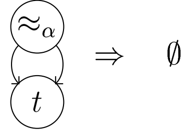
Definition 62 *Two term nodes n_1 and n_2 are incompatible if it is impossible to unify a graph rooted by n_1 with a graph rooted by n_2.*

In other words, two nodes n_1, n_2 are incompatible if they are labelled by two different, non-variable, term constructors (e.g., different atoms a, b; an atom and an abstraction; two different function symbols; a function and an abstraction; etc.).

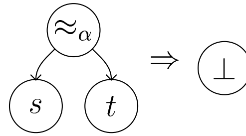
We divide the rules into two groups: simplification and propagation rules. Simplification rules have priority.

$\approx_\alpha$-simplification rules

- Rule *Id-Unif* erases trivial equations:

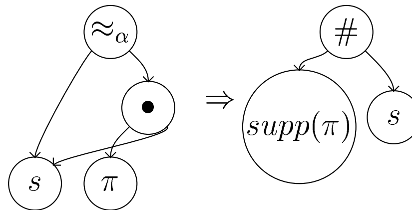


- Rule $\bot$-*Unif* detects failure due to incompatibility:



if s and t are incompatible

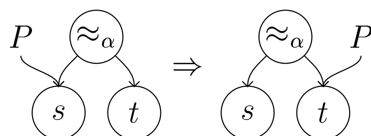
- Rule *Same-Term* solves equations where both sides involve the same term s, with a suspended permutation on the right-hand side:



Note that $supp(\pi)$ may be empty; we deal with freshness constraints later.

$\approx_\alpha$-propagation rules

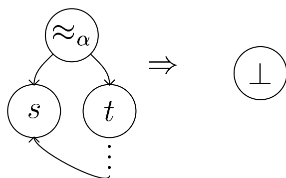
Each application of a propagation rule will be combined with the meta rule *Redirect*:



if no node in t has s as a child.
Here P is any parent node of s.

The aim of this rule is to move all the pointers to the left-hand side of the equation (except the pointer from the root of the equation) towards the right-hand side. The idea is that if $s \approx_\alpha t$, then we can replace any occurrence of s by t. However, this operation should not create a cycle; in other words, if s and t are α-equivalent then s cannot be a subterm of t. We call the latter a *generalised occur check*.

If *Redirect* cannot be applied because a node in t has s as a child, then the equation has no solution, and we rewrite it to $\perp$:

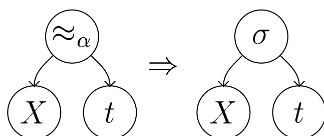


if a node in t has s as a child (generalised occur-check).

We will assume that each time an $\approx_\alpha$-propagation rule R has been selected to be applied to an equation, the *Redirect* rule is applied *before* applying R. Note that the *Redirect* rule does not change the terms s and t, only pointers to s are redirected to an α-equivalent term.

The propagation rules are:

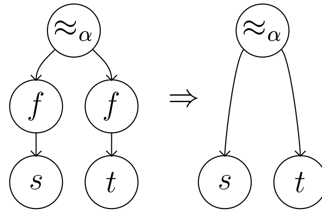
- Rule *Subst*:



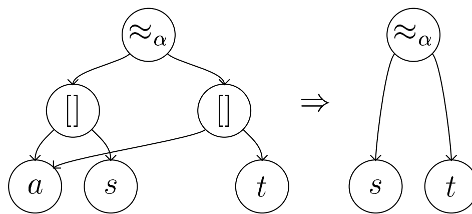
There is no need to perform an occur check in the rule *Subst*, since it is done by *Redirect*.

Recall that once this rule is selected to be applied, all pointers to X in the left are redirected to t by the *Redirect* rule (provided X does not occur in t), which corresponds to applying the substitution $[X \mapsto t]$ in the graph.

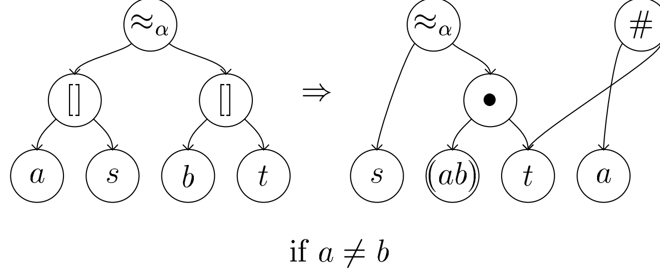
- Rule *Fct-Unif*:



- Rule *Abs-Unif-Same*:

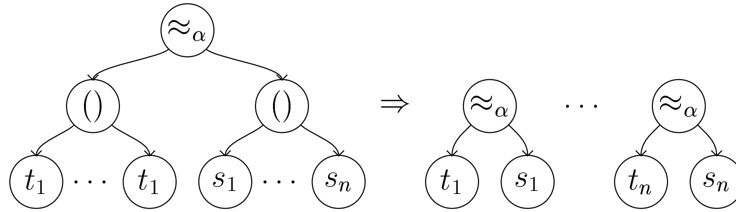


- Rule *Abs-Unif-Diff*:



if $a \neq b$

- Rule *Tpl-Unif*:



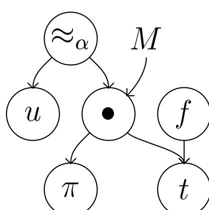
Here if the tuple has arity 0, the right-hand side is empty (i.e., the constraint is eliminated).

Notice that the rules *Abs-Unif-Diff* and *Tpl-Unif* rewrite a constraint into a set of constraints. The graph in the right-hand side has two roots in the case of *Abs-Unif-Diff*, and may have zero or more roots in the case of *Tpl-Unif*. According to the definition of graph rewriting, in a reduction step using one of these rules, each pointer to the root of the redex is replaced by a set of pointers to the roots of the right-hand side. Since the parent node of $\approx_\alpha$ is Pr, which has a variable arity, these rules are well-defined.

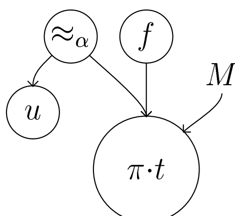
Before showing that the $\approx_\alpha$-simplification and $\approx_\alpha$-propagation rules are sound, we introduce a rule to deal with suspended permutations that prevent the application of the rules above.

8.1.1.4 Computing permutations: Neutralisation

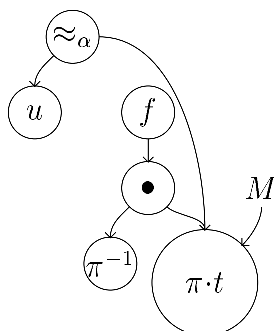
In order to be able to match the left-hand side of an $\approx_\alpha$-rule, we may need to (partially) compute a permutation in a normalised graph. We will do this in a lazy way, moving the permutation down the term only the minimum that is needed to match an $\approx_\alpha$-rule. However, due to sharing, such propagations have to be computed carefully. Consider the following example:



where M represents any node pointing towards $\bullet$. If we apply directly π on t, we obtain:

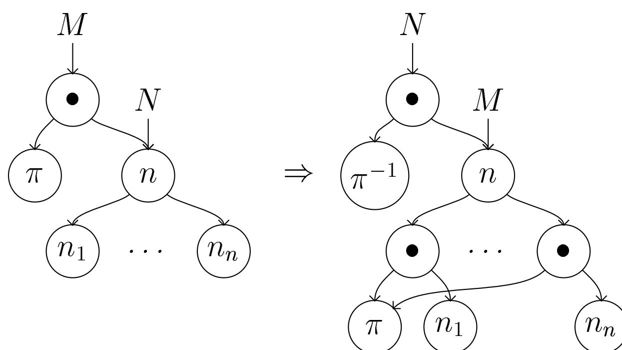


which is incorrect ($f\ t$ becomes $f\ \pi{\cdot}t$). However, since $t = \pi^{-1}{\cdot}(\pi{\cdot}t)$, we can apply π on t and redirect parents of $\bullet$ towards $(\pi{\cdot}t)$ and parents of t towards $(\pi^{-1}{\cdot}(\pi{\cdot}t))$ as follows:



This is the basis of the *neutralisation* rule defined below.

Definition 63 (Neutralisation rule) *Let n be a non leaf node. The* Neutralisation *of $\pi{\cdot}n$ is defined by:*



The neutralisation rule will only be applied to $\pi{\cdot}t$ when it occurs immediately below an $\approx_\alpha$ node, as shown in Figure 8.1 below. This rule is used only to trigger the application of an $\approx_\alpha$-rule in a normalised graph, by bringing a term constructor up to match the left-hand side of an $\approx_\alpha$-rule.

In the graph shown in Figure 8.1, if no $\approx_\alpha$ rule can be applied, the neutralisation rule applies. It is easy to see that after applying this rule, the node $\approx_\alpha$ will point to s and to the root of t.

Proposition 29 *If G is a graph representing a nominal problem, and G normalises to G', then either there are no $\approx_\alpha$ nodes in G' or an $\approx_\alpha$- rule can be applied, possibly preceded by an application of Neutralisation.*

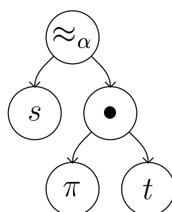


Figure 8.1: Graphs where neutralisation may be applied

Proof Suppose the graph G' contains an $\approx_\alpha$ node. By Proposition 28, this node is the root of a subgraph G'' representing a constraint $s \approx_\alpha t$, and we can distinguish two cases:

If t is not of the form $\pi \cdot v$, then using Propositions 27 and 28 we can check that there is an $\approx_\alpha$ rule for each kind of normal form (i.e., G'' matches the left-hand side of an $\approx_\alpha$-rule).

If G'' is a graph as depicted in Figure 8.1 (note that s and t must be different, otherwise the graph would not be normalised since rule *Same-Term* applies) then after applying neutralisation, $\approx_\alpha$ will point to s and to the root of t, and again using Proposition 28 we can check that the graph can be reduced by an $\approx_\alpha$-rule. $*$

Proposition 30 (Correctness of $\approx_\alpha$ rules) *When applied to a normalised graph representing a nominal problem, all the $\approx_\alpha$ rules, as well as Neutralisation, preserve the solutions.*

Proof First notice that the $\approx_\alpha$-simplification rules deal with 'simple' equations (constraints of the form $t \approx_\alpha t$, which are trivially solvable and can be eliminated; equations between incompatible terms, which have no solution and thus are replaced by $\perp$; and equations of the form $t \approx_\alpha \pi \cdot t$, which have solutions only if the atoms affected by π are fresh in t). It is easy to see that the $\approx_\alpha$-propagation rules follow the format of the rules given in Section 4. The only interesting case is *Subst*, which replaces the $\approx_\alpha$ node by a σ node representing a substitution, or by $\perp$ (depending on the result of the occur check). In the first case, the meta-rule Redirect performs the substitution by updating the relevant pointers. Rule *Redirect* generates $\perp$ only if the equation has no solution. $*$

8.1.1.5 Putting all together

In the first phase of the algorithm to solve nominal unification problems, we solve $\approx_\alpha$ constraints in the graph representation of the problem, using Normalisation rules, $\approx_\alpha$-rules, and the Neutralisation rule. We deal with freshness constraints in the second phase of the algorithm. More precisely, the first phase of the algorithm can be described as follows: we start by normalising the graph G_0 representing

the initial nominal problem and then we apply cycles of $\approx_\alpha$-rules followed by normalisation, until no $\approx_\alpha$ constraints are left (if necessary, we use the Neutralisation rule before applying an $\approx_\alpha$-rule). The algorithm is given below in pseudo-code:

$\approx_\alpha$ algorithm:

Normalise G
while G has at least one $\approx_\alpha$ node **do**
  if G is irreducible for $\approx_\alpha$-rules **then**
    Apply *Neutralisation* on G
  end if
  Apply an $\approx_\alpha$-rule on G
  Normalise G
end while

If at any point $\perp$ is generated, we raise an exception since this means that the problem has no solution.

Below we show that the algorithm above terminates and all the $\approx_\alpha$ nodes are eventually eliminated, obtaining $\perp$ or the graph representation of a substitution and a set of freshness constraints. We analyse the cost of the algorithm in terms of number of iterations, and reduction steps.

8.1.1.6 An upper bound on the number of iterations

We use a measure based on the arities of nodes (note that the number of nodes does not necessarily decrease with each rule application, because of sharing and because rule *Tpl-Unif* creates several $\approx_\alpha$ nodes).

Definition 64 *Let G be a nominal graph and n a node in G.*
$ResSize(G) = \sum_{n \in G} ResSize(n)$*, where $ResSize(n)$ is defined by:*

$$ResSize(n) = \begin{cases} 0 & \text{if } n \text{ is a } \bullet,\ \sigma,\ \#\ \text{node} \\ 2 \times arity(n) & \text{if } n \text{ is a tuple node} \\ arity(n) & \text{otherwise} \end{cases}$$

For example, $ResSize(\approx_\alpha) = 2$, $ResSize(\bot) = 0$.

Remark 6 *$ResSize(n)$ depends only on n (not G).*

Proposition 31 *Let G be a graph and G' be the graph obtained by applying an $\approx_\alpha$-rule on G. Then $ResSize(G') < ResSize(G)$.*

Proof Let n be the root of the subgraph on which the rule is applied, we consider each rule:

- *Id-Unif*, *$\bot$-Unif*, *Subst*, *Same-Term* delete n (that is, an $\approx_\alpha$-node of arity 2), and create only nodes of null $ResSize()$. Therefore $ResSize(G') \leq ResSize(G) - ResSize(n) = ResSize(G) - 2$.

- *Fct-Unif*, *Abs-Unif-Same* and *Abs-Unif-Diff* only create nodes of null $ResSize$. They do not erase the two roots (f or [], resp.) because there may be other pointers to these nodes. Let l (resp. r) be the left (resp. right) child of n. *Redirect*, which is combined with each rule, transforms parents of l into parents of r so that after applying the rule l becomes useless and is garbage-collected. Note that $ResSize(l) > 0$, except if l is a 0-ary function, in which case the $\approx_\alpha$-node n is erased. Hence $ResSize(G') < ResSize(G)$.

- *Tpl-Unif*: Let m be the arity of the tuple. If $m = 0$ then the $\approx_\alpha$-node n is erased. Otherwise, this rule creates $m-1$ $\approx_\alpha$-nodes but as in the previous case the left tuple-node (l) becomes useless and so is erased. $ResSize(G') = ResSize(G) + (m-1) * ResSize(n) - ResSize(l) = ResSize(G) + 2m - 2 - 2m = ResSize(G) - 2$.

$*$

Proposition 32 *Let G be a graph and G' be the graph obtained by applying a normalisation or neutralisation rule on G. Then $ResSize(G') \leq ResSize(G)$.*

Proof For each rule, only nodes of null $ResSize$ are created. $*$

Proposition 33 *There are at most $ResSize(G_0)$ iterations of the while-loop.*

Proof Normalisation rules do not increase $ResSize(G)$, and neither does *Neutralisation*, by Proposition 32. Also, $\approx_\alpha$-rules decrease $ResSize(G)$ by Proposition 31, and by Proposition 29 one of these rules is applicable in each iteration of the while-loop, hence $ResSize(G)$ decreases. $*$

As a corollary of Proposition 31 we can also deduce that the $\approx_\alpha$-rules terminate on their own (but in the algorithm we apply only one step of $\approx_\alpha$-reduction in each iteration of the while-loop).

8.1.1.7 An upper bound on the number of normalisation steps

Using Proposition 2, we know that in each iteration of the while-loop there are at most $2Nb(G, \bullet) + 4Nb(G, \approx_\alpha)$ normalisation steps. We will now compute an upper bound which depends only on the initial graph G_0.

Definition 65 $MaxArity(G) = max\{arity(n)|n \in G\}$

Proposition 34 *Let G be a nominal graph and G' be the graph obtained by applying a normalisation, $\approx_\alpha$, or neutralisation rule on G. Then*

1. $MaxArity(G') \leq MaxArity(G)$, *and*
2. $|Ports_{sk}(G')| \leq |Ports_{sk}(G)|$.

Proof For the first part: note that each node on the right-hand side of a rule has an arity which is less than or equal to the arities of the nodes in the left-hand side. The second part is a consequence of the fact that rules do not create skeleton nodes. $*$

Proposition 35 *Let G_0 be the initial graph representing a problem, and G a reduced version of G_0 at any moment in the algorithm.*

$$Nb(G, \approx_\alpha) \leq Nb(G_0, \approx_\alpha) + Nb(G_0, ()) \times MaxArity(G_0) = K_\alpha.$$

Proof The only rule that adds $\approx_\alpha$ nodes in the graph is *Tpl-Unif*. It consumes a tuple node (denoted by $()$) and creates at most $MaxArity(G_0)$ $\approx_\alpha$-nodes. No rule creates tuple nodes, hence at most $Nb(G_0, ()) \times MaxArity(G_0)$ $\approx_\alpha$ nodes can be created. $*$

Proposition 36 *Let G be a normalised graph.*

$$Nb(G, \bullet) \leq |Ports_{sk}(G_0)| + 2K_\alpha.$$

Proof By Proposition 26, we know that in a normalised graph, $Nb(G, \bullet) \leq |Ports_{sk,\alpha}(G)| = |Ports_{sk}(G)| + |Ports_\alpha(G)|$. Now using Propositions 34 and 35 we deduce $Nb(G, \bullet) \leq |Ports_{sk}(G_0)| + 2K_\alpha$. $*$

Proposition 37 *Let G be a normalised graph. By applying an $\approx_\alpha$-rule and/or Neutralisation, at most $MaxArity(G_0) + 1$ $\bullet$-nodes are created.*

Proof By inspection of the rules we see that at most $MaxArity(G) + 1$ nodes are created, and using Proposition 34 we deduce that $MaxArity(G) + 1 \leq MaxArity(G_0) + 1$. $*$

Proposition 38 *At any step in the iteration:*

1. $Nb(G, \bullet) \leq |Ports_{sk}(G_0)| + 2K_\alpha + MaxArity(G_0) + 1 = K_\bullet$.
2. *Let $|G|$ denote the size of the graph.*

$$|G| \leq K_\alpha + 2K_\bullet + ResSize(G_0) + Nb(G_0, sk) = K_{||}$$

Proof

1. In each iteration, we start with a normalised graph G and apply at most one $\approx_\alpha$-rule (possibly with an application of *Neutralisation*), before normalising again. Since G is normalised, by Proposition 38 $Nb(G, \bullet) \leq |Ports_{sk}(G_0)| + 2K_\alpha$, and by applying a neutralisation and $\approx_\alpha$ rule, at most $(MaxArity(G_0) + 1)$ $\bullet$-nodes are created (by Proposition 37). There are then at most $|Ports_{sk}(G_0)| + 2K_\alpha + MaxArity(G_0) + 1$ $\bullet$-nodes.

2. We count the number of nodes of each kind: #-nodes and σ-nodes are only created by $\approx_\alpha$-rules, one at a time, hence $Nb(G, \#) + Nb(G, \sigma) \leq ResSize(G_0)$. Since every permutation is attached to a $\bullet$-node, $Nb(G, \pi) \leq Nb(G, \bullet)$. Hence:

$$\begin{array}{rcl} |G| & = & Nb(G, \approx_\alpha) + (Nb(G, \#) + Nb(G, \sigma)) + Nb(G, \mathrm{sk}) + \\ & & Nb(G, \bullet) + Nb(G, \pi) \\ & \leq & K_\alpha + ResSize(G_0) + Nb(G_0, \mathrm{sk}) + 2K_\bullet \end{array}$$

*

The constant K_{N} defined in the proposition below is an upper bound on the number of normalisation steps in each iteration of the algorithm.

Proposition 39 *There are at most $2K_\bullet + 4K_\alpha = K_N$ normalisation steps in each iteration of the while-loop.*

Proof By Proposition 2, when we normalise the graph in each iteration of the while-loop there are at most $2Nb(G, \bullet) + 4Nb(G, \approx_\alpha)$ steps, hence (by Propositions 35 and 38) we obtain an upper bound: $2K_\bullet + 4K_\alpha = K_{\mathrm{N}}$. *

We have therefore an upper bound for the number of iterations, and the number of normalisation steps, which shows that the algorithm is indeed terminating. Below we will analyse the cost of each operation, in order to establish the complexity of the algorithm, but we can already show its correctness.

Proposition 40 (Soundness and completeness of $\approx_\alpha$-solving) *The algorithm defined above is sound and complete.*

Proof Consequence of Propositions 21 and 30, together with Propositions 39 and 33, which entail the termination of the loop. *

At the end of the first phase of the unification algorithm no $\approx_\alpha$-constraints remain, and provided that $\perp$ has not been generated, the second phase can start. At this point, the only children of the node Pr will be either # nodes or σ nodes introduced by the $\approx_\alpha$ rules.

8.1.1.8 Cost of the rules

Let A_0 be the set of atoms in the initial problem. A permutation π is implemented as an AVL tree of pairs $(a, \pi(a))$ for $a \in supp(\pi)$.

The algorithm does not introduce new atoms in the problem, so at each step there are at most $|A_0|$ atoms in G, and in particular in permutations in G. Thus, applying π to a requires searching the value $\pi(a)$ in π's AVL, which can be done in $O(log(|A_0|))$[1]. Inversion and union of permutations are also $O(|A_0| \times log(|A_0|))$.

Below we calculate the cost of applying each normalisation rule:

[1] We follow the notations of [49], to which we refer the reader for a definition of O.

- Rule *Apply-Perm*: Since we use AVLs to represent permutations, computing $\pi \cdot t$ for a value node t has a cost $O(log(|A_0|))$.

- Rule *Id-Perm* eliminates identity permutations, its cost is constant: $O(1)$

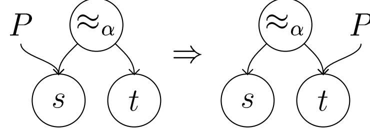
- Rule *Order-Unif* also has a constant cost: $O(1)$

- Rule *Order-Perm* involves the computation of the inverse of a permutation. This can be done in $O(|A_0| \times log(|A_0|))$ with AVLs.

- Rule *Consec-Perm* composes consecutive permutations. Again the cost is $O(|A_0| \times log(|A_0|))$ with AVLs as permutations.

Hence, applying a normalisation rule is at most in $O(|A_0| \times log(|A_0|))$.

We now turn our attention to $\approx_\alpha$ rules. First note that some of these rules require a redirection of all parents of a node to another node. This operation is linear in the number of parents, so linear in $|G|$. The generalised occur check (i.e., checking whether a node in t has s as a child, see below) is also linear in the size of t, therefore linear in $|G|$.



if no node in t has s as a child (generalised occur-check),
otherwise the right-hand side is $\perp$.

Cost: at most $|G|$

- Rule *Id-Unif* erases trivial equations, its cost is constant: $O(1)$

- Rule $\perp$-*Unif* detects incompatibility failure, again its cost is constant: $O(1)$

- Rule *Same-Term* requires the computation of the support of π. The cost is $O(|A_0| \times log(|A_0|))$ with permutations as AVLs.

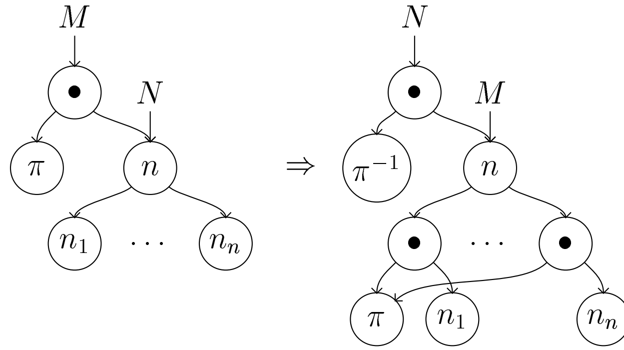
- Rule *Subst* creates a substitution, its cost is constant: $O(1)$

- Rules *Fct-Unif*, *Abs-Unif-Same* and *Abs-Unif-Diff* involve simple pointer operations, with constant cost $O(1)$.
- Rule *Tpl-Unif* has a cost proportional to the number of elements in the tuple: $O(\text{arity of the tuple})$, that is $O(\text{MaxArity}(G_0))$.

Hence, the cost of applying an $\approx_\alpha$ rule is at most $O(max(|G|+\text{MaxArity}(G_0), |A_0| \times log(|A_0|))) \leq O(|G| + \text{MaxArity}(G_0) + |A_0| \times log(|A_0|))$.

Note that $\text{MaxArity}(G_0) \leq K_{||}$ and $|G|$ is bounded by $K_{||}$, therefore the cost is at most $O(|A_0| \times log(|A_0|) + K_{||})$.

Finally, we consider an application of the Neutralisation rule:



Cost: $O(|G| + \text{MaxArity}(G_0) + |A_0| \times log(|A_0|))$

The application of an $\approx_\alpha$ rule or *Neutralisation* is in $O(|G| + \text{MaxArity}(G_0) + |A_0| \times log(|A_0|))$, hence again it is $O(|A_0| \times log(|A_0|) + K_{||})$.

Proposition 41 *This phase is* $O(ResSize(G_0) \times (K_{||} + K_N \times |A_0| \times log(|A_0|)))$.

Proof For each iteration, by Proposition 39, there are at most K_N steps of normalisation, each one in $O(|A_0| \times log(|A_0|))$. In addition, one $\approx_\alpha$-rule is applied, and at most one neutralisation step, each in $O(|A_0| \times log(|A_0|) + K_{||})$. The cost of an iteration is then $O(K_\text{N} \times |A_0| \times log(|A_0|) + 2(|A_0| \times log(|A_0|) + K_{||})$, or, simplifying, $O(K_\text{N} \times |A_0| \times log(|A_0|) + K_{||})$.

Because there are, by Proposition 33, at most $ResSize(G_0)$ iterations, the loop is in $O(ResSize(G_0) \times (K_{||} + K_\text{N} \times |A_0| \times log(|A_0|)))$. The normalisation phase before entering the loop is also $O(K_\text{N} \times |A_0| \times log(|A_0|))$ and hence does not change the complexity of the algorithm. $\ast$

8.1.2 Freshness constraints

The algorithm to solve nominal problems has two phases, as mentioned above. The first phase deals with $\approx_\alpha$ constraints, as described in the previous section. Assuming the problem has a solution, at the end of the first phase we are left with a set of freshness constraints of the form $A\#t$ (where A is a set of atoms, see rule *Same-Term* in Section 8.1.1.3) and a substitution of the form $[X_i \mapsto t_i]_{i\in I}$. We describe in this section an algorithm to check that a set of atoms is fresh in a term t. The idea is to traverse the graph representing the term, and annotate term nodes with the set of atoms for which freshness has already been checked (to avoid repeated computations).

Definition 66 (Annotated terms) *Given a graph representing a term t, we annotate each term node by a set of atoms, and denote the set of atoms attached to the node n by $\xi(n)$. Initially, all term nodes are annotated with the empty set of atoms.*

Definition 67 *Let G be the graph representing a problem Pr. Let G' be the subgraph representing a freshness constraint $A\#t$ and G'' the graph representing t where n is its root node.*

One step of freshness checking for G' is defined by the following rule:

$$(\textit{fresh}) \qquad A \# n \quad \Rightarrow \quad \begin{cases} \textit{fresh}_l(A \setminus \xi(n) \# n) \\ \textit{with side effect } \xi(n) := \xi(n) \cup A \end{cases}$$

where the function $fresh_l$ is defined as follows (we assume $A \neq \emptyset$):

$$\begin{array}{lcl}
fresh_l(\emptyset \# n) & = & \emptyset \\
fresh_l(A \# f(n)) & = & A \# n \\
fresh_l(A \# (n_i)_{i=1\dots m}) & = & (A \# n_i)_{i=1\dots m} \\
fresh_l(A \# [a]n) & = & (A \setminus a) \# n \\
fresh_l(A \# \pi{\cdot}n) & = & \pi^{-1}(A) \# n \\
fresh_l(A \# X) & = & A \#_! X \\
fresh_l(A \# a) & = & \begin{cases} \emptyset & a \notin A \\ \bot & a \in A \end{cases}
\end{array}$$

Note that when $fresh_l$ is applied to $A \# X$ we generate a new constraint $A \#_! X$ which will be part of the solution to the problem.

Definition 68 *Let G be the graph representation of a nominal problem Pr. Assume G' is the graph obtained after running the first phase of the algorithm and G' does not contain $\perp$. The algorithm to check freshness constraints simply applies the rule (fresh) defined above until the graph is irreducible.*

Proposition 42 (Irreducible forms) *Irreducible graphs do not contain $\#$ nodes. Only freshness constraints of the form $A \#_! X$ may remain.*

Proof If there is a $\#$ node, rule (fresh) can be applied. $*$

Proposition 43 *The freshness check is correct.*

Proof The rule (fresh) memorises the freshness information that has already been checked (or is being checked), to prevent checking multiple times that the same atom is fresh in the same term. Formally, checking $A \# t$ is equivalent to checking $(a \# t)_{a \in A}$. So checking $A \# t, B \# t$ is equivalent to checking $(A \backslash B) \# t, (A \cap B) \# t, (B \backslash A) \# t, (B \cap A) \# t$. Although we have the constraint $(B \cap A) \# t$ twice, we only need to check it once, so we do $A \# t, (B \setminus A) \# t$.

The definition of $fresh_l$ follows the inductive definition of the freshness predicate (see Section 4). When the propagation reaches a variable $(A \# X)$ we save this resolved constraint as $A \#_! X$ so that (fresh) can no longer be applied. $*$

Proposition 44 *The freshness check terminates in at most $2 \times |G| \times |A_0|$ steps of rewriting.*

Proof We compute the maximum number of steps by counting the number of times that an atom is checked for freshness in a given term node in this phase. Note that if $A = \emptyset$ or if we have already checked all atoms in A, i.e. $A \setminus \xi(n) = \emptyset$, the constraint is simply removed. Let $|G|_{tn}$ be the number of term nodes in G, term nodes are never created in this phase. Thanks to annotations there is for each term node n at most $|A_0|$ steps of (fresh) on n where $A \setminus \xi(n) \neq \emptyset$ so $|A_0| \times |G|_{tn}$ application of (fresh) where $A \setminus \xi(n) \neq \emptyset$. An empty set can either be created in the previous phase, or be generated by $fresh_l(A \# [a]n)$; hence there

are at most $Nb(G, \#) + |A_0| \times |G|_{tn}$ applications of (fresh) where $A \setminus \xi(n) = \emptyset$. Therefore, we have at most $Nb(G, \#) + 2 \times |A_0| \times |G|_{tn} \leq 2 \times |A_0| \times |G|$ steps of rewriting. $*$

Before considering the cost of the freshness check, we can state the following property, which is a direct consequence of Propositions 40 and 43:

Proposition 45 *The algorithm to solve nominal problems by*

1. *transforming the problem into a graph;*
2. *applying the first phase to eliminate $\approx_\alpha$ constraints;*
3. *applying the second phase to eliminate freshness constraints;*

is sound and complete.

8.1.3 Cost

We now analyse the complexity of the second phase in the algorithm (freshness checking). We start by analysing the cost of $fresh_l$ for each case:

$fresh_l(\emptyset \# n)$	:	$O(1)$
$fresh_l(A \# f(n))$	:	$O(1)$
$fresh_l(A \# (n_i)_{i=1...m})$	:	$O(m)$
$fresh_l(A \# [a]n)$	:	$O(log(\|A_0\|))$ since sets are represented as AVL trees
$fresh_l(A \# \pi \cdot n)$	:	$O((\|A_0\| \times log(\|A_0\|))^2)$; we discuss this case below
$fresh_l(A \# X)$	:	$O(1)$
$fresh_l(A \# a)$	:	$O(log(\|A_0\|))$ due to the use of AVL trees

The cost of $fresh_l(A \# \pi \cdot n)$ is due to the computation of π^{-1} in $O(|A_0| \times log(|A_0|))$ and the application of π^{-1} to A in $O(|A_0| \times log(|A_0|))$ too.

Proposition 46 *Freshness checking is $O(|G| \times |A_0|^3 \times log(|A_0|)^2)$.*

Proof A rewrite step of (fresh) consists of one step of $fresh_l$ and the update of $\xi(n)$, which is a union of sets and can be done in $O(|A_0| \times log(|A_0|))$. We deduce that each step of rewriting with these rules is at most in $O((|A_0| \times log(|A_0|))^2)$. Since there are at most $2 \times |G| \times |A_0|$ steps of rewriting by Proposition 44, the result follows. $*$

8.1.4 Total cost in time for the unification algorithm

An upper bound on the total cost of the unification algorithm can be obtained by adding the cost of the phase of resolution of $\approx_\alpha$ constraints and the phase of freshness checking.

Proposition 47 *The following is an upper bound for the unification algorithm:*

$$O(ResSize(G_0) \times (K_{||} + K_N \times |A_0| \times log(|A_0|)) + K_{||} \times |A_0|^3 \times log(|A_0|)^2)$$

Proof The input graph G for the freshness algorithm is the output of the $\approx_\alpha$ resolution algorithm, hence $|G| \leq K_{||}$. The result is then obtained by adding the two bounds (see Propositions 41 and 46). $*$

To simplify the formula above, we use the following properties:

Proposition 48

1. $ResSize(G_0) = O(|G_0| \times MaxArity(G_0))$
2. $K_\alpha = O(|G_0| \times MaxArity(G_0))$
3. $K_\bullet = O(|G_0| \times MaxArity(G_0))$
4. $K_{||} = O(|G_0| \times MaxArity(G_0))$
5. $K_N = O(|G_0| \times MaxArity(G_0))$

Proof

1. For each node n in G_0, $ResSize(n) \leq 2MaxArity(G_0)$ then $ResSize(G_0) \leq 2|G_0| \times MaxArity(G_0)$.
2. $K_\alpha = Nb(G_0, \approx_\alpha) + Nb(G_0, ()) \times MaxArity(G_0)$
   $\leq |G_0| + |G_0| \times MaxArity(G_0) = O(|G_0| \times MaxArity(G_0))$.
3. $K_\bullet = |Ports_{sk}(G_0)| + 2K_\alpha + MaxArity(G_0) + 1$.
   $|Ports_{sk}(G_0)| \leq |G_0| \times MaxArity(G_0)$ because the arity of each node is less than or equal to $MaxArity(G_0)$.
4. $K_{||} = K_\alpha + 2K_\bullet + ResSize(G_0) + Nb(G_0, \mathrm{sk})$ and each element in the addition is bounded by $O(2|G_0| \times MaxArity(G_0))$.

5. $K_{\mathrm{N}} = 2K_{\bullet} + 4K_{\alpha}$ and again each element in the addition is bounded by $O(2|G_0| \times MaxArity(G_0))$.

$*$

Proposition 49 (Cost in time for the unification algorithm) *The following is also an upper bound for the unification algorithm:*

$$O(K^4 \times log^2(K))$$

where $K = |G_0| \times MaxArity(G_0)$

Proof Consequence of Propositions 47 and 48, using the fact that $|A_0| \leq |G_0| \leq K$. $*$

We have presented 3 algorithms to solve nominal unification. *QNU* is the most efficient but as it has been seen, it is not easy to use in into a larger program. Though being a bit less efficient that *QNU*, *AQNU* is simple and can easily be part of a larger program, which is nice in practice. The graph rewriting algorithm is the less efficient but shows an interesting way of solving nominal equivalence.

QNU and *AQNU* extend the corresponding algorithms on first-order terms to nominal terms without modifying the structure of the algorithm but only adding name management. This management adds an extra cost which depends on the implementation of atoms, permutations and sets of atoms. Generally, the complexity of the extended algorithm is the product of $\mathcal{C}_{\mathcal{A}}$ and the complexity of the original algorithm.

Finally, we presented a concrete implementation of *QNU* and *AQNU*.

Part III

α-equivalence, Matching and Rewriting

Let $Pr = \{s \approx_\alpha t\}$ be a nominal α-equivalence (resp matching) problem. Pr can be solved by considering it as a nominal unification problem whose solutions (Δ, σ) have to satisfy that the domain of σ ($dom(\sigma)$ is empty (resp. $dom(\sigma) \cap \mathrm{V}(s) = \emptyset$). However, in some cases, some optimisations can lead to more efficient nominal α-equivalence and matching algorithms.

This part presents a very modular algorithm which adapts itself to the input problem to reach better efficiency. This algorithm is based on the one we presented in [10]. The implementation has been completely rethought and is now using the ideas presented in the chapter 3 and streams. This implementation is better than the previous one on many points:

- it is as close as possible to the abstract description of the algorithm.
- it completely abstracts the actual implementation of permutations and sets.
- it is extremely modular.
- it is much easier to read, understand and to use.

The implementation in *Haskell* is in appendices A.6, A.9 and A.10.

Chapter 9 introduces the notions of **environment** and presents the algorithm in an abstract way. Then chapter 10 describes the implementation.

Chapter 9

A modular algorithm to check α-equivalence and solve matching constraints

Efficient implementations of nominal unification rely on the use of lazy permutations: permutations are only pushed down a term when this is needed to apply a transformation rule, and then, they are only pushed one level down the term. Since lazy permutations may grow (they accumulate), in order to obtain an efficient algorithm we will compose the swappings eagerly. The key idea is to work with a single *current* permutation, represented by an **environment**.

The algorithms to check α-equivalence constraints and to solve matching problems (linear or non-linear) will be built in a modular way. The core module is composed of four phases and is common to both algorithms; only the final phase will be specific to matching or α-equivalence.

9.1 Environments

We begin by introducing the notion of an environment. Environments will be associated to terms and used to store a permutation and a set of atoms.

Definition 69 *Let s and t be terms, π be a permutation, and A be a finite set of atoms. An* ***environment*** *ξ is a pair (π, A). We denote by ξ_π the permutation*

*(resp. ξ_A the set of atoms) of an environment. We write $s \approx_\alpha \xi \Diamond t$ to represent $s \approx_\alpha \xi_\pi \cdot t,\ \xi_A \# t$ and call $s \approx_\alpha \xi \Diamond t$ an **environment constraint**.*

For example, the environment constraint $X \approx_\alpha \xi \Diamond b$ where $\xi_\pi = (a\ b)$ and $\xi_A = \{a\}$ represents the problem $X \approx_\alpha (a\ b) \cdot b, a \# b$.

Definition 70 *An **environment problem** Pr is either $\perp$ or has the form*

$$s_1 \approx_\alpha \xi_1 \Diamond t_1, \ldots, s_n \approx_\alpha \xi_n \Diamond t_n$$

where $s_i \approx_\alpha \xi_i \Diamond t_i$ ($1 \leq i \leq n$) are environment constraints. We will sometimes abbreviate it as $(s_i \approx_\alpha \xi_i \Diamond t_i)_1^n$.

Definition 71 *The problems defined in Section 4 will be called **standard** to distinguish them from environment problems (standard problems have no environments). The **standard form** of an environment problem is obtained by applying the rule:*

$$s \approx_\alpha \xi \Diamond t \Longrightarrow s \approx_\alpha \xi_\pi \cdot t,\ \xi_A \# t$$

as many times as possible. We denote by $[\![Pr]\!]$ the standard form of an environment problem Pr. Thus:

$$[\![s_1 \approx_\alpha \xi_1 \Diamond t_1, \ldots, s_n \approx_\alpha \xi_n \Diamond t_n]\!] =$$
$$s_1 \approx_\alpha (\xi_1)_\pi \cdot t_1, (\xi_1)_A \# t_1, \ldots, s_n \approx_\alpha (\xi_n)_\pi \cdot t_n, (\xi_n)_A \# t_n$$

This rule is terminating because it consumes a $\Diamond$ each time, without creating any. There are no superpositions, so the system is locally confluent and because it terminates it is confluent [38]. Therefore the standard form of an environment problem exists and is unique.

Definition 72 *The **solutions** of an environment problem are the solutions of its standard form (see Section 4). A problem $\perp$ has no solutions. Two environment problems are **equivalent** if their standard forms are equivalent, i.e., have the same solutions.*

As a consequence of Definition 72, two equivalent environment problems have the same set of solutions.

The set of reduction rules on standard problems given in Section 4 transform a problem into an equivalent one (i.e., solutions are preserved, see [51]). Below we will give a set of transformation rules for environment problems, and we will prove that a step of rewriting $Pr \Longrightarrow Pr'$ on an environment problem can be simulated by some steps of reduction on its standard form using the rules in Section 4, that is, $[\![Pr]\!] \stackrel{*}{\Longrightarrow} [\![Pr']\!]$. Therefore Pr and Pr' are equivalent.

It is easy to translate standard problems into environment problems, as shown below.

Definition 73 *The translation Env from standard problems into environment problems is inductive, with base cases:*
$Env(s \approx_\alpha t) = s \approx_\alpha \xi \Diamond t$ where $\xi = (\mathsf{Id}, \emptyset)$ and
$Env(A \# t) = t \approx_\alpha \xi \Diamond t$ where $\xi = (\mathsf{Id}, A)$.

This transformation is linear in time and in space. Therefore, from a (log-)linear algorithm solving environment problems we can derive a (log-)linear algorithm to solve standard problems.

In the following sections, we restrict our attention to checking α-equivalence constraints and solving matching problems. In the latter case, in environment constraints $s \approx_\alpha \xi \Diamond t$, the term t will not be instantiated and variables in s and t are disjoint. If right-hand sides t are ground terms, we will say that the problem is *ground*, and *non-ground* otherwise.

9.2 Core algorithm

The core of the algorithm transforms an environment problem $(s_i \approx_\alpha \xi_i \Diamond t_i)_1^n$ into an equivalent standard problem of the form $(X_i \approx_\alpha t_i)_{i \in I}$, $(A_j \# X_j)_{j \in J}$. There are four phases in the core algorithm. The first one reduces $\approx_\alpha$ constraints, by propagating ξ_i over t_i. The second phase eliminates permutations on the left-hand side of constraints, and the third phase reduces freshness constraints, also by propagating ξ_i over t_i. Finally, the fourth phase computes the standard form of the resulting problem.

Definition 74 *Let Pr be an environment problem. We denote by $\overline{Pr}^{\,c}$ the result of applying the core algorithm on Pr.*

Below we describe each phase of the core algorithm.

Phase 1 The input is an environment problem $Pr = (s_i \approx_\alpha \xi_i \lozenge t_i)_1^n$, which we will reduce by applying the following transformation rules.

$$Pr,\quad a \quad\approx_\alpha \xi\lozenge t \Longrightarrow \begin{cases} Pr & \text{if } a = \xi_\pi{\cdot}t \text{ and } t \notin \xi_A \\ \bot & \text{otherwise} \end{cases}$$

$$Pr,\ (s_1,\ldots,s_m) \approx_\alpha \xi\lozenge t \Longrightarrow \begin{cases} Pr,\ (s_i \approx_\alpha \xi\lozenge u_i)_1^m & \text{if } t = (u_1,\ldots,u_m) \\ \bot & \text{otherwise} \end{cases}$$

$$Pr,\quad f\ s \quad\approx_\alpha \xi\lozenge t \Longrightarrow \begin{cases} Pr,\ s \approx_\alpha \xi\lozenge u & \text{if } t = f\ u \\ \bot & \text{otherwise} \end{cases}$$

$$Pr,\quad [a]s \quad\approx_\alpha \xi\lozenge t \Longrightarrow \begin{cases} Pr,\ s \approx_\alpha \xi'\lozenge u & \text{if } t = [b]u \\ \bot & \text{otherwise} \end{cases}$$

where $\xi' = ((a\ \xi_\pi{\cdot}b) \circ \xi_\pi,\ (\xi_A \cup \{\xi_\pi^{-1}{\cdot}a\}) \setminus \{b\})$ in the last rule, and a, b could be the same atom.

We show below that the rules transform an environment problem into another equivalent environment problem.

The environment problems that are irreducible for the rules above will be called **phase 1 normal forms** or **ph1nf** for short.

Proposition 50 (Phase 1 normal forms) *The normal forms for phase 1 rules are either $\bot$ or $(\pi_i{\cdot}X_i \approx_\alpha \xi_i\lozenge s_i)_1^n$ where s_i are nominal terms and $n \geq 0$.*

Proof If the left-hand side of an $\approx_\alpha$ constraint is an atom, function application, tuple or abstraction, then the constraint can be reduced using the phase 1 rules. Note that if the problem is ground, the ph1nf is either $\bot$ or empty. $\ast$

Proposition 51 (Correctness) *Let Pr be an environment problem and assume $Pr \Longrightarrow Pr'$ using the phase 1 rules. Then Pr and Pr' are equivalent.*

Proof To prove that Pr and Pr' are equivalent we have to show that their standard forms have the same solutions. We prove this by showing that if a problem reduces to $\bot$ then it has no solutions, and that if $Pr \Longrightarrow Pr' \neq \bot$ then there is a problem Q such that $[\![Pr]\!] \overset{*}{\Longrightarrow} Q$ using standard reduction rules

(given in Section 4), and Q is equivalent to $[\![Pr']\!]$. Since standard reduction steps preserve solutions, this implies the equivalence of Pr and Pr'. We distinguish cases according to the rule applied:

- $Pr,\ a \approx_\alpha \xi\Diamond t \Longrightarrow Pr'$. Then $[\![a \approx_\alpha \xi\Diamond t]\!] = a \approx_\alpha \xi_\pi{\cdot}t,\ \xi_A \# t$ so if t is not an atom or if t is an atom but $a \neq \xi_\pi{\cdot}t$ or $t \in \xi_A$ then there is no solution.

  If t is an atom b, $\xi_\pi{\cdot}t = a$ and $t \notin \xi_A$, then $Pr' = Pr$, and the reduction step can be simulated as follows:
  $$\begin{array}{ll} & [\![Pr]\!], a \approx_\alpha \xi_\pi{\cdot}t, \xi_A \# t \\ \equiv & [\![Pr]\!], a \approx_\alpha a, (c \# b)_{c\in\xi_A} \\ \Longrightarrow & [\![Pr]\!], (c \# b)_{c\in\xi_A} \\ \stackrel{*}{\Longrightarrow} & [\![Pr]\!] \end{array}$$

- $Pr, f\ s \approx_\alpha \xi\Diamond t \Longrightarrow Pr'$. If t is not a term of the form $f\ u$ then there is no solution. If $t \equiv f\ u$ the step can be simulated by standard reduction steps as follows:
  $$\begin{array}{ll} & [\![Pr]\!],\ f\ s \approx_\alpha \xi_\pi{\cdot}f\ u,\ \xi_A \# f\ u \\ \stackrel{*}{\Longrightarrow} & [\![Pr]\!],\ s \approx_\alpha \xi_\pi{\cdot}u,\ \xi_A \# u \\ = & [\![Pr,\ s \approx_\alpha \xi\Diamond u]\!] \\ = & [\![Pr']\!] \end{array}$$

- $Pr, (s_1, \ldots, s_n) \approx_\alpha \xi\Diamond t \Longrightarrow Pr'$. If t is not a tuple, there is no solution. If $t = (u_1, \ldots, u_n)$ the step can be simulated by standard steps as follows:
  $$\begin{array}{ll} & [\![Pr]\!],\ (s_1, \ldots, s_n) \approx_\alpha \xi_\pi{\cdot}(u_1, \ldots, u_n),\ \xi_A \# (u_1, \ldots, u_n) \\ \stackrel{*}{\Longrightarrow} & [\![Pr]\!],\ (s_i \approx_\alpha \xi_\pi{\cdot}u_i)_1^n,\ (\xi_A \# u_i)_1^n = [\![Pr,\ (s_i \approx_\alpha \xi\Diamond u_i)_1^n]\!] = [\![Pr']\!] \end{array}$$

- $Pr,\ [a]s \approx_\alpha \xi\Diamond t \Longrightarrow Pr'$. If t is not an abstraction, there is no solution. If $t \equiv [b]u$, where a and b denote two atoms not necessarily different, then we distinguish two cases:

  If $\xi_\pi{\cdot}b = a$, then
  $$\begin{array}{ll} & [\![Pr]\!],\ \ [a]s \approx_\alpha \xi_\pi{\cdot}\ [b]u\ ,\ \xi_A \#\ \ [b]u \\ \stackrel{*}{\Longrightarrow} & [\![Pr]\!],\ s \approx_\alpha \xi_\pi{\cdot}u,\ \xi_A \setminus \{b\} \# u \\ = & [\![Pr]\!],\ s \approx_\alpha ((a\ \xi_\pi{\cdot}b) \circ \xi_\pi){\cdot}u,\ (\xi_A \cup \{\xi_\pi^{-1}(a)\}) \setminus \{b\} \# u \\ = & [\![Pr,\ s \approx_\alpha \xi'\Diamond u]\!] \\ = & [\![Pr']\!] \end{array}$$

  where $\xi' = ((a\ \xi_\pi{\cdot}b) \circ \xi_\pi),\ (\xi_A \cup \{\xi_\pi^{-1}{\cdot}a)\} \setminus \{b\})$.

If $\xi_\pi \cdot b \neq a$, then, since freshness is preserved by permutation (i.e., it is an equivariant relation, see [20, 51]), we obtain:

$$\begin{array}{ll} & [\![Pr]\!],\ \ [a]s \approx_\alpha \xi_\pi\cdot\ [b]u\ ,\ \xi_A \#\ \ [b]u \\ \stackrel{*}{\Longrightarrow} & [\![Pr]\!],\ s \approx_\alpha ((a\ \xi_\pi\cdot b)\circ\xi_\pi)\cdot u,\ \xi_A \setminus \{b\} \# u\ ,\ a \#\ \ (\xi_\pi\cdot u) \\ = & [\![Pr]\!],\ s \approx_\alpha ((a\ \xi_\pi\cdot b)\circ\xi_\pi)\cdot u,\ (\xi_A \cup \{\xi_\pi^{-1}(a)\}) \setminus \{b\} \# u \\ = & [\![Pr,\ s \approx_\alpha \xi' \Diamond u]\!] \\ = & [\![Pr']\!] \end{array}$$

where $\xi' = ((a\ \xi_\pi\cdot b)\circ\xi_\pi),\ (\xi_A \cup \{\xi_\pi^{-1}\cdot a)\} \setminus \{b\})$.

$*$

Phase 2 This phase takes as input an environment problem in ph1nf, and moves the permutations to the right-hand side. More precisely, given a problem in ph1nf, we apply the rule:

$$\pi\cdot X \approx_\alpha \xi\Diamond t \Longrightarrow X \approx_\alpha (\pi^{-1}\cdot\xi)\Diamond t \qquad (\pi \neq \mathsf{Id})$$

where $\pi^{-1}\cdot\xi = (\pi^{-1}\circ\xi_\pi,\ \xi_A)$. Note that π^{-1} applies only to ξ_π here, because $\pi\cdot X \approx_\alpha \xi\Diamond t$ represents $\pi\cdot X \approx_\alpha \xi_\pi\cdot t,\ \xi_A \# t$.

If the problem is irreducible (i.e., it is a normal form for the rule above), we say it is a **phase 2 normal form**, or **ph2nf** for short.

Proposition 52 (Phase 2 normal forms) *Given a ph1nf problem, it has a unique normal form for the rule above, and it is either $\perp$ or a problem of the form $(X_i \approx_\alpha \xi_i\Diamond t_i)_1^n$, where the terms t_i are standard nominal terms.*

Proof The rule is clearly terminating (each application consumes a permutation on the left) and it does not overlap with itself, therefore it is confluent [38]. This implies the unity of normal forms. Given a ph1nf problem, its normal form cannot contain a suspended variable on the left-hand side. $*$

Proposition 53 (Correctness) *$\pi\cdot X \approx_\alpha \xi\Diamond t$ is equivalent to $X \approx_\alpha (\pi^{-1}\cdot\xi)\Diamond t$.*

Proof We need to show that the standard forms of both problems are equivalent. Since $[\![\pi\cdot X \approx_\alpha \xi\Diamond t]\!] = \pi\cdot X \approx_\alpha \xi_\pi\cdot t\ ,\ \xi_A \#\ t$, and $[\![X \approx_\alpha (\pi^{-1}\cdot\xi)\Diamond t]\!] = X \approx_\alpha (\pi^{-1}\circ\xi_\pi)\cdot t\ ,\ \xi_A \#\ t$, the result follows directly from the preservation of $\approx_\alpha$ by permutations (see [20, 51]). $*$

Phase 3 In the phases 1 and 2 we deal with $\approx_\alpha$ constraints. Phase 3 takes a ph2nf and simplifies freshness constraints, by propagating environments over terms. Since the input is a problem in ph2nf, each constraint has the form $X \approx_\alpha \xi\Diamond t$. We reduce it with the following rewrite rules, which propagate ξ over t and deal with problems containing $\perp$ (denoted $Pr[\perp]$):

$$\begin{array}{rcl} \xi\Diamond a & \Longrightarrow & \begin{cases} \xi_\pi \cdot a & a \notin \xi_A \\ \perp & a \in \xi_A \end{cases} \\ \xi\Diamond f\ t & \Longrightarrow & f\ (\xi\Diamond t) \\ \xi\Diamond(t_1, \ldots, t_j) & \Longrightarrow & (\xi\Diamond t_i)_1^j \\ \xi\Diamond\ [a]s & \Longrightarrow & [\xi_\pi \cdot a]((\xi \setminus \{a\})\Diamond s) \\ \xi\Diamond(\pi \cdot X) & \Longrightarrow & (\xi \circ \pi)\Diamond X \\ Pr[\perp] & \Longrightarrow & \perp \end{array}$$

where $\xi \setminus \{a\} = (\xi_\pi, \xi_A \setminus \{a\})$ and $\xi \circ \pi = ((\xi_\pi \circ \pi), \pi^{-1}(\xi_A))$.

These rules move environments inside terms, so formally we need to extend the definition of nominal term, to allow us to attach an environment at any position inside the term. We omit the definition of terms with suspended environments, and give just the grammar for the normal forms, which may have environments suspended only on variable leaves:

Definition 75 *The language of normal environment terms is defined by:*

$$T_\xi = a \mid f\ T_\xi \mid (T_\xi, \ldots, T_\xi) \mid\ [a]T_\xi \mid \xi\Diamond X$$

If the problem is irreducible (i.e., it is a normal form for the rules above), we say it is a **phase 3 normal form**, or **ph3nf** for short.

Proposition 54 (Phase 3 normal forms - ph3nf) *The normal forms for this phase are either $\perp$ or $(X_i \approx_\alpha t_i)_1^n$ where $t_i \in T_\xi$.*

Proof By inspection of the rules, it is easy to see that environments move down, and suspend on variables. $*$

To give a semantics to the problems generated in phase 3, we extend the definition of a standard form (see Definition 70) as follows:

Definition 76 (Standard Form) *The standard form of a problem with suspended environments is obtained by normalising with the rewriting rule*

$$s \approx_\alpha C[\xi \Diamond t] \Longrightarrow s \approx_\alpha C[\xi_\pi \cdot t],\ \xi_A \# t$$

where s and t are nominal terms.

Remark 7 *The rule in Definition 71 is a particular case of this one, taking an empty context, that is, $C[\ ] = [\ ]$.*

This rule is terminating and confluent, therefore normal forms are unique; we will use the notation $[\![Pr]\!]$ as before.

Proposition 55 (Correctness) *Let Pr be a problem in ph2nf, and $Pr \Longrightarrow Pr'$ using a phase 3 rule. Then Pr and Pr' are equivalent.*

Proof As before, it is sufficient to show that if $\perp$ is obtained then the problem has no solutions, and otherwise the standard forms of Pr and Pr' are equivalent. We show the three interesting cases (recall that standard reductions preserve solutions).

- Assume we have $s \approx_\alpha C[\xi \Diamond a]$. Then its standard form is $[\![s \approx_\alpha C[\xi_\pi \cdot a]]\!],\ \xi_A \# a$. Notice that $\perp$ can only be obtained if $a \in \xi_A$, and in this case the standard form has no solutions.

  If $a \notin \xi_A$ then $[\![s \approx_\alpha C[\xi_\pi \cdot a]]\!],\ \xi_A \# a$ is equivalent to $[\![s \approx_\alpha C[\xi_\pi \cdot a]]\!]$.

- Assume we have $s \approx_\alpha C[\xi \Diamond\ [a]t]$. Then its standard form is

  $$[\![s \approx_\alpha C[\ [\xi_\pi \cdot a]\xi_\pi \cdot t]]\!],\ \xi_A \#\ \ [a]t$$

  which is equivalent to $[\![s \approx_\alpha C[\ [\xi_\pi \cdot a]\xi_\pi \cdot t]]\!],\ \xi_A \setminus \{a\} \# t$, which is the standard form of $s \approx_\alpha C[\ [\xi_\pi \cdot a](\xi \setminus \{a\})\Diamond t]$.

- Assume we have $s \approx_\alpha C[\xi \Diamond (\pi \cdot X)]$. Then its standard form is

  $$[\![s \approx_\alpha C[(\xi_\pi \circ \pi) \cdot X]]\!],\ \xi_A \# \pi \cdot X$$

  and using standard reduction rules we obtain $[\![s \approx_\alpha C[(\xi_\pi \circ \pi) \cdot X]]\!],\ \pi^{-1}(\xi_A) \# X$, equivalent to $[\![s \approx_\alpha C[(\xi \circ \pi)\Diamond X]]\!]$.

$*$

Phase 4 This phase computes the standard form of a ph3nf, using a particular case of the rule given in Definition 76:

$$X \approx_\alpha C[\xi \Diamond X'] \Longrightarrow X \approx_\alpha C[\xi_\pi \cdot X'] \ , \ \xi_A \ \# \ X'$$

Definition 77 *We define the sets of variables on the left-hand (resp. right-hand) side of an $\approx_\alpha$-constraint as $V_l(s \approx_\alpha t) = V(()s)$ (resp. $V_r(s \approx_\alpha t) = V(t)$). We extend the notation to problems as follows: $V_{l/r}(A \ \# \ t) = \emptyset$ and $V_{l/r}(Pr_1, \ Pr_2) = V_{l/r}(Pr_1) \cup V_{l/r}(Pr_2)$.*

Proposition 56 (Phase 4 normal forms - ph4nf) *If we normalise a ph3nf using the rule above, we obtain either $\perp$ or $(X_i \approx_\alpha u_i)_{i \in I}, (A_j \ \# \ X_j)_{j \in J}$ where u_i are nominal terms and I, J may be empty.*

Moreover, $(X_i)_{i \in I} \subseteq V_l(Pr)$ and $(X_j)_{j \in J} \subseteq V_r(Pr)$. Thus, if the right-hand sides of $\approx_\alpha$-constraints in Pr are ground, there are no freshness constraints in the ph4nf (because $V_r(Pr) = \emptyset$).

Proof By Proposition 54, environments can only be suspended on variables in a ph3nf. So if the problem is irreducible by the rule above, it cannot contain any environment.

The variable property follows from the fact that the rules never move subterms of the left-hand side of an $\approx_\alpha$-constraint into the right-hand side, and all the freshness constraints generated involve subterms of the right-hand side. $\ast$

Clearly, this phase preserves the set of solutions, it terminates and it does not raise $\perp$, therefore if the resulting problem is $\perp$ so was the ph3nf.

Since all the reduction rules, except the rule dealing with $\perp$, are local (i.e. only modify one constraint), the result of applying the core algorithm to a set of constraints is the union of the results obtained for each individual constraint (assuming $\perp, Pr \equiv \perp$):

$$\overline{(s_i \approx_\alpha \xi_i \Diamond t_i)_1^n}^{\,c} = (\overline{s_i \approx_\alpha \xi \Diamond t_i}^{\,c})_1^n$$

Thus, without loss of generality we can consider the input of the core algorithm to be one single constraint $s \approx_\alpha \xi \Diamond t$.

9.3 Checking the validity of α-equivalence constraints

To check that a set Pr of α-equivalence constraints is valid, we first run the core algorithm on Pr and then reduce the result $\overline{Pr}^{\,c}$ by the following rule:

$$(\alpha) \quad Pr\ ,\ X \approx_\alpha t \Longrightarrow \begin{cases} Pr\ ,\ supp(\pi) \# X & \text{if } t = \pi \cdot X \\ \bot & \text{otherwise} \end{cases}$$

where $supp(\pi)$ is the **support** of π: $supp(\pi) = \{a \mid \pi \cdot a \neq a\}$.

Since this rule is terminating (each application consumes one $\approx_\alpha$-constraint) and locally confluent, it is confluent [38], therefore normal forms exist and are unique.

Definition 78 *We will denote by $\overline{Pr}^{\,\approx_\alpha}$ the normal form of $\overline{Pr}^{\,c}$ by the rule above.*

Proposition 57 (Normal forms $\overline{Pr}^{\,\approx_\alpha}$) *$\overline{Pr}^{\,\approx_\alpha}$ is either $\bot$ or $(A_i \# X_i)_1^n$.*

Proof $\overline{Pr}^{\,c}$ is a ph4nf, so by Proposition 56 it is either $\bot$ or $(X_i \approx_\alpha t_i)_1^n, (A_j \# X_j)_{j \in J}$ where t_i are standard terms. While there are constraints $X_i \approx_\alpha t_i$, the problem is reducible by (α), and each $\approx_\alpha$-constraint is replaced by a set of freshness constraints. ∗

Proposition 58 (Correctness) *If $\overline{Pr}^{\,\approx_\alpha}$ is $\bot$ then Pr is not valid. If $\overline{Pr}^{\,\approx_\alpha}$ is $(A_i \# X_i)_1^n$ then $\overline{Pr}^{\,\approx_\alpha} \vdash Pr$.*

Proof The core algorithm preserves solutions, as a consequence of Propositions 51, 53, 55. Moreover, $X \approx_\alpha t$ is valid (see Section 4) if and only if t is $\pi \cdot X$ and $supp(\pi) \# X$, because $ds(\pi, \mathsf{Id}) = supp(\pi)$. Hence (α) is correct. ∗

If the left-hand sides of $\approx_\alpha$-constraints in Pr are ground, then $\overline{Pr}^{\,c} = \overline{Pr}^{\,\approx_\alpha}$; rule (α) is not necessary in this case (by Prop 56). Formally:

Proposition 59 *Let $Pr = (s_i \approx_\alpha \xi_i \lozenge t_i)_1^n$ where $V(s_i) = \emptyset$ for $1 \leq i \leq n$. Then $\overline{Pr}^{\,\approx_\alpha} = \overline{Pr}^{\,c}$.*

Remark 8 *The rule* (α)*, when it does not involve* $\bot$*, is also local (i.e. it only affects one constraint in the problem), so in practice we will implement two functions, one taking only* $s \approx_\alpha \xi \Diamond t$ *as input and computing* $\overline{s \approx_\alpha \xi \Diamond t}^{\approx\alpha}$*, and the other taking an environment problem as input and applying the first function on each constraint:* $\overline{(s_i \approx_\alpha \xi_i \Diamond t_i)_i^n}^{\approx\alpha} = (\overline{s_i \approx_\alpha \xi_i \Diamond t_i}^{\approx\alpha})_i^n$

In the case of ground terms, our algorithm to check α-equivalence relies only on the information stored in ξ_π. Thus, it could be seen as an implementation of the inductive definition of ground α-equality used in [25] (and attributed to Shankar [47]), which also relies on a list of pairs of names.

9.4 Solving Matching Problems

To solve a matching problem Pr, we first run the core algorithm on Pr and then if the problem is non-linear we normalise the result $\overline{Pr}^{c}$ by the following rule.

$$({}_?\approx) \quad Pr,\ X \approx_\alpha s\ ,\ X \approx_\alpha t \Longrightarrow \begin{cases} Pr\ ,\ X \approx_\alpha s\ ,\ \overline{s \approx_\alpha t}^{\approx\alpha} & \text{if } \overline{s \approx_\alpha t}^{\approx\alpha} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

This rule is terminating: each reduction consumes at least one $\approx_\alpha$-constraint since $\overline{Pr}^{\approx\alpha}$ does not introduce $\approx_\alpha$-constraints (by Proposition 57) and is also terminating.

Definition 79 *We denote by* $\overline{Pr}^{?\approx}$ *a normal form of* $\overline{Pr}^{c}$ *by the rule* $({}_?\approx)$*.*

Proposition 60 (Normal forms $\overline{Pr}^{?\approx}$**)** *If we normalise* $\overline{Pr}^{c}$ *using the rule above, we obtain either* $\bot$ *or* $(X_i \approx_\alpha t_i)_1^n$*,* $(A_i \# X_i)_1^m$ *where* t_i *are standard terms, all* X_i *in the equations* $(X_i \approx_\alpha t_i)_1^n$ *are different variables and* $\forall i, j : X_i \notin V(t_j)$*.*

Proof By Proposition 57, rule $({}_?\approx)$ does not produce $\approx_\alpha$-constraints. The property then follows from Proposition 56, since $\mathrm{V}(()t_i) \subseteq V_r(Pr)$, $X_i \in V_l(Pr)$, and $V_l(Pr) \cap V_r(Pr) = \emptyset$ in a matching problem. $*$

A problem of the form $(X_i \approx_\alpha t_i)_1^n$ where all X_i are distinct variables and $X_i \notin \mathrm{V}(t_j)$ is the coding of an idempotent substitution σ defined by

$$\sigma(X) = \begin{cases} t_i & \text{if } \exists i \quad X = X_i \\ X & \text{otherwise.} \end{cases}$$

$(A_i \# X_i)_1^n$ is a freshness context Δ. So the result of the algorithm is either $\bot$ or a pair (σ, Δ) of a substitution and a freshness context.

Proposition 61 (Correctness) *$\overline{Pr}^{\,?\approx}$ is a most general solution of the matching problem Pr.*

Proof Rule $({}_?\approx)$ is justified by the equivalence:

$$X \approx_\alpha s,\ X \approx_\alpha t \Leftrightarrow X \approx_\alpha s,\ s \approx_\alpha t$$

If $\bot$ is raised, either:

- $\overline{Pr}^{\,c} = \bot$ and the problem has no solution by correctness of the core algorithm.
- Rule $({}_?\approx)$ raised $\bot$, so $\overline{s \approx_\alpha t}^{\,\approx\alpha} = \bot$, i.e., s and t are not α-equivalent, and the problem has no solution.

$*$

Note that if variables occur linearly in patterns (i.e., we have a linear matching problem), then the result of the core algorithm is already the solution of the problem.

Formally:

Proposition 62 *Let $Pr = (s \approx_\alpha \xi \Diamond t)$ where for all $X \in V(s)$, X occurs only once in s. Then $\overline{Pr}^{\,?\approx} = \overline{Pr}^{\,c}$.*

Proof If all variables X in s are linear, then $\overline{Pr}^{\,c}$ contains only one equation for each X and rule $({}_?\approx)$ cannot be applied. $*$

Chapter 10

Implementation

In this chapter we describe the implementation of the core algorithm, then the implementation of the final phases specific to α-equivalence and matching respectively. In the sequel, Pr_0 will denote the input problem and A_0 the set of atoms occurring in Pr_0.

The core algorithm is implemented using a monadic tower made of first-class monadic signatures as presented in chapter 3. The lower layer is a *store* management layer. A **store** is a structure associating a value to some variables. Substitution, freshness context and arr_{uf} are stores. Because we want the description of the algorithm and the implementation to be as simple as possible, we consider in the rest of the chapter that stores are arrays but it may sometimes be easier to take another implementation of stores: for example as persistent data. In the implementation, the actual representation of stores is abstracted and all algorithms use the first-class monadic signature StoreCall defined in A.4 to manage stores.

The next layer is the exception handling monad transformer

$$\text{SCont }(\text{DMonadError }\mathbf{error})$$

presented in chapter 3.5.2.

For the sake of simplicity the lifting operations required to call an operation of a layer from an upper layer will not be shown. For a detailed implementation of these algorithm, please refer to appendices A.4, A.6, A.9 and A.10.

The complexity of the `core()` algorithm depends heavily on the implementation of sets and permutations. Unfortunately the appropriate implementation choice depends on the input. So we add the two layers SetCall and PermCall defined in section 4.4. For now the tower has four layers, the lowest is StoreCall for store management, the next one is DMonadError for exceptions, then SetCall set for native sets and the top one is PermCall perm for native permutations. The Monad is

```
1  SContT  (PermCall     perm  )
2 (SContT  (SetCall      set   )
3 (SContT  (DMonadError  error )
4 (SCont   (StoreCall    store Var))))
```

10.1 The Freshness Layer

In the process of the `core()` algorithm, freshness constraints $A \# X$ are generated. For solving α-equivalence and matching problem they have to be added to the solution if any but for rewriting they will have to be checked against the freshness context of the rule. Once again we define a new layer to the tower. This layer is implemented in the *Haskell* module A.10 as the first class monadic signature data type FreshContextCall set where set is the native type of sets. It provides the following calls:

- FreshConstraint deals with a freshness constraint $A \# X$
- FreshContextRule gives, for a variable X, the set of atoms

  $$A = \{a \mid a \# X \in \Delta\}$$

  where Δ is the freshness context of the rewriting rule. When solving an α-equivalence or matching problem, Δ is considered empty.
- FreshToList returns the list corresponding to the freshness context

Let arr_Δ an array of sets of atoms indexed by variables. arr_Δ represents the freshness context Δ if and only if

$$arr_\Delta[X] = \{a \mid a \# X \in \Delta\}$$

When solving an α-equivalence or matching problem, we start with an empty context and fill it with constraints generated by the algorithm. The interpretation of FreshContextCall for this chapter is described in the algorithm 11.

```
1  interpretFreshContextSolve(c) = ;
2    switch c do
3      case FreshConstraint A # X k
4        arr_Δ[X] = arr_Δ[X] ∪ A ;
5        k(())
6      case FreshContextRule X k
7        k(∅)
8      case FreshToList k
9        let l be the list corresponding to arr_Δ;
10       k(l)
11
12   endsw
```

Algorithm 11: Interpretation of Freshness when Solving an α-equivalence or matching problem

We now have five layers in the tower:

```
1  SContT  (FreshContextCall  set  )
2 (SContT  (PermCall          perm )
3 (SContT  (SetCall           set  )
4 (SContT  (DMonadError       error)
5 (SCont   (StoreCall         store Var)))))
```

10.2 Only one environment: The environment layer

As it is presented above, the core algorithm uses many environments, created by either copying (for example in the tuple case of phases 1 and 3) or modifying (for example in abstraction case of phase 1 and 3) an existing one. Here we will use only on environment. Modifications will be done in place and copies will point to the same environment.

In practice this single global environment ξ is abstracted into the sixth monadic layer implemented in the *Haskell* module A.9. The first-class monadic signature data type is EnvCall set perm where set (resp. perm) is the type of native set and (resp. permutations) . The layer provides the following operations:

- envImage of type $A \to A \to m\ A$. envImage a returns the image of a by the environment
- envImageInv of type $A \to A \to m\ A$. envImage a returns the inverse image of a by the environment
- envIsFresh of type $A \to m$ Bool. envIsFresh a returns whether or not there is a constraint $a\ \#$ in the environment
- envSetFresh of type $A \to$ Bool $\to m\ ()$. envSetFresh a b add (resp. remove) the atom a from the freshness constraint set if b is true (resp. false).
- envComposeLeft of type $\Pi - > m\ ()$. envComposeLeft **pi** perform $\pi{\cdot}\xi$ in place
- envComposeRight of type $\Pi - > m\ ()$. envComposeLeft **pi** perform $\xi\Diamond\pi$ in place
- envLocal of type $FORALLaDOTm\ a \to m\ a$. envLocal c performs the computation c locally which means that even if c modifies the environment, it will be the same as before after the call.
- envPerm of type $m\ \Pi$ returns the permutation of ξ
- envSet of type $m\ S$ returns the freshness set of ξ

A environment layer is not a simple state monad because envLocal would require to save the whole environment before computing c, which would be done in linear time in the size of the environment but we need constant time. Instead the monad maintains a stack of the modifications done and undoes the operations on the stack when exiting envLocal. Storing these modification is cheap: storing envComposeLeft **pi** (resp. envComposeRight) will only require to store a data (ComposeLeft, **pi**) what is linear in the size of **pi**. To undo it we only need to call envComposeLeft (resp. envComposeRight) with π^{-1}. Storing envSetFresh a b is also easy, we have only to remember whether or not a is in the freshness set and, when exiting envLocal, adding (resp. removing) it if it was (resp. was not) in it.

Phase2 and Phase4 Phase 2 is:

$$\pi{\cdot}X \approx_\alpha \xi\Diamond t \Longrightarrow X \approx_\alpha (\pi^{-1}{\cdot}\xi)\Diamond t \qquad (\pi \neq \mathsf{Id})$$

and Phase 4:

$$X \approx_\alpha C[\xi\Diamond X'] \Longrightarrow X \approx_\alpha C[\xi_\pi{\cdot}X'] \ , \ \xi_A \# X'$$

.

The two functions are described in algorithm 12. Actually Phase 2 has been a bit modified to be able to perform nominal rewriting. Because `freshContextRule()` returns always $\emptyset$ when solving a matching problem, these modification affect neither the correctness nor the complexity of Phase 2.

Input: A suspended variable $\pi{\cdot}X$ and a term t

```
1  ;
2  phase2(π·X, t) = ;
3    envComposeLeft(π^{-1}) ;
4    A = freshContextRule(X);
5    π' = envPerm;
6    for a ∈ (π'^{-1}(A) do
7    |    envSetFresh(a,⊤)
8    end
9    ;
10   return (X, t);
```

Input: A variable X

```
11 ;
12 phase4(X) = ;
13   A = envSet();
14   π = envPerm();
15   freshConstaint(A, # X);
16   return π·X;
```

Algorithm 12: Phase 2 and 4

10.3 Combining the phases: Streams

Modifying the environment in place requires running the phases in a specific order. For example, let us consider the reduction of a $\xi\Diamond(nomAbsat, u)$ by the phase3, it is reduced into ($[\xi_\pi{\cdot}a](\xi \setminus \{a\})\Diamond s, \xi\Diamond u$. We can keep virtually two different environment thanks to envLocal, but switching from one environment to another would require to undo all the modification made on the first, to apply those of the second. It would require too many operations. It is more efficient to reduce completely one branch, for example $(\xi \setminus \{a\})\Diamond s$ and only then restore the environment to reduce completely $\xi\Diamond u$. Such a reduction is called a **local reduction strategy**.

The problem is running phases one after another is not a local reduction strategy. The idea is to interleave the phases. It could be done simply by calling phase 2 in phase 1, phase 3 in phase 2 and phase 4 on phase 3 but this solution is neither flexible nor aesthetic. Instead we transform phase 1 and 3 into a stream. Phase 2 and 4 do not need be transformed as they do not fork the computation. Phase 1 and 3 become (To simplify the code, lifting operations have been hidden) as described in algorithms 13 and 14.

streamStep is the only call of the seventh layer of the tower. The signature data type of the layer is

```
1 data StreamCall arg ret n r = StreamStep arg (ret -> r)
```

streamStep give a value to the interpretation function and returns what the interpretation send back. This can be seen as stream for which it is possible to act on the flow. It is implemented in the *Haskell* module A.10.

Combining the phases is done by instantiating the stream layer for each function. `core()` is then as described in algorithm 15.

`core()` is also a stream because of `streamStep(`(X, s')`)` in `interpretPhase1()`. The interpretation of this stream will give either the α-equivalence or the matching algorithm.

```
    Input: Two nominal terms t and u
 1  ;
 2  phase1(t,u) = switch (t,u) do
 3  |  case ([a]t, [b]u)
 4  |  |  a' = envImageInv(a);
 5  |  |  b' = envImage(b);
 6  |  |  envLocal(;
 7  |  |    envSetFresh(a', ⊤) ;
 8  |  |    envSetFresh(b, ⊥) ;
 9  |  |    envComposeLeft((a b')) ;
10  |  |    phase1(t , u);
11  |  |    );
12  |  case ((t_1, ..., t_n), (u_1, ..., u_n))
13  |  |  phase1(t_1, u_1) ;
14  |  |  ... phase1(t_n, u_n)
15  |  case ((f t') , (f u'))  where f is a function symbol
16  |  |  phase1(t', u')
17  |  case (a, b) where a and b are two atoms
18  |  |  if envIsFresh(b) then
19  |  |  |  dthrowError(freshness constraints not satisfied)
20  |  |  end
21  |  |  if a = envImage(b) then
22  |  |  |  return ()
23  |  |  else
24  |  |  |  dthrowError(atoms not equal)
25  |  |  end
26  |  case (π·X, u)
27  |  |  streamStep((π·X, u))
28  |  otherwise
29  |  |  dthrowError(t and u do not match)
30  |  endsw
31  endsw
```

Algorithm 13: Streamed version of Phase 1

```
Input: A nominal term t
1  ;
2  phase3(t) = switch t do
3  |  case a where  a  is an atom
4  |  |  if envIsFresh(a) then
5  |  |  |  dthrowError(freshness constraints not satisfied)
6  |  |  else
7  |  |  |  envImage(a)
8  |  |  end
9  |  case f u → phase3(u);
10 |  case (u_1, ..., u_n)
11 |  |  s_1 = phase3(u_1);
12 |  |  ...;
13 |  |  s_n = phase3(u_n);
14 |  |  return (s_1, ..., s_n)
15 |  case [a]u
16 |  |  s = envLocal(;
17 |  |        envSetFresh(a, ⊥);
18 |  |        phase3(u);
19 |  |        );
20 |  |  b = envImage(a);
21 |  |  return [b]s
22 |  case π·X
23 |  |  envlocal(;
24 |  |    envComposeRight(π);
25 |  |    streamStep(X);
26 |  |    );
27 |
28 endsw
```

Algorithm 14: Streamed version of Phase 3

Input: Two nominal terms t and u

```
1  ;
2  core(t,u) = instantiate(interpretPhase1() , phase1(t,u)) ;
3    where;
4        interpretPhase3(StreamStep X k) = k(phase4(X));
5        ;
6        phase3and4(t) = instantiate(interpretPhase3() , phase3(t));
7        ;
8        interpretPhase1(StreamStep (π·X, u') k) = ;
9            (X, s) = phase2(π·X, u');
10           s' = phase3and4(s) ;
11           streamStep((X, s'));
12           k(());
```

Algorithm 15: The core algorithm

10.4 α-equivalence and Matching

Interpretation as α-equivalence Solving the α-equivalence problem $s \approx_\alpha t$ is interpreting the stream of (X, s) that `core(s,t)` returns with the rule (α) 9.3. The rule (α) is implemented as the `interpretAlpha()` interpretation function in algorithm 16.

Interpretation as Matching Solving the matching problem $s \approx_\alpha t$ is interpreting the stream of (X, s) that `core(s,t)` returns with the rule $({}_?\approx)$ 9.4. The rule $({}_?\approx)$ is implemented as the `InterpretMatch()` interpretation function in algorithm 17. This function uses a global array representing the current substitution. Such an array is defined by:

Definition 80 *arr_σ is an array representing the substitution σ if and only if*

$$arr_\sigma[X] = \begin{cases} \textsf{Nothing} & \textit{if } \sigma(X) = X \\ \textsf{Just t} & \textit{if } \sigma(X) = t \textit{ and } t \neq X \end{cases}$$

```
Input: two nominal terms t and u
1 ;
2 alpha(t,u) = ;
3   instantiate(interpretAlpha(), core(t,u)) freshContextToList()
    where;
4     interpretAlpha((X, s)) = Switch(X, s) case (X, π·X)
5       freshConstaint(supp(π) # X)
6     otherwise
7       dthrowError(Substitutions not allowed)
8     endsw
```

Algorithm 16: α-equivalence Interpretation of *core*

```
Input: two nominal terms t and u
1 ;
2 match(t,u) = ;
3     fill arr_σ with Nothing;
4     instantiate(InterpretMatch, core(t,u));
5     return arr_σ;
6     where InterpretMatch((X, s), k) = ;
7         if arr_σ[X] = Nothing then
8             arr_σ[X] = Just s
9         else
10            (X, t') = arr_σ[X] ;
11            if | s |<| t' | then
12                arr_σ[X] = Just s
13            end
14            ;
15            alpha(s,t')
16        end
17        k(())
```

Algorithm 17: Matching Interpretation of *core*

Choosing the implementation of Permutations and Sets We will use either interpret permutations and sets of atoms as either as mutable arrays or as persistent data depending on the kind of problem to be solved:

Case	Alpha-equivalence	Matching
Ground	mutable arrays	mutable arrays
Linear	persistent data	persistent data
Non-linear	persistent data	mutable arrays

Note that when the problem is ground, the streams of all the phases are empty, so no interpretation of StreamCall is required. Since in this case we only need to access and update the environment, arrays are more efficient. With linear, non-ground problems, we need to interpret StreamCall, and the cost is quadratic using arrays, but log-linear using functional maps. We will discuss the non-linear case in Section 10.5.

10.5 Complexity

In this section we analyse the complexity of the core algorithm and the final α-equivalence and matching phases.

Let $Pr_0 = s \approx_\alpha \xi \lozenge t$ be the input problem; s and t are coded as trees and ξ is coded as a pair of a permutation and a set of atoms (permutations being a tuple of the actual permutation, its inverse, and its support), as discussed in Section 10. Atoms are coded as integers, as explained above. Let M_{A_0} be the maximum atom in A_0 (the set of atoms occurring in Pr_0). Let $|t|_n$ be the number of nodes in the tree representing t. Let P be the multiset of permutations in s, t, and $|\pi|$ be the size of the array representing $\pi \in P$ (or the size of the map if we are dealing with non-ground problems). Finally, let $\mathrm{MV}(t)$ be the multiset of the occurrences of variables in t.

Core algorithm Below we analyse the cost of the traversal of the data structure, and the cost of the operations involved in the rules.

Proposition 63 *The core algorithm is linear in the size of the problem Pr_0 in the ground case, using mutable arrays. In the non-ground case it is log-linear using functional maps and $\vartheta(|s \approx_\alpha t| + |M_{A_0}| \times |t|_n)$ using mutable arrays.*

The idea of the proof is that the core algorithm is essentially a traversal of the data structure representing the problem. Phases 1 to 3 are trivially linear with arrays and log-linear with functional maps. Phase 4, with functional maps is done in $\vartheta(|\mathrm{M}_{A_0}|)$ time and $\vartheta(|\mathrm{M}_{A_0}| \times |t|_n)$ with arrays.

Proof There are three kinds of rules in the core algorithm:

- propagation rules, such as the rules in phase 1 and phase 3, except the rule $\xi\Diamond(\pi{\cdot}X) \Longrightarrow (\xi \circ \pi)\Diamond X$;
- rules dealing with input permutations (i.e., permutations that occur in the initial problem Pr_0), such as phase 2 rules and the rule $\xi\Diamond(\pi{\cdot}X) \Longrightarrow (\xi \circ \pi)\Diamond X$;
- rules dealing with $\xi\Diamond X$ to compute standard form.

The propagation rules visit each node in the tree representation of t at most once. They may apply at most one swapping (resp. adding/removing an atom) to modify the environment and the same again (resp. removing/adding this atom) to restore it. All these operations are done a constant number of times and in constant (resp. logarithmic) time with arrays (resp. maps). Hence, they take $\vartheta(|t|_n)$ time with arrays and $\vartheta(log(|t|_n) \times |t|_n)$ with maps.

Rules dealing with permutations, i.e., phase 2 rules and the rule $\xi\Diamond(\pi{\cdot}X) \Longrightarrow (\xi \circ \pi)\Diamond X$, compose an input permutation π and the environment. This takes $\vartheta(|\pi|)$ (resp. $\vartheta(log(|\pi|) \times |\pi|)$) time per input permutation π with arrays (resp. maps), as described in Section 4.4. They are applied at most once per input permutation, so take $\sum_{\pi\in P} |\pi|$ in total with arrays and $\sum_{\pi\in P} log(|\pi|) \times |\pi|$ with maps.

If the right-hand side terms are ground, only these two kind of rules can be applied, so the algorithm is linear in the size of the problem. Otherwise, there's also the rule dealing with $\xi\Diamond X$ in phase 4 to compute standard form (i.e., the output of the algorithm). It is applied at most once per occurrence of a variable in t, and is done in constant time with functional maps and $\vartheta(|\mathrm{M}_{A_0}|)$ time and space with arrays. $*$

If the problem consists of more than one matching or α-equivalence constraint, they can be solved independently. Thus, we run the core algorithm on each of

them and concatenate the results, obtaining again an algorithm of the same complexity.

Remark 9 *Our algorithm checks $\approx_\alpha$-constraints and the freshness constraints that are generated, all in one traversal of the data structure representing the problem (relying on the environment to provide information about the freshness constraints generated). We have chosen to develop the algorithm in this way because it is simpler, but it is possible to solve the $\approx_\alpha$-constraints and the generated freshness constraints separately, while keeping the same complexity. Instead of using an environment with a permutation and a set of atoms, we can split the environment into two parts, one storing just a permutation and the other just a set of atoms. In phase 1, it is sufficient to reduce all $\approx_\alpha$-constraints without propagating any freshness constraints. That is, when we reach an abstraction $[a]s \approx_\alpha [b]t$, instead of adding a to the set of atoms in the environment and propagating it, we just put a tag $a\ \#$ on the current node of t. At the end of the $\approx_\alpha$-reduction phase, an environment (consisting just of a set of atoms) will be attached to the root of t and some nodes will be tagged with freshness constraints. It remains to propagate the environment with the set of atoms over t, adding a to the set each time we reach a node tagged by a.*

To obtain an efficient algorithm it is important to avoid doing several freshness checks of the form $(a \# b)_{a \in A}$, which would require time proportional to $|A|$. Instead, we should do just one check $b \in_? A$, which can be done in constant time using an array.

Alpha-equivalence To check the validity of an $\approx_\alpha$-constraint, after running the core algorithm we have to normalise the problem using the rule (α), as described in Section 9.3.

If the right-hand sides of $\approx_\alpha$-constraints are ground, the core algorithm is sufficient and it is linear (ground case). Otherwise, each application of the rule (α) requires to know the support of a permutation, which we do because supports are always created with permutations and maintained when they are updated. Thanks to the use of functional maps, the support is copied in constant time when the permutation is copied, therefore the algorithm is also log-linear in the size of the problem in the non-ground case.

Matching The algorithm to solve matching constraints consists of the core algorithm, followed by a normalisation phase in which the result of the core algorithm is reduced using a rule that deals with variables occurring multiple times in the pattern (called $_?\approx$ in Section 9.4). In the case of linear matching this rule is not needed – the core algorithm is sufficient.

In Section 10 we discussed the implementation of the rule $_?\approx$ using an array S indexed by variables and a rule which we called *Rl-Check-Subst*. The construction of S requires the traversal of the term s and every term in the output of the core algorithm. This is done in time proportional to the size of the output of the core algorithm. At worst, the size is $|\mathrm{M}_{A_0}| \times \mathrm{MV}(t) + |s \approx_\alpha t|$ because phase 4 can add a suspended permutation and freshness constraints on every variable occurring in t. Therefore the output can be quadratic in the size of the input.

Then *Rl-Check-Subst* will compute $\overline{S[X_i] \approx_\alpha t_i}^{\approx\alpha}$ for each constraint $X_i \approx_\alpha t_i$ in the result of the core algorithm. Phase 1 to 3 are linear in its size and phase 4 has a complexity $\vartheta(|\mathrm{M}_{A_0}| \times \mathrm{MV}(t_i))$, hence at worst quadratic in time in the size of the input problem. The worst case complexity arises when phase 4 suspends permutations on all variables, making the output terms bigger than the input ones. Since permutations are bounded in size, in the worst case (an input problem with no permutations but many variables and abstractions), the output of phase 4 may be quadratic in the size of the input problem. On the other hand, if the input problem already has in each variable a permutation of size $|\mathrm{M}_{A_0}|$ (i.e. variables are 'saturated' with permutations), then, since permutations cannot grow, the α-equivalence and matching algorithms are linear even using arrays.

Note that the representation of a matching problem or an α-equivalence problem using higher-order abstract syntax does saturate the variables (they have to be applied to the set of atoms they can capture). Thus, our algorithms (whether for ground or non-ground, linear or non-linear problems) are all linear in time with respect to the size of the problem represented in higher-order abstract syntax. The table below summarises the results:

Case	Alpha-equivalence	Matching
Ground	linear	linear
Non-ground and linear	log-linear	log-linear
Non-ground and non-linear	log-linear	quadratic

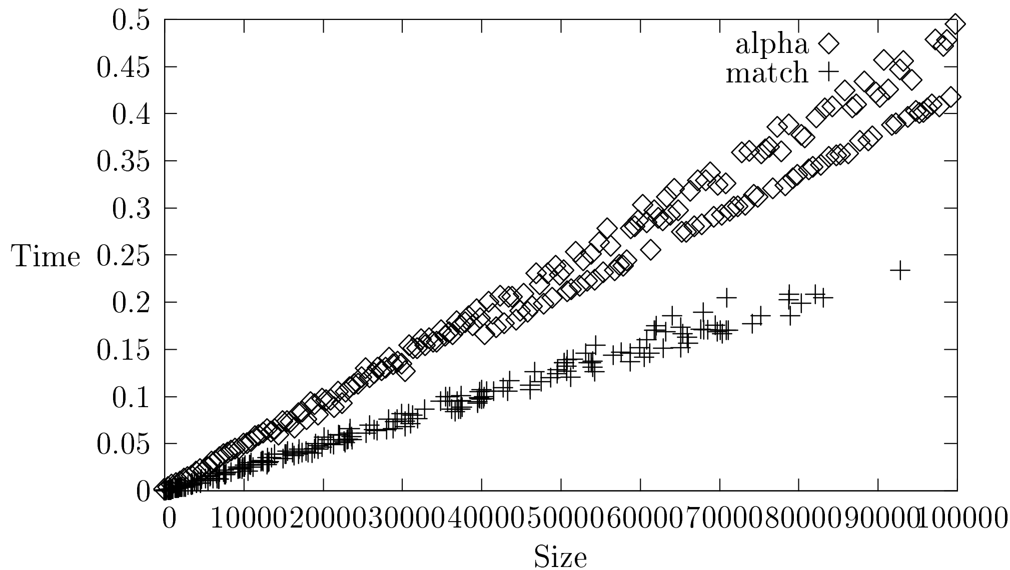


Figure 10.1: Benchmarks

10.6 Benchmarks

The algorithms described above to check α-equivalence and to solve ground matching problems have also been implemented in OCAML [6], using arrays.

In Figure 10.1, we show benchmarks generated by building random solvable ground problems (i.e., problems that do not lead to $\perp$) and measuring the time taken by the α-equivalence and matching algorithms to give the result (marked as $\diamond$ and $+$ in the graph)[1].

The benchmarks suggest that for problems of similar size, the α-equivalence algorithm takes more time than the matching algorithm. This may be explained by the fact that to check α-equivalence we need to traverse the whole problem, whereas a full traversal is not always needed to solve a matching problem. Note that, unlike first-order matching, nominal matching might produce freshness constraints that have to be checked by traversing the term.

[1] The program is available from: `www.dcs.kcl.ac.uk/staff/maribel/CANS`

Chapter 11

A Simple Nominal Rewriting Framework

In this chapter, we recall the definition of nominal rewriting systems [20] and we show that by only changing the interpretation of the freshness context layer, the nominal matching algorithm can become a nominal rewriting algorithm.

11.1 Nominal rewriting

A **nominal rewriting rule** is a tuple (Δ_{rl}, l, r), written $\Delta_{rl} \vdash l \to r$, where Δ_{rl} is a freshness context and l,r are nominal terms such that $V(r, \Delta_{rl}) \subseteq V(l)$.

Intuitively, in a rewriting rule, variables represent unknown terms and the rewrite relation is generated by instantiating the variables in the rules using substitutions. For example, the two rules

$$a\#X \vdash (\lambda[a]X)Y \to X \quad \text{and} \quad a\#Y \vdash (\lambda[a]Y)X \to Y$$

have different syntax but should generate the same rewrite relation. Atoms are not substituted, but they can be swapped: We write $R^{(a\,b)}$ for that rule obtained by swapping a and b in the rule R throughout. For example, if $R \equiv a\#X \vdash [a]X \to X$ then $R^{(a\,b)} \equiv b\#X \vdash [b]X \to X$. In general, we write R^{π} for that rule obtained by applying the permutation π to the atoms in R.

A set of rewrite rules is **equivariant** when it is closed under $(-)^{(a\,b)}$ for all atoms a and b.

Definition 81 *A* ***nominal rewriting system*** $\mathcal{R}$ *is an equivariant set of nominal rewriting rules.*

Nominal rewriting rules generate a rewrite relation on "nominal terms in context". We denote a term t with a freshness context Δ_t by $\Delta_t \vdash t$. In order to apply a rewrite rule at the root position in t, we use nominal matching. Not only l has to match t, but also the constraints in Δ_{rl} have to be satisfied by Δ_t. For example, consider the rule $a \# X \vdash f\ X \to X$ and let $f\ a$ be the term to rewrite. In this case, even if $f\ X$ and $f\ a$ match (with the substitution $\sigma = [X \mapsto a]$) the rule cannot be applied because a is not fresh in $X\sigma$.

Formally, given a nominal rewrite rule $\Delta_{rl} \vdash l \to r$ and a term t with freshness context Δ_t, we say that $\Delta_t \vdash t$ matches the left-hand side l of the rule under the constraints Δ_{rl} if and only if the matching problem $l \approx_\alpha t$ has a solution $(\sigma, \Delta_{\approx_\alpha})$ and $\Delta_{\approx_\alpha} \cup \overline{Env(\sigma(\Delta_{rl}))}^{\approx_\alpha} \subseteq \Delta_t$, where $\sigma(\Delta_{rl}) = \{a\#X\sigma \mid a\#X \in \Delta_{rl}\}$. The result of applying this rewrite rule to $\Delta_t \vdash t$ is then the term $\Delta_t \vdash r\sigma$.

Note that since nominal rewriting systems are equivariant, in order to decide whether there exists a rule in the system that matches a term at a given position we need equivariant nominal matching (i.e., nominal matching with respect to variants of the rules obtained by changing the names of the atoms). However, nominal matching is sufficient if the rules are *closed*. Closed nominal rules are equivalent to the rules used in standard notions of higher-order rewriting (for instance, any Combinatory Reduction System [31] can be defined as a closed nominal rewriting system, see [21]). Equivariant nominal matching, which is an NP-complete problem in general [12], can be avoided for closed rules, simply by using a copy of the rewrite rule where all the variables and atoms are chosen to be different from the ones in the term to be rewritten (this is always possible because nominal rewriting systems are equivariant). We refer the reader to [20] for the formal definition and properties of closed nominal rewriting.

For example, the η-rule in the λ-calculus is usually written informally as a rule scheme

$$\lambda x.Mx \to M \qquad \text{if } x \notin FV(M)$$

We can specify η using a nominal rewriting system defined by the (closed) rule

$$a\#X \vdash \lambda([a]Ap(X,a)) \to X$$

and then use this system to rewrite the term $\lambda([b]Ap(a,b))$ in an empty freshness context as follows:

1. We choose a fresh copy of the rewrite rule that does not use any variables or atoms occurring in the term to be rewritten. For instance, we take

$$c\#X \vdash \lambda([c]Ap(X,c)) \to X$$

2. We solve the matching problem:

$$\lambda([c]Ap(X,c)) \approx_\alpha \lambda([b]Ap(a,b))$$

   which has solution $([X \mapsto a], \emptyset)$. Since $c\#a$ is valid, our term rewrites to the term a, as expected.

11.2 A nominal rewriting algorithm: two improvements on the matching algorithm

To check whether $\Delta_t \vdash t$ matches $\Delta_{rl} \vdash l \to r$ efficiently, we will use a modified version of our matching algorithm. Since checking $\Delta_{\approx_\alpha} \cup \overline{Env(\sigma(\Delta_{rl}))}^{\approx_\alpha} \subseteq \Delta_t$ can be expensive because of the set unions and α-equivalence problems it involves, we will do these operations on-the-fly, directly in the matching algorithm.

For this, the first modification consists of computing $\overline{Env(\sigma(\Delta_{rl}))}^{\approx_\alpha}$ as we traverse the term, so that it will directly be in $\Delta_{\approx_\alpha}$.

Below we use the notation Δ_X for the set of atoms in constraints relating to X in a freshness context Δ, more precisely:

$$\Delta_X = \{a \mid a\#X \in \Delta\}$$

Proposition 64 *If for each equation $X \approx_\alpha \xi\Diamond t$ at the end of phase 2, we add the set $\xi_\pi^{-1}(\Delta_{rlX}) = \{\xi_\pi^{-1}\cdot a \mid a \# X \in \Delta_{rl}\}$ to the current freshness set ξ_A, this is equivalent to computing $\overline{Env(\sigma(\Delta_{rl}))}^{\approx_\alpha} \cup \Delta_{\approx_\alpha}$.*

In fact, it is sufficient to do this once for each distinct variable X.

Proof By Proposition 52, at the end of phase 2, the problem, if it has a solution, is of the form $(X_i \approx_\alpha \xi_i \lozenge t_i)_1^n$. Let $X_j \approx_\alpha \xi_j \lozenge t_j$ be one of these constraints. Notice that for each i such that $X_i = X_j$, $\xi_{i\pi} \cdot t_i$ and $\xi_{j\pi} \cdot t_j$ must be α-equivalent for the problem to have a solution σ. In addition to satisfying $X_j \sigma \approx_\alpha (\xi_{j\pi} \cdot t_j)$, σ must also satisfy $\Delta_{rlX_j} \# X_j \sigma$.

Since $\Delta_{rlX_j} \# X_j \sigma$ is equivalent to $\Delta_{rlX_j} \# \xi_{j\pi} \cdot t_j$, which is in turn equivalent to $\xi_{j\pi}^{-1}(\Delta_{rlX_j}) \# t_j$, any solution σ of the original problem is also a solution of:

$$X_j \approx_\alpha (\xi_{j\pi}, \xi_{jA} \cup \xi_{j\pi}^{-1}(\Delta_{rlX_j})) \lozenge t_j$$

Furthermore, since $\xi_{j\pi} \cdot t_j \approx_\alpha \xi_{i\pi} \cdot t_i$ when $X_i = X_j$, we also have

$$\Delta_{rlX_j} \# \xi_{j\pi} \cdot t_j \Leftrightarrow \Delta_{rlX_j} \# \xi_{i\pi} \cdot t_i$$

which means we only need to do it once for each distinct variable. $*$

The second improvement consists of computing $\Delta_{\approx_\alpha} \subseteq \Delta_t$ directly without computing $\Delta_{\approx_\alpha}$. For this, in phase 4, when reaching $\xi \lozenge X$, instead of adding $\xi_A \# X$ to $\Delta_{\approx_\alpha}$ we directly check if $\xi_A \subseteq \Delta_{tX}$.

Proposition 65 *In phase 4, when reaching $\xi \lozenge X$, checking whether $\xi_A \subseteq \Delta_{tX}$ is equivalent to adding $\xi_A \# X$ to $\Delta_{\approx_\alpha}$ and then checking $\Delta_{\approx_\alpha} \subseteq \Delta_t$.*

Proof Phase 4 computes $\Delta_{\approx_\alpha}$ by generating constraints $\xi_A \# X$ for each $\xi \lozenge X$ in the problem. Therefore, checking $\Delta_{\approx_\alpha} \subseteq \Delta_t$ is equivalent to checking whether every $\xi_A \# X \subseteq \Delta_t$, which is equivalent to $\xi_A \subseteq \Delta_{tX}$. $*$

These two modifications can actually be made without changing the phases. All we need to do it to interpret FreshnessContextCall differently. The interpretation of FreshnessContextCall to match terms in context is described in the algorithm 18.

Because we no longer need to copy freshness sets, we can always use mutable arrays to represent ξ_A. Unfortunately we still need to copy permutations ξ_π, so in the following we will use mutable arrays for sets, and persistent maps for permutations. We will represent Δ_{rl} using an array indexed by variables, such that the element X in the array will contain the list of atoms that should be fresh for X according to Δ_{rl}.

```
1  interpretFreshContextCheck(c) = ;
2    switch c do
3      case FreshConstraint A # X k
4        if A ⊈ Δ_{t_X} then
5          dthrowError("Constraints not satisfied")
6        end
7        ;
8        k(())
9      case FreshContextRule X k
10       S = Δ_{rl_X} ;
11       remove Δ_{rl_X} from Δ_{rl} ;
12       k(S)
13     case FreshToList k
14       k([])
15
16   endsw
```

Algorithm 18: Interpretation of Freshness for matching terms in context

11.3 Complexity of nominal rewriting

We start by defining the size of a matching problem for terms in context.

Definition 82 *The size of a matching problem* $Pr = \Delta_{rl} \vdash l \approx_\alpha \Delta_t \vdash t$ *is defined as*

$$|Pr| = |\Delta_{rl}| + |l| + |\Delta_t| + |t|$$

where the sizes of Δ_{rl} *and* Δ_t *are the size of their representations.*

Proposition 66 *The total cost of the algorithm to solve the matching problem for terms in context:*

$$\Delta_{rl} \vdash l \approx_\alpha \Delta_t \vdash t$$

is at most

$$\vartheta((|\Delta_{rl}| + |l| + |t|) \times \log |A| + m \times |MVar(t)|)$$

where A *is the set of atoms occurring in the problem,* $m = max_X(|\Delta_{tX}|)$ *and* $MVar(t)$ *is the multiset of variable occurrences in* t.

Proof With this interpretation of FreshContextCall, $FreshContextRule$ adds $\xi_\pi^{-1}(\Delta_{rlX})$ to the current freshness set, once for each variable X. Freshness sets being coded as mutable arrays and permutations as functional maps, it takes $\vartheta(\log |A|)$ time to access each atom a in Δ_{rlX} (coded as a list of atoms), compute its image by ξ_π^{-1} and add it to ξ_A. The cost of Phase 2 is therefore $\vartheta(|\Delta_{rl}| \times \log |A|)$.

Phase 4, when reaching $\xi \Diamond X$, rewrites it into $\xi_\pi \cdot X$ and checks whether $\xi_A \subseteq \Delta_{tX}$. The former is done in constant time thanks to the use of functional maps. The latter, if done in a naive way, would take time proportional to $|A|$ since ξ_A is represented as an array of size $|A|$. However, checking $\xi_A \subseteq \Delta_{tX}$ is equivalent to checking $|\Delta_{tX} \cap \xi_A| = |\{a \mid a \in \xi_A\}|$. We call this number s_{ξ_A}. We can, as we did for permutations, compute this number when ξ_A is created and then update it in constant time on every update of ξ_A. In this way, we only need to compute the number of atoms in Δ_{tX} that are also in ξ_A and compare it with s_{ξ_A}, which can be done in time proportional to Δ_{tX} (because Δ_{tX} is a list of atoms and ξ_A an array). Thus, for any X, we can bound the cost by m, so in the worst case the total cost is $m \times MVar(t)$.

Summarising:

$$\begin{array}{ll} \text{Phase 1} & : \vartheta(|l| \times \log|A|) \\ \text{Phase 2} & : \vartheta(|l| \times \log|A|) \\ \text{Phase 2}' & : \vartheta(|\Delta_{rl}| \times \log|A|) \\ \text{Phase 3} & : \vartheta(|t| \times \log|A|) \\ \text{Phase 4}' & : \vartheta(m \times |MVar(t)|) \end{array}$$

$*$

11.3.0.1 Special cases

First-order matching is a particular case of nominal matching, and it is interesting to see that the rewriting algorithm specified above behaves as expected when the terms involved are ground or first-order.

Proposition 67 *If t is ground or if Δ_t is empty, the complexity of the algorithm is at most $\vartheta((|\Delta_{rl}| + |l| + |t|) \times \log|A|)$.*

Proof If t is ground, $|MVar(t)| = 0$ and if Δ_t is empty, $m = 0$. $*$

Proposition 68 *For first-order terms, where t is ground, the algorithm is linear in time.*

Proof In this case there are neither variables ($|MVar(t)| = 0$) nor atoms ($|A| = 0$). $*$

11.4 The Zipper Layer

A **zipper** is a data structure that has been introduced by *Huet* [26] to represent a pointer within a term in a purely functional way. There are many implementations of zippers. The *Haskell* module Zipper in appendix A.11 presents a very simple implementation of a generic zipper on algebraic datatypes. This section presents the zipper on compact nominal terms using the implementation in appendix A.11.

Let t be the compact nominal term $[a]((a, b), [b]X)$. We want a simple way to navigate withing t in a purely functional way.

Definition 83 *Let H be a new variable symbol ($H \notin \mathcal{X}$). Until now we have only considered compact nominal terms whose variables where in $\mathcal{X}$. We will keep calling compact nominal terms such terms. Compact nominal terms c over $(\mathcal{X} \cup \{H\})$ with exactly one occurrence of X are called* ***compact nominal context*** *or* ***context*** *for short. H is called a* ***hole****.*

For example, $[a](X, H)$ is a context but $[a](X, X)$ and $[a](H, H)$ are not. The following equalities are valid:

$$\begin{aligned} t = \; & H[X \mapsto t] \\ & [a]H[H \mapsto ((a, b), [b]X)] \\ & ([a](H, [b]X))[H \mapsto (a, b)] \\ & [a]((a, b), [b]H)[H \mapsto X] \end{aligned}$$

As we can see any can see any compact term u can be represented as a pair (c, u') where c is a context and u' a subterm of u and $u = c[H \mapsto u']$. Furthermore let c_1 and c_2 two contexts, $c_1[H \mapsto c_2]$ is also a context.

Definition 84 *A* ***zipper*** *z on a nominal term is a pair $(t, [c_1, \ldots, c_n])$ where t is a nominal term and $[c_1, \ldots, c_n]$ a list of contexts. z represents the term $c_n[H \mapsto c_{n-1}[H \mapsto \ldots c_1[H \mapsto t]]]$. t is called the* ***subterm of the zipper***

Let $z = ([a]u, [c_1, \ldots, c_n])$ be a zipper. We can move a step downward by the new zipper $z' = (u, [[a]H, c_1, \ldots, c_n])$. Let $z = ((u_1, u_2), [c_1, \ldots, c_n])$, there are two downward move possible: moving to u_1 by $z' = (u_1, [(H, u_2), c_1, \ldots, c_n])$ or to u_2 by $z' = (u_2, [(u_1, H), c_1, \ldots, c_n])$. Going a step upward from $z = (u, [c_1, \ldots, c_n])$ is as simple as $z' = (c_1[H \mapsto u], [c_2, \ldots, c_n])$ if $n > 0$.

It also possible to update the subterm of a zipper. Let $z = (u, [c_1, \ldots, c_n])$. We can replace u by t by $z' = (t, [c_1, \ldots, c_n])$.

The *Haskel* module *Zipper* in appendix A.11 implements a state monad whose state is a zipper. Along with the rewrite function it makes a very simple but complete rewriting framework. Any stategy can be written in *Haskell*, using the zipper to navigate within the term and the rewrite function to rewrite at the desired position.

Part IV

Conclusions

Related Works

On Nominal Algorithms *Gabbay*, *Pitts* and *Urban* [51] were the first to prove that nominal unification is decidable. They presented a simple algorithm based on problem rewriting. Unfortunately, because subterms were not shared, the algorithm is exponential in time and space.

Cheney proved that **equivariant unification** [11] (deciding whether two terms involving swapping can be made equal "up to a permutation") is NP-complete.

Higher-Order Patterns **Higher-Order Patterns** syntax is another syntax, introduced by *Miller* [35] to represent systems with binders. Higher-Order Patterns are λ-terms in η-long forms, in which free variables F only occur in the form $F(\overline{x_k})$ with $\overline{x_k}$ is a sequence of k ($k > 0$) distinct bound variables.

For example, $\lambda x.\lambda y.\lambda z.F(y, x)$ and $\lambda x.\lambda y.y(\lambda z.F(z, y), F(x, y), G(y))$ are higher-order patterns, provided that they are η-long forms. But $\lambda x.F(c, x)$, $\lambda x.\lambda y.F(x, x)$ and $\lambda z.F(G(x))$ are not.

Higher-order patterns and nominal unification are strongly related. Cheney [13] showed that higher-order pattern unification can be reduced to nominal unification. Latter Levy and Villaret [33] proved that nominal unification can be reduced to higher-order pattern unification. Their reduction proves that nominal unification can be decided in quadratic deterministic time. Their proof is based on Qian's linear higher-order pattern algorithm [45]. This algorithm is intricate and, to our knowledge, has never been implemented (certainly because of its intricacy), whereas *QNU* (presented in chapter 6) is simple, based on a well known first-order algorithm and has been implemented.

Qian's algorithm is linear in time and space on the size of the input problem whereas the nominal unification algorithm is quadratic in time and space in the size of the input problem. This does not mean that Qian's algorithm is faster because the syntax is different. Nominal syntax can be much more concise than higher-order patterns. The reason of that lies with the variables. In nominal syntax a variable is just X and there is no restriction on the possible free variable of X. This is not the case with higher-order patterns where a variable X is always followed by the list of its possible free atoms (called bound variables in higher-order pattern terminology). It means that if A_0 is the set of atoms in the problem, the same problem in higher-order pattern syntax can be $\mid A_0 \mid$ times bigger than expressed in nominal syntax. Furthermore, as seen in chapter 6, when all variables in a nominal problem are suspended, then unification become also linear.

First-class monadic signature The idea of using continuations to stop the computation to return value and a computation is not new. It has been used in many situations: *Claessen* used them to implement a concurrency monad [14], *Kiselyov*, *Shan* and *Sabry* showed they can be used in operating systems to implement system calls [30], Filinski used them to represent monads and monadic towers [22, 23], they have been used to implements streams, ...

To our knowledge it has never been used to implement monadic classes. *Kiselyov*, *Shan* and *Sabry*'s approach is the closest to ours as system calls and monadic operations are related (the operation system can be seen as a monad). However, unlike their, our concern is not only to implement system calls but to have a correct and complete way to implement monadic classes. Furthermore their calls are only monomorphic and do not have arguments of the same monad (the example of section 3.3) whereas we presented how to implement monomorphic and polymorphic calls even in that case. Their approach works fine for one layer but problems arise with more: our approach explicits the number of layers and the type of each layer. With more typing information, programs are easier to read, to develop and overall safer. The *prompt not defined error* that can occur at run time with their approach is impossible with ours. Every level error is detected by the type checker at compile time. Furthermore our approach is based on the

CPS Hierarchy, which is simple and well known [9, 18] whereas they rely on a complex monadic framework [19].

Future Works

The main problem when implementing an algorithm in a syntax involving binders is the amount of technical details needed everywhere like freshness check or applying the right permutation at the right place at the right time. Our modular approach has two main benefits:

- most of these details are handled automatically by the corresponding layer
- instead of interacting in a chaotic way, nominal aspects (naming, handling of the context, actual implementation, combing the phases, ...) are separated so that it is easier to check each of them is OK.

The idea of *environment* shows an important aspect of syntax involving aspect. On the contrary to first-order syntax where a term comes alone, nominal terms are the combination of a term and a position map. This position map called environment is the link between names and positions. For example: let us consider two nominal terms $[a]a$ and $[b]b$. a (resp. b) is just the name of the abstraction in $[a]a$ (resp. $[b]b$). The two abstractions have the very same position, only their name is different. All nominal terms come with such a map and the complexity of nominal algorithms comes from translating position names from terms to terms.

The reason why the nominal α-equivalence and matching algorithms can be more efficient than unification whereas in first-order syntax they have all the same complexity is that for α-equivalence and matching algorithms, the closer two nodes are in the directed acyclic graph the closer are their environments. That is not true for unification. The key to a faster than $\mid A_0 \mid \times Pr_0$ (where A_0 is the set of atoms in the problem Pr_0) complexity algorithm for nominal unification

is either to be able to compute the suspended permutation and set of freshness constraints at any node in less time than $\mid A_0 \mid$ or to prove that knowing it for a smaller subset of nodes is sufficient. Unfortunately the dependency relation between permutations and freshness set in a unification problem is chaotic and depend only on the problem.

A possibility could be to have a less chaotic dependency. Let us consider the term $[a](X,[a]X)$. This term means explicitly that a free occurrence of a in X in bound by two different abstractions. Having $[a](a,[a]a)$ is not annoying because the two a are not the same occurrence and $[a](a,[a]a)$ is equivalent to $[a](a,[b]b)$ but in the $[a](X,[a]X)$, the two occurrences of X represent the very same term. Do we need in practice to have many different environment in a term? Maybe not.

Let $t = [a](X,((f,X),[b]y))$. If c is an free atom in X all the occurrences of c in $t[X \mapsto c]$ will be either all bound and bound by the same abstraction or all free. Let $u = (X,[a]X)$ and $X = a$, a in at the same time free and bound in $u[X \mapsto a]$. More generally, when in a term s there is a variable X such that a free atom a in X can be bound by different abstractions or at the same time free and bound, then u is said to be **schizophrenic**.

Definition 85 *The **anti-schizophrenic restriction** is: let t be a nominal term and $X \in V(t)$. For any atom a, if two occurrences of a (from X) in $t[X \mapsto a]$ are under two different abstractions, or one free in $t[X \mapsto a]$ and not the other, then a has to be fresh in X.*

This restriction ensures that all occurrences of X in t have the same environment. Because of this, when reducing a nominal problem $\pi{\cdot}X \approx_\alpha s \quad \pi'{\cdot}X \approx_\alpha t$ we know that $ds(\pi,\pi') \# X$ so we can directly write $s \approx_\alpha t$. If managing the set of freshness constraints for every variable can be done efficiently, then we conjecture a better than quadratic unification algorithm can be found under this restriction

Conclusion

We showed in this thesis that even if nominal algorithms may seem simple at first look, developing and implementing efficient nominal algorithms requires powerful theoretical and practical tools. Their complexity depends a lot on the implementation of permutations and sets and on the strategy used. Furthermore there are not (yet) a unique implementation and strategy that are the best for all the situations. The algorithms need to adapt themselves to the input problem to perform specific optimisations.

This complexity is not specific to nominal algorithm. Even if a linear algorithm to solve Higher-Order Pattern Unification exist since 1993 [44], to our knowledge it has never been implemented in practice. Practical higher-order pattern unification implementations, like *Teyjus* [7, 37], seem to prefer using a simpler, but less efficient, algorithm. The unification algorithms presented in part II are simple and efficient.

The α-equivalence and matching algorithms presented in part III are the most efficient known algorithm for these problems. On linear matching problems without permutations, the nominal algorithm complexity is log-linear, but the size of these problems in higher-order pattern syntax can be $\mid A_0 \mid$ (where A_0 is the set set of atoms in the problem) times bigger. So even if higher-order pattern matching is linear, on these problems, it is less efficient than nominal matching.

The techniques developed in the thesis are interesting results by themselves. The nominal union-find algorithm and the theory behind can be useful in any situation where a union-find algorithm able to handle names is required. The permutations, sets of atoms, freshness contexts and environment monadic layers can be used as-is in any program manipulating nominal terms thus enabling it to

use several actual implementations of permutations and sets of atoms, different interpretations of freshness context calls and one single environment.

First-class monadic signatures can be partially implemented in other languages than *Haskell* thus bringing monadic classes to theses languages. For languages with native first-class continuations (*Scheme*, *Ruby*, *Objective CAML (bytecode)*, ...) it can even be done transparently using Filinski's implementation of shift / reset in terms of callcc [22].

Bibliography

[1] Brics website:. http://www.brics.dk/. 11

[2] Ghc documentation:. http://www.haskell.org/ghc/docs/latest/html/. 15

[3] Haskell 98 revised report:. http://www.haskell.org/onlinereport/. 15

[4] Haskell website:. http://haskell.org/. 15

[5] Hnt website:. http://www.brics.dk/. 13

[6] Objective caml website:. http://www.dcs.kcl.ac.uk/pg/calves/hnt/. 69, 155

[7] Teyjus website:. http://teyjus.cs.umn.edu/. 170

[8] S. Adams. Functional Pearls Efficient setsâĂŤa balancing act. *Journal of Functional Programming*, **3**(04):553–561, 2008. 59

[9] M. Biernacka, D. Biernacki, and O. Danvy. *An operational foundation for delimited continuations in the CPS hierarchy*. BRICS. 32, 167

[10] C. Calvès and M. Fernändez. Nominal matching and alpha-equivalence. *Logic, Language, Information and Computation. LNCS*, **5110**:111–122. 128

[11] J. Cheney. The complexity of equivariant unification. *Lecture notes in computer science*, pages 332–344, 2004. 165

[12] J. Cheney. Equivariant unification. *Lecture Notes in Computer Science*, **3467**:74–89, 2005. 157

[13] J. Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*, pages 104–119. Citeseer, 2005. 165

[14] K. Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, **9**(03):313–323, 1999. 166

[15] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM New York, NY, USA, 1990. 20

[16] O. Danvy and K. Malmkjær. *On the Idempotence of the CPS Transformation*. BRICS, Computer Science Department, University of Aarhus. 20

[17] O. Danvy and K. Millikin. Refunctionalization at work. *Science of Computer Programming*, **74**(8):534–549, 2009. 21

[18] O. Danvy and Z. Yang. An operational investigation of the CPS hierarchy. *Lecture notes in computer science*, pages 224–242, 1999. 32, 167

[19] R.K. Dyvbig, S.P. Jones, and A. Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, **17**(06):687–730, 2007. 167

[20] M. Fernandez and M.J. Gabbay. Nominal rewriting. *Information and Computation*, **205**(6):917–965, 2007. 99, 134, 156, 157

[21] M. Fernández, M.J. Gabbay, and I. Mackie. Nominal rewriting systems. In *ACM Symposium on Principles and Practice of Declarative Programming (PPDPâĂŹ04), ACM Press*, 2004. 157

[22] A. Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457. ACM New York, NY, USA, 1994. 166, 171

[23] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 175–188. ACM New York, NY, USA, 1999. 33, 166

[24] M.J. Gabbay and A.M. Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, **13**(3):341–363, 2002. 10

[25] P. Homeier. A proof of the Church-Rosser theorem for the lambda calculus in higher order logic. In *TPHOLs*, pages 207–222, 2001. 139

[26] G. Huet. The zipper. *Journal of Functional Programming*, **7**(05):549–554, 1997. 162

[27] M. Jaskelioff. Monatron: An Extensible Monad Transformer Library. 33

[28] M. Jaskelioff. Modular Monad Transformers. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 64–79. Springer, 2009. 25, 33

[29] M.P. Jones. Functional programming with overloading and higher-order polymorphism. *Lecture Notes in Computer Science*, **925**:97–136, 1995. 30

[30] O. Kiselyov, C. Shan, and A. Sabry. Delimited dynamic binding. In *Proceedings of the 2006 ICFP conference*, **41**, pages 26–37. ACM New York, NY, USA, 2006. 166

[31] J.W. Klop, V. Oostrom, and F. Raamsdonk. Combinatory reduction systems: introduction and survey. *TCS*, **121**(1&2):279–308, 1993. 157

[32] P.J. Landin. Getting rid of labels. *Report, UNIVAC Systems Programming Research*, 1965. 19

[33] Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. 2008. 165

[34] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **4**(2):258–282, 1982. 80, 86

[35] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, **1**(4):497–536, 1991. 165

[36] E. Moggi. Notions of computation and monads. *INF. COMPUT.*, **93**(1):55–92, 1991. 19

[37] G. Nadathur and D.J. Mitchell. System description: Teyjus-a compiler and abstract machine based implementation of lambdaprolog. *Lecture notes in computer science*, pages 287–291, 1999. 170

[38] MHA Newman. On theories with a combinatorial definition of" equivalence". *Annals of Mathematics*, pages 223–243, 1942. 130, 134, 138

[39] J. Nievergelt and EM Reingold. Binary search trees of bounded balance. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 137–142. ACM New York, NY, USA, 1972. 59

[40] MS Paterson and MN Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186. ACM New York, NY, USA, 1976. 69, 70, 72

[41] S.L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987. 10

[42] A.M. Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, **186**(2):165–193, 2003. 10

[43] D. Plump. Term graph rewriting. *Handbook of Graph Grammars and Computing by Graph Transformation, Applications, Languages and Tools*, **2**:3–61. 67

[44] Z. Qian. Linear unification of higher-order patterns. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 391–391, 1993. 170

[45] Z. Qian. Unification of higher-order patterns in linear time and space. *Journal of Logic and Computation*, **6**(3):315–341, 1996. 165

[46] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference (ACM'72)*, pages 717–740, 1972. 20, 21

[47] N. Shankar. *Metamathematics, machines and Gödel's proof.* Cambridge Univ Pr, 1997. 139

[48] M.R. Shinwell, A.M. Pitts, and M.J. Gabbay. FreshML: Programming with binders made simple. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 263–274. ACM New York, NY, USA, 2003. 10

[49] J. Stern. Fondements mathématiques de l'informatique. *Ediscience International*, 1990. 118

[50] R.E. Tarjan. On the efficiency of a good but not linear set union algorithm. 1972. 80

[51] C. Urban, A.M. Pitts, and M.J. Gabbay. Nominal unification. *Theoretical Computer Science*, **323**(1):473–498, 2004. 50, 51, 53, 54, 99, 131, 134, 165

Appendix A

Haskell Code

A.1 Base definitions

```
1  {-# LANGUAGE RankNTypes,
2         FlexibleInstances,
3         MultiParamTypeClasses,
4         NoMonomorphismRestriction,
5         KindSignatures,
6         ExistentialQuantification,
7         GeneralizedNewtypeDeriving #-}
8
9
10
11 module Base where
12
13 import Control.Monad
14 import Control.Monad.State
15
16
17 liftStateT :: (Monad m) => m a -> StateT s m a
18 liftStateT = lift
19
20 mkFst :: (a -> c) -> (a, b) -> (c, b)
21 mkFst f (a,b) = (f a, b)
22
23
24 mkSnd :: (b -> d) -> (a, b) -> (a, d)
```

```
25 mkSnd f (a,b) = (a, f b)
26
27 -- Linear Continuation
28
29 newtype LContT m a = LContT { unLContT ::
30       forall r . (a -> m r) -> m r }
31
32 instance (Monad m) => Monad (LContT m) where
33   return a           = LContT $ \k -> k a
34   (LContT m) >>= f   = LContT $ \k -> m (\x -> unLContT (f x) k)
35
36 runLContT :: (Monad m) => LContT m a -> m a
37 runLContT (LContT m) = m return
38
39
40 liftLContT :: (Monad m) => m a -> LContT m a
41 liftLContT m = LContT $ \k -> m >>= k
42
43 instance MonadTrans LContT where
44  lift = liftLContT
45
46 -- ErrorT of the mtl
47
48 newtype EitherT error m a =
49     EitherT { unEitherT ::m (Either error a) }
50
51 instance (Monad m) => Monad (EitherT error m) where
52     return              = EitherT . return . Right
53     (EitherT m) >>= f   = EitherT $ do v <- m
54                                        case v of
55                                           Left  e -> return $ Left e
56                                           Right a -> unEitherT $ f a
57
58
59
60 liftEitherT m = EitherT $ m >>= return . Right
61
62 instance MonadTrans (EitherT error) where
63   lift = liftEitherT
64
65 throwEitherT :: (Monad m) => error -> EitherT error m a
```

```
66 throwEitherT error = EitherT $ return $ Left error
67
68 catchEitherT :: (Monad m) =>
69           EitherT error m a
70      -> (error -> EitherT error m a)
71      -> EitherT error m a
72 catchEitherT (EitherT m) h = EitherT $
73     do v <- m
74         case v of
75             Left  e -> unEitherT $ h e
76             Right a -> return $ Right a
77
78
79
80 -- Usefull printing
81
82 instance Show (a -> b) where
83      show _ = "<fun>"
```

A.2 Fist-Class Monadic Signatures

```
1  {-# LANGUAGE RankNTypes,
2                FlexibleInstances,
3                MultiParamTypeClasses,
4                NoMonomorphismRestriction,
5                GeneralizedNewtypeDeriving,
6                KindSignatures,
7                ExistentialQuantification #-}
8
9  module Call
10     ( call ,
11       callT ,
12       instanciateStream ,
13       instanciateStreamT ,
14       instanciate ,
15       instanciateT ,
16       instanciateNoT ,
17       IdentityT (..) ,
18       liftIdentityT ,
19       liftSContT ,
20       SCall ,
21       SCont ,
22       SCallT ,
23       SContT ,
24       stream ,
25       streamT,
26       clv'p1 ,
27       clv'p2 ,
28       clv'p3 ,
29       clv'p4 ,
30       clv'p5 ,
31       clv'p6 ,
32       clv'p7 ,
33       clv'p8
34     )
35   where
36
37
38 import Control.Monad
39 import Control.Monad.Trans
```

```
40
41
42
43 data SCall theclass finalvalue =
44      End finalvalue
45    | Call (theclass (SCont theclass) (SCall theclass finalvalue))
46
47 newtype SCont theclass a = SCont { unSCont ::
48   forall e. (a -> SCall theclass e) -> SCall theclass e }
49
50 instance Monad (SCont theclass) where
51   return a          = SCont $ \k -> k a
52   (SCont m) >>= f   = SCont $ \k -> m (\x -> unSCont (f x) k)
53   -- return a >>= f   âĽą  f a
54   -- m >>= return     âĽą  m
55   -- (m >>= f) >>= g  âĽą  m >>= (\x -> f x >>= g)
56
57 call ::    (forall r . (a -> r) -> thecall (SCont thecall) r)
58         -> SCont thecall a
59 call a = SCont $ \k -> Call (a k)
60
61
62 stream :: SCont theclass a -> SCall theclass a
63 stream m = unSCont m End
64
65
66 instanciateStream :: (Monad m) =>
67       (theclass (SCont theclass) (SCall theclass finalvalue)
68                                -> m (SCall theclass finalvalue))
69     -> SCall theclass finalvalue
70     -> m finalvalue
71 instanciateStream interpret = aux
72   where aux (End   r)  = return r
73         aux (Call  c)  = interpret c >>= aux
74
75 instanciate :: (Monad m) =>
76          (theclass (SCont theclass) (SCall theclass a)
77                                -> m (SCall theclass a))
78       -> SCont theclass a
79       -> m a
80 instanciate interpret t = instanciateStream interpret $ stream t
```

```
81
82
83
84
85
86 -- TRANSFORMERS
87
88
89 data SCallT theclass m e =
90       EndT   e
91     | CallT (theclass     (SContT theclass m)
92                           (m (SCallT theclass m e))
93             )
94
95 newtype SContT theclass m a = SContT { unSContT ::
96     forall e . (a -> m (SCallT theclass m e))
97                 -> m (SCallT theclass m e) }
98
99 instance (Monad m) => Monad (SContT theclass m) where
100  return x          = SContT $ \k -> k x
101  (SContT m) >>= f = SContT $ \k -> m (\x -> unSContT (f x) k)
102
103 liftSContT m = SContT $ \k -> m >>= k
104
105 instance MonadTrans (SContT thecall) where
106  lift = liftSContT
107
108
109
110 callT :: (Monad m) => (forall r .    (a -> r)
111                                   -> thecall (SContT thecall m) r
112                       )
113                    -> SContT thecall m a
114 callT a = SContT (\k -> return $ CallT (a k))
115
116
117 streamT :: (Monad m) =>       SContT theclass m a
118                            -> m (SCallT theclass m a)
119 streamT m = unSContT m (return . EndT)
120
121
```

```
122 instanciateStreamT :: (Monad m) =>
123      (theclass (SContT theclass t) (t (SCallT theclass t a))
124                                   -> m (SCallT theclass t a))
125      -> SCallT theclass t a
126      -> m a
127 instanciateStreamT interpret = aux
128   where aux (EndT   r) = return r
129         aux (CallT  c) = interpret c >>= aux
130
131
132
133
134 instanciateT :: (Monad (t m), MonadTrans t, Monad m) =>
135      (theclass (SContT theclass m) (m (SCallT theclass m a))
136                                  -> t m (SCallT theclass m a))
137    -> SContT theclass m a -> t m a
138 instanciateT interpret t = do c <- lift $ streamT t
139                               instanciateStreamT interpret c
140
141
142 clv'p1 = liftSContT
143 clv'p2 = clv'p1 . clv'p1
144 clv'p3 = clv'p1 . clv'p2
145 clv'p4 = clv'p1 . clv'p3
146 clv'p5 = clv'p1 . clv'p4
147 clv'p6 = clv'p1 . clv'p5
148 clv'p7 = clv'p1 . clv'p6
149 clv'p8 = clv'p1 . clv'p7
150
151 -- Identity Transformer
152
153 newtype IdentityT m a = IdentityT { runIdentityT :: m a }
154  deriving (Monad)
155
156 liftIdentityT :: m a -> IdentityT m a
157 liftIdentityT = IdentityT
158
159 instance MonadTrans IdentityT where
160  lift = liftIdentityT
161
162
```

```
163 instanciateNoT :: (Monad m) =>
164          (theclass (SContT theclass m) (m (SCallT theclass m a))
165                                       -> m (SCallT theclass m a))
166       -> SContT theclass m a -> m a
167 instanciateNoT interp m = streamT m >>= instanciateStreamT interp
```

A.3 Error Handling Layer

```
1  {-# LANGUAGE RankNTypes,
2                FlexibleInstances,
3                MultiParamTypeClasses,
4                NoMonomorphismRestriction,
5                KindSignatures,
6                ExistentialQuantification,
7                GeneralizedNewtypeDeriving #-}
8
9
10 module Error where
11
12 import Base
13 import Call
14
15 import Control.Monad
16 import Control.Monad.Identity
17 import Control.Monad.Trans
18
19
20 -- Error Class
21
22 data ErrorCall e (m :: * -> *) r =
23        forall a . RaiseError e                (a -> r)
24      | forall a . Try        (m a) (e -> m a) (a -> r)
25
26 raiseError :: (Monad m) => e -> SContT (ErrorCall e) m a
27 raiseError e = callT $ RaiseError e
28
29 try :: (Monad m) =>
30          SContT (ErrorCall e) m a
31      -> (e -> SContT (ErrorCall e) m a)
32      -> SContT (ErrorCall e) m a
33 try      m h = callT $ Try m h
34
35
36 try'lv :: (Monad m, Monad m1) =>
37       (SContT (ErrorCall e) m1 b -> m (m a)) -> b -> (e -> b) -> m a
38 try'lv lv m h = join $ lv $ try (return m) (return . h)
39
```

```
40
41 -- Monad Error Done Wright!
42
43 newtype ErrorT error m a = ErrorT {
44       unErrorT :: LContT (EitherT error m) a}
45  deriving (Monad)
46
47 liftErrorT :: (Monad m) => m a -> ErrorT error m a
48 liftErrorT = ErrorT . lift . lift
49
50 instance MonadTrans (ErrorT error) where
51     lift = liftErrorT
52
53 runErrorT :: (Monad m) =>      ErrorT error m a
54                             -> m (Either error a)
55 runErrorT = unEitherT . runLContT . unErrorT
56
57 raiseErrorT :: (Monad m) => error -> ErrorT error m a
58 raiseErrorT error = ErrorT $ lift $ throwEitherT error
59
60 tryErrorT :: (Monad m) =>        ErrorT error m a
61                              -> (error -> ErrorT error m a)
62                              -> ErrorT error m a
63 tryErrorT (ErrorT m) h = ErrorT $ LContT $ \k ->
64   catchEitherT (unLContT m k) (\e -> unLContT (unErrorT $ h e) k)
65
66
67
68 -- Interpretation of Call as ErrorT
69
70 interpretErrorT :: (Monad m) =>
71      ErrorCall e (SContT (ErrorCall e) m)
72                  (m (SCallT (ErrorCall e) m a))
73  -> ErrorT e m (SCallT (ErrorCall e) m a)
74 interpretErrorT (RaiseError e    k) =
75       raiseErrorT e >>= liftErrorT . k
76 interpretErrorT (Try          m h k) =
77       do r <- tryErrorT   ( instanciateT interpretErrorT     m)
78                           ((instanciateT interpretErrorT) . h)
79          liftErrorT $ k r
80
```

```
81 runErrorCallErrorT :: (Monad m) =>
82           SContT (ErrorCall e) m a -> m (Either e a)
83 runErrorCallErrorT = runErrorT . (instanciateT interpretErrorT)
```

A.4 Store Layer

```
1  {-# LANGUAGE RankNTypes,
2        FlexibleInstances,
3        MultiParamTypeClasses,
4        NoMonomorphismRestriction,
5        KindSignatures,
6        GeneralizedNewtypeDeriving,
7        ExistentialQuantification #-}
8
9  module Store where
10
11 import Base
12 import Call
13
14 import Control.Monad
15 import Control.Monad.State
16 import Control.Monad.Trans
17
18 import qualified Data.Map as Map
19
20
21
22 -- Implements store calls
23
24
25 data StoreCall store elem (m :: * -> *) r =
26      forall dat . StoreGet    (store elem dat)
27                                elem
28                               (dat -> r)
29
30   | forall dat . StorePut    (store elem dat)
31                                elem
32                                dat
33                               (store elem dat -> r)
34
35   | forall dat . StoreToFun (store elem dat)
36                               ((elem -> dat) -> r)
37
38   | forall dat . StoreInit    dat
39                               (store elem dat -> r)
```

```
40
41    | forall dat a n . StoreFoldM (a -> (elem , dat) -> m a)
42                                  a
43                                  (store elem dat)
44                                  (a -> r)
45
46
47 storeFoldM     f a store      = callT $ StoreFoldM     f a store
48 storeInit      dat            = callT $ StoreInit      dat
49 storeToFun     store          = callT $ StoreToFun     store
50
51 storeGet       store elem     = callT $ StoreGet store elem
52 storePut       store elem dat = callT $ StorePut store elem dat
53
54 storeGets      store elem f   = storeGet store elem >>= return . f
55 storeGetsM     store elem f   = storeGet store elem >>= f
56
57 storeModify    store elem f   =
58    storeGet store elem >>= (storePut store elem) . f
59
60 storeModifyM   store elem f   =
61    storeGet store elem >>= f >>= (storePut store elem)
62
63 storeToList :: (Monad m) =>
64       store elem dat -> SContT (StoreCall store elem) m [(elem, dat)]
65
66 storeToList store = storeFoldM (\l d -> return $ d : l) [] store
67
68
69 storeFromList :: (Monad m) =>
70          dat
71       -> [(elem, dat)]
72       -> SContT (StoreCall store elem) m (store elem dat)
73
74 storeFromList deflt []          = storeInit deflt
75 storeFromList deflt ((e,v) : l) = do store <- storeFromList deflt l
76                                      storePut store e v
77
78
79
80 -- Interpretation of Store as Data.Map
```

```
81
82
83 data StoreP elem dat = StoreP { storeP'init :: dat ,
84                                 storeP'map  :: Map.Map elem dat
85                               }
86  deriving (Eq,Show,Read,Ord)
87
88
89 runStoreP :: (Monad m, Ord elem) =>
90    SContT (StoreCall StoreP elem) m a -> m a
91
92 runStoreP = instanciateNoT interpretStoreP
93   where
94     interpretStoreP (StoreGet   (StoreP d m) e   k) =
95       k (maybe d id $ Map.lookup e m)
96
97     interpretStoreP (StoreToFun (StoreP d m)     k) =
98       k (\e -> maybe d id $ Map.lookup e m)
99
100    interpretStoreP (StorePut   (StoreP d m) e v k) =
101      k (StoreP d (Map.insert e v m))
102
103    interpretStoreP (StoreInit               d   k) =
104      k (StoreP d Map.empty)
105
106    interpretStoreP (StoreFoldM f a (StoreP d m) k) =
107      do r <- runStoreP $ foldM f a (Map.toList m)
108         k r
109
110
111 -- Implements a state
112
113 data StateCall state (m :: * -> *) r =
114       StateGet         (state -> r)
115    | StatePut state   (()    -> r)
116
117 stateGet   = callT   StateGet
118 statePut s = callT $ StatePut s
119
120 stateGets  f = stateGet >>= return . f
121 stateGetsM f = stateGet >>= f
```

```
122
123 stateModify  f = stateGet >>= statePut . f
124 stateModifyM f = stateGet >>= f >>= statePut
125
126
127
128 interpretStateT :: (Monad m) =>
129         StateCall s t (m a)
130    -> StateT s m a
131
132 interpretStateT (StateGet   k) = get   >>= liftStateT . k
133 interpretStateT (StatePut s k) = put s >>= liftStateT . k
134
135
136 runStateCall :: (Monad m) => s -> SContT (StateCall s) m a -> m a
137 runStateCall s m =
138     do r <- runStateT (instanciateT interpretStateT m) s
139        return $ fst r
140
141
142 -- Implements a global stack
143
144
145
146 data StackCall stack element (m :: * -> *) r =
147        StackEmpty                      (Bool          -> r)
148     |  StackPush   element             (()            -> r)
149     |  StackPop                        (Maybe element -> r)
150
151
152 stackEmpty   = callT $ StackEmpty
153 stackPush  e = callT $ StackPush e
154 stackPop     = callT $ StackPop
155
156
157
158 interpretStackStateT :: (Monad m) =>
159            StackCall t a t1 (m b) -> StateT [a] m b
160 interpretStackStateT (StackEmpty   k) =
161            gets   (not . null) >>= liftStateT . k
162 interpretStackStateT (StackPush  e k) =
```

```
163             modify (e :)          >>= liftStateT . k
164 interpretStackStateT (StackPop      k) =
165             do s <- get
166                r <- case s of
167                        []     -> return Nothing
168                        x : l -> put l >> return (Just x)
169                liftStateT $ k r
170
171
172 runStackStateT :: (Monad m) =>
173            SContT (StackCall t a) m a1 -> m a1
174 runStackStateT m =
175            do r <- runStateT (instanciateT interpretStackStateT m) []
176               return $ fst r
177
178
179 -- Implements Output
180
181 data OutputCall out (m :: * -> *) r = OutputCall out (() -> r)
182
183 output out = callT $ OutputCall out
184
185 interpretOutputP (OutputCall out k) = modify (out:) >>= liftStateT . k
186
187
188 runOutputP :: (Monad m) => SContT (OutputCall out) m a -> m (a , [out])
189
190 runOutputP m = runStateT (instanciateT interpretOutputP m) []
```

A.5 Nominal Terms

```
1  module Term where
2
3  import Naming
4  import Call
5
6  import Control.Monad
7  import Control.Monad.Identity
8
9  -- Atoms defined in Naming
10 type Cst = Integer
11 type Var = Integer
12
13 data Leaf = Atm Atm
14           | Cst Cst
15           | Var Var
16   deriving (Eq,Ord,Show,Read)
17
18
19 data Nominal set perm = Abs   Atm
20                       | NPerm (Perm set perm)
21   deriving (Eq,Ord,Show,Read)
22
23
24 -- | Nominal Terms Data Structure
25 data Term set perm =
26       Leaf     Leaf
27     | Pair     (Term    set perm) (Term set perm)
28          | Nominal (Nominal set perm) (Term set perm)
29   deriving (Eq,Ord,Show,Read)
30
31
32 -- ** Term Construction
33
34 atm :: Atm -> Term set perm
35 atm = Leaf . Atm
36
37 cst :: Cst -> Term set perm
38 cst = Leaf . Cst
39
```

```
40 var :: Var -> Term set perm
41 var = Leaf . Var
42
43 perm :: Perm set perm -> Term set perm -> Term set perm
44 perm PermId t = t
45 perm p      t = Nominal (NPerm p) t
46
47 swap :: Atm -> Atm -> Term set perm -> Term set perm
48 swap a b = perm (permSwapping a b)
49
50 nomAbs :: Atm -> Term set perm -> Term set perm
51 nomAbs a = Nominal $ Abs a
52
53 pair :: Term set perm -> Term set perm -> Term set perm
54 pair = Pair
55
56
57 termSize :: Term set perm -> Int
58 termSize (Leaf     _  ) = 1
59 termSize (Nominal _ u) = 2 + (termSize u)
60 termSize (Pair t1  t2) = 1 + (termSize t1) + (termSize t2)
61
62
63
64 substituteM :: (Monad m) =>
65              (Var -> m (Maybe (Term set perm)))
66           -> Term set perm
67           -> m (Term set perm)
68 substituteM f = substM'
69   where f' v = f v >>= return . (maybe (var v) id)
70
71         substM' (Leaf    (Var v)) = f' v
72         substM' (Nominal m t     ) = do x <- substM' t
73                                          return $ Nominal m x
74         substM' (Pair      s t   ) = do s' <- substM' s
75                                          t' <- substM' t
76                                          return $ Pair s' t'
77         substM'            t        = return t
78
79
80 -- | "substitute f t" substitute any variable v by (f v)
```

```
81 --        in t. f is a pure function
82 substitute ::   (Var -> Maybe (Term set perm))
83               -> Term set perm
84               -> Term set perm
85 substitute f = runIdentity . (substituteM $ Identity . f)
86
87
88 -- Lazy Term Multi Set
89
90 data LazyTermSet set perm =
91       LtsNil
92     | LtsSingle (Term         set perm)
93     | LtsNode   (LazyTermSet set perm) (LazyTermSet set perm)
94     | LtsRemap  (Perm         set perm) (LazyTermSet set perm)
95   deriving (Eq,Show,Read)
96
97
98 ltsMerge ::    LazyTermSet set perm
99             -> LazyTermSet set perm
100            -> LazyTermSet set perm
101 ltsMerge LtsNil t = t
102 ltsMerge t LtsNil = t
103 ltsMerge t1    t2 = LtsNode t1 t2
104
105
106
107 flattenLTS :: (Monad m) =>
108           LazyTermSet set perm
109      -> SContT (PermCall perm)
110         (SContT (SetCall set) m) [Term set perm]
111
112 flattenLTS lts = aux PermId [] lts
113   where
114     aux p l  LtsNil            = return l
115
116     aux p l (LtsNode   n1 n2) =
117        do l' <- aux p l n1
118           aux p l' n2
119
120     aux p l (LtsRemap  p2 n ) =
121        do p' <- permCompose MutateNone p p2
```

```
122             aux p' l n
123
124     aux p l (LtsSingle    t   ) =
125         return $ (perm p   t) : l
126
127
128 toLTS :: [Term set perm] -> LazyTermSet set perm
129 toLTS []        = LtsNil
130 toLTS (t : l) = LtsNode (LtsSingle t) (toLTS l)
131
132
133 -- Head Permutation
134
135 data Susp set perm element = Susp {
136        suspPerm :: Perm set perm ,
137        suspElem :: element
138     }
139   deriving (Eq,Ord,Show,Read)
140
141
142 termHeadPerm :: (Monad m) =>
143           Term set perm
144       ->  SContT (PermCall perm)
145           (SContT (SetCall set) m)
146             (Susp set perm (Term set perm))
147
148 termHeadPerm (Nominal (NPerm   p) t) =
149          do Susp p2 u <- termHeadPerm t
150             p3 <- permCompose MutateNone p p2
151             return $ Susp p3 u
152 termHeadPerm t =
153          return $ Susp PermId t
154
155
156
157 -- Head Reduction Function
158
159 permStepDown :: (Monad m) =>
160            Term set perm
161       ->  SContT (PermCall perm)
162           (SContT (SetCall set) m) (Term set perm)
```

```
163 permStepDown (Nominal (NPerm p)  (Leaf (Atm a))           ) =
164         do b <- permImage p a
165            return $ Leaf (Atm b)
166 permStepDown (Nominal (NPerm _)  (Leaf (Cst c))           ) =
167         return $ Leaf (Cst c)
168 permStepDown (Nominal (NPerm p1) (Nominal (NPerm p2) u)) =
169         do p3 <- permCompose MutateNone p1 p2
170            permStepDown (perm p3 u)
171 permStepDown (Nominal (NPerm p)  (Nominal (Abs a)     u)) =
172         do b <- permImage p a
173            return $ nomAbs b (perm p u)
174 permStepDown (Nominal (NPerm p)  (Pair t1 t2)             ) =
175         return $ Pair (perm p t1) (perm p t2)
176 permStepDown t                                              =
177         return t
178
179
180
181 -- Directions for Zipper on Nominal Terms
182
183 data TermDir = PairLeft | PairRight | NominalDown
184   deriving (Eq,Ord,Read,Show)
185
186 termDirs (Leaf _       ) = []
187 termDirs (Nominal n t) = [(NominalDown , (t  , (\x -> Nominal n x))) ]
188 termDirs (Pair t1  t2) = [(PairLeft    , (t1 , (\x -> Pair    x t2))) ,
189                           (PairRight   , (t2 , (\x -> Pair   t1 x)))
190                          ]
```

A.6 Sets and Permutation Layer

```haskell
{-# LANGUAGE RankNTypes,
             FlexibleInstances,
             MultiParamTypeClasses,
             NoMonomorphismRestriction,
             KindSignatures #-}


module Naming where

import Call

import qualified Data.Set as Set
import qualified Data.Map as Map

import Control.Monad

type Atm = Integer

data MutateSide = MutateLeft | MutateRight | MutateNone
 deriving (Eq,Ord,Read,Show)

msFlip :: MutateSide -> MutateSide
msFlip MutateNone  = MutateNone
msFlip MutateLeft  = MutateRight
msFlip MutateRight = MutateLeft



data SetCall  set (m :: * -> *) r =
    SetIsIn       set  Atm            (Bool  -> r)
  | SetIsSubset set  set              (Bool  -> r)
  | SetSize       set                 (Int   -> r)
  | SetSet        Bool Atm  Bool set  (set   -> r)
      -- True : Mutate , False : Do NOT Mutate
  | SetEmptyNative                    (set   -> r)
  | SetToList     set                 ([Atm] -> r)
  | SetUnion      MutateSide set set  (set   -> r)
  | SetInter      MutateSide set set  (set   -> r)

```

```
40
41 data Set set = SetEmpty
42              | SetNative set
43   deriving (Eq,Ord,Show,Read)
44
45 setIsIn :: (Monad m) =>   Set set
46                        -> Atm
47                        -> SContT (SetCall set) m Bool
48
49 setIsIn  SetEmpty        _ = return False
50 setIsIn  (SetNative set) a = callT $ SetIsIn set a
51
52
53
54 setSize :: (Monad m) => Set t -> SContT (SetCall t) m Int
55
56 setSize  SetEmpty        = return 0
57 setSize (SetNative set) = callT $ SetSize set
58
59 setIsSubSet :: (Monad m) =>   Set t
60                            -> Set t
61                            -> SContT (SetCall t) m Bool
62
63 setIsSubSet  SetEmpty              _             =
64          return True
65
66 setIsSubSet (SetNative set )  SetEmpty           =
67          do n <- callT $ SetSize set
68             return (n == 0)
69
70 setIsSubSet (SetNative set1) (SetNative set2) =
71          callT $ SetIsSubset set1 set2
72
73
74
75
76 setNativeEmpty :: (Monad m) =>
77         SContT (SetCall set) m (Set set)
78
79 setNativeEmpty = do s <- callT SetEmptyNative
80                     return $ SetNative s
```

```
81
82 setSet :: (Monad m) =>     Bool
83                         -> Atm
84                         -> Bool
85                         -> Set set
86                         -> SContT (SetCall set) m (Set set)
87
88 setSet mutate a v  SetEmpty        =
89          do s <- setNativeEmpty
90             setSet mutate a v s
91
92 setSet mutate a v (SetNative set) =
93          do s <- callT $ SetSet mutate a v set
94             return $ SetNative s
95
96
97 setToList :: (Monad m) =>     Set set
98                            -> SContT (SetCall set) m [Atm]
99
100 setToList  SetEmpty        = return []
101 setToList (SetNative set) = callT $ SetToList set
102
103 setFromList :: (Monad m) =>     [Atm]
104                              -> SContT (SetCall set) m (Set set)
105
106 setFromList    []      = return SetEmpty
107 setFromList (a : l) = do s <- setFromList l
108                          setSet True a True s
109
110 setUnshare :: (Monad m) =>
111        Set set -> SContT (SetCall set) m (Set set)
112 setUnshare set = setToList set >>= setFromList
113
114
115 setSwap :: (Monad m) =>      Bool
116                          -> Set set
117                          -> Atm
118                          -> Atm
119                          -> SContT (SetCall set) m (Set set)
120
121 setSwap mutate set a b = do va <- setIsIn set a
```

```
122                               vb <- setIsIn set b
123                               s1 <- setSet mutate a vb set
124                               setSet mutate b va s1
125
126 setUnion :: (Monad m) =>     MutateSide
127                            -> Set set
128                            -> Set set
129                            -> SContT (SetCall set) m (Set set)
130
131 setUnion _  SetEmpty                     set2  = return set2
132
133 setUnion _             set1   SetEmpty         = return set1
134
135 setUnion ms (SetNative set1) (SetNative set2) =
136         do s <- callT $ SetUnion ms set1 set2
137            return $ SetNative s
138
139
140 setInter :: (Monad m) =>      MutateSide
141                             -> Set set
142                             -> Set set
143                             -> SContT (SetCall set) m (Set set)
144
145 setInter _  SetEmpty                     set2  = return set2
146
147 setInter _             set1   SetEmpty         = return set1
148
149 setInter ms (SetNative set1) (SetNative set2) =
150         do s <- callT $ SetInter ms set1 set2
151            return $ SetNative s
152
153 {-
154   Permutations
155 -}
156
157
158 data PermCall perm (m :: * -> *) r =
159           PermImage                 perm Atm      (Atm  -> r)
160         | PermSwapRight Bool        perm Atm  Atm (perm -> r)
161             -- True : Mutate, False : Do Not Mutate
162         | PermCompose   MutateSide perm perm      (perm -> r)
```

```
163         | PermSupp                perm          ([Atm] -> r)
164         | PermIdNative                          (perm  -> r)
165         | PermUnshare             perm          (perm  -> r)
166
167
168
169
170
171
172 data Perm set perm = PermId
173                    | PermSwap   Atm Atm
174                    | PermNative perm perm (Set set)
175   deriving (Eq,Ord,Show,Read)
176
177
178 permSupp :: (Monad m) =>
179        Perm set t -> SContT (SetCall set) m (Set set)
180
181 permSupp  PermId               = return SetEmpty
182 permSupp (PermSwap a b)        = if a == b
183                                   then return SetEmpty
184                                   else setFromList [a,b]
185 permSupp (PermNative _ _ s) = return s
186
187
188 permInverse :: Perm set perm -> Perm set perm
189 permInverse (PermNative p i s) = PermNative i p s
190 permInverse p                  = p
191
192 permImage :: (Monad m) =>   Perm set perm
193                          -> Atm
194                          -> SContT (PermCall perm) m Atm
195
196 permImage   PermId           c = return c
197
198 permImage  (PermSwap a b)    c = if c == a
199                                   then return b
200                                   else if c == b
201                                         then return a
202                                         else return c
203
```

```
204 permImage   (PermNative p _ _) c = callT $ PermImage p c
205
206
207 permImageInv :: (Monad m) =>      Perm set perm
208                                -> Atm
209                                -> SContT (PermCall perm) m Atm
210 permImageInv = permImage . permInverse
211
212
213
214 permSwapping :: Atm -> Atm -> Perm set perm
215 permSwapping a b = if a == b
216                    then PermId
217                    else PermSwap a b
218
219 permCompose :: (Monad m) =>
220          MutateSide
221       -> Perm set perm
222       -> Perm set perm
223       -> SContT    (PermCall perm)
224            (SContT (SetCall set) m) (Perm set perm)
225
226 permCompose _   PermId p = return p
227
228 permCompose _   p PermId = return p
229
230 permCompose ms   (PermSwap a b) p =
231     do ia <- permImageInv p a
232        ib <- permImageInv p b
233        permSwapRight (ms == MutateRight) p ia ib
234
235 permCompose ms   p (PermSwap a b) =
236     permSwapRight (ms == MutateLeft) p a b
237
238 permCompose ms (PermNative p1 i1 _) (PermNative p2 i2 _) =
239   do p3 <- callT  $ PermCompose          ms  p1 p2
240      i3 <- callT  $ PermCompose (msFlip ms) i2 i1
241      l  <- callT  $ PermSupp p3
242      s3 <- clv 'p1 $ setFromList l
243      return $ PermNative p3 i3 s3
244
```

```
245
246 permFromSwappings :: (Monad m) =>
247         [(Atm, Atm)]
248     -> SContT       (PermCall a)
249            (SContT (SetCall set) m) (Perm set a)
250
251 permFromSwappings []       = return   PermId
252
253 permFromSwappings [(a,b)] = return $ permSwapping a b
254
255 permFromSwappings l        = aux PermId l
256   where
257     aux p [] = return p
258     aux p ((a,b) : l) = do p2 <- permSwapRight True p a b
259                            aux p2 l
260
261
262
263 permSwapRight :: (Monad m) =>
264          Bool
265     -> Perm set a
266     -> Atm
267     -> Atm
268     -> SContT      (PermCall a)
269          (SContT (SetCall set) m) (Perm set a)
270
271 permSwapRight mutate perm a b =
272  if a == b
273   then return perm
274   else case perm of
275          PermId              ->
276              return $ PermSwap a b
277
278          PermSwap c d        ->
279              if sameSwap (a,b) (c,d)
280                then return PermId
281                else do p <- permFromSwappingsNative [(c,d),(a,b)]
282                        i <- permFromSwappingsNative [(a,b),(c,d)]
283                        s <- partialSupp p [a,b,c,d]
284                        return $ PermNative p i s
285
```

```
286           PermNative p i s ->
287               do pa <- callT $ PermImage p a
288                  pb <- callT $ PermImage p b
289                  p2 <- callT $ PermSwapRight mutate p a  b
290                  i2 <- callT $ PermSwapRight mutate i pa pb
291                  s' <- clv'p1 $ setSet mutate a (a /= pb) s
292                           -- p b = p (a b) a = p2 a
293                  s2 <- clv'p1 $ setSet mutate b (b /= pa) s
294                           -- p a = p (a b) b = p2 b
295                  return $ PermNative p2 i2 s2
296
297   where
298     sameSwap :: (Atm, Atm) -> (Atm, Atm) -> Bool
299     sameSwap (a,b) (c,d) =     ((a == c) && (b == d))
300                             || ((a == d) && (b == c))
301
302     permFromSwappingsNative l =
303         do pid <- callT $ PermIdNative
304            auxFSN pid l
305
306     auxFSN p []           = return p
307     auxFSN p ((a,b) : l) =
308        do p2 <- callT $ PermSwapRight True p a b
309           auxFSN p2 l
310
311     partialSupp p l =
312          do sl <- mapM (\x -> do px <- callT $ PermImage p x
313                                  return (x , (x /= px))
314                        )
315                        l
316             foldM (\set (a,v) -> clv'p1 $ setSet True a v set)
317                   SetEmpty
318                   sl
319
320
321
322
323 permDS :: (Monad m) =>
324           Perm set perm
325        -> Perm set perm
326        -> SContT     (PermCall perm)
```

```
327              (SContT (SetCall set) m) (Set set)
328
329 permDS p1 p2 = do p3 <- permCompose MutateNone
330                                     (permInverse p1)
331                                     p2
332                    clv'p1 $ permSupp p3
333
334
335 setPermute :: (Monad m) =>
336           Bool
337       -> Perm set perm
338       -> Set set
339       -> SContT      (PermCall perm)
340              (SContT (SetCall set) m) (Set set)
341
342
343 setPermute mutate  PermId          set = return set
344
345 setPermute mutate (PermSwap a b) set =
346       clv'p1 $ setSwap mutate set a b
347
348 setPermute mutate  p              set =
349       do l  <- clv'p1 $ setToList set
350          l' <- mapM (permImage p) l
351          clv'p1 $ setFromList l'
352
353
354 permUnshare :: (Monad m) =>
355         Perm set perm
356     -> SContT (PermCall perm)
357               (SContT (SetCall set) m)
358               (Perm set perm)
359
360 permUnshare p@PermId               = return p
361
362 permUnshare p@(PermSwap _ _)       = return p
363
364 permUnshare    (PermNative p i s) = do p' <- callT $ PermUnshare p
365                                         i' <- callT $ PermUnshare i
366                                         s' <- clv'p1 $ setUnshare s
367                                         return $ PermNative p' i' s'
```

```
368
369
370
371
372 {-
373     Interpretation of Sets and Perms as Data.Map
374 -}
375
376 interpretSetPersistent :: SetCall (Set.Set Atm) m r -> r
377
378 interpretSetPersistent (SetIsIn        set a   k) =
379        k (Set.member     a set)
380
381 interpretSetPersistent (SetSize        set     k) =
382        k (Set.size          set)
383
384 interpretSetPersistent (SetIsSubset    s1 s2   k) =
385        k (Set.isSubsetOf s1 s2)
386
387 interpretSetPersistent (SetSet         _ a v s k) =
388        k (if v
389             then Set.insert a s
390             else Set.delete a s
391          )
392 interpretSetPersistent (SetEmptyNative         k) =
393        k  Set.empty
394
395 interpretSetPersistent (SetToList      set     k) =
396        k (Set.toList set)
397
398 interpretSetPersistent (SetUnion       _ s1 s2 k) =
399        k (Set.union s1 s2)
400 interpretSetPersistent (SetInter       _ s1 s2 k) =
401        k (Set.intersection s1 s2)
402
403
404
405 runSetPersistent :: (Monad m) =>
406          SContT (SetCall (Set.Set Atm)) m a -> m a
407 runSetPersistent = instanciateNoT interpretSetPersistent
408
```

```
409
410 mapImage :: (Ord a) => Map.Map a a -> a -> a
411 mapImage    p    a = maybe a id $ Map.lookup a p
412
413 mapCompose :: (Ord a) =>     Map.Map a a
414                           -> Map.Map a a
415                           -> Map.Map a a
416
417 mapCompose p1 p2 = Map.foldWithKey f p1 p2
418   where f atm image p = Map.insert atm (mapImage p1 image) p
419
420
421 mapSupp :: (Ord b) => Map.Map b b -> [b]
422 mapSupp     p      = foldl (\l a -> if a == mapImage p a
423                                     then l
424                                     else a : l
425                          ) [] (Map.keys p)
426
427
428
429 interpretPermPersistent :: PermCall (Map.Map Atm Atm) m r -> r
430
431 interpretPermPersistent (PermImage        p a   k) =
432       k (mapImage p a)
433
434 interpretPermPersistent (PermSwapRight _ p a b k) =
435       k (let pa = mapImage p a
436              pb = mapImage p b
437          in Map.insert a pb (Map.insert b pa p)
438         )
439
440 interpretPermPersistent (PermCompose _   p1 p2 k) =
441       k $ mapCompose p1 p2
442
443 interpretPermPersistent (PermSupp         p     k) =
444       k $ mapSupp p
445
446 interpretPermPersistent (PermIdNative           k) =
447       k   Map.empty
448
449 interpretPermPersistent (PermUnshare      p     k) =
```

```
450     k    p
451
452
453
454
455 runPermPersistent :: (Monad m) =>
456     SContT (PermCall (Map.Map Atm Atm)) m a -> m a
457
458 runPermPersistent = instanciateNoT interpretPermPersistent
459
460
461
462
463 runNamingPersistent :: (Monad m) =>
464         SContT (PermCall (Map.Map Atm Atm))
465        (SContT (SetCall  (Set.Set Atm)) m) a
466     -> m a
467 runNamingPersistent = runSetPersistent . runPermPersistent
```

A.7 Nominal Union-Find Algorithm

```
1  {-# LANGUAGE RankNTypes,
2               FlexibleInstances,
3               MultiParamTypeClasses,
4               NoMonomorphismRestriction,
5               KindSignatures,
6               GeneralizedNewtypeDeriving #-}
7
8  module UnionFind where
9
10 import Call
11 import Naming
12 import Term
13 import Store
14
15
16 import Control.Monad
17 import Control.Monad.Trans
18
19
20 -- Abstraction of Nominal Datas
21
22 data NominalDataCall set perm dat (m :: * -> *) r =
23      NominalDataFresh   (Set set)       dat (dat -> r)
24   |  NominalDataPermute (Perm set perm) dat (dat -> r)
25   |  NominalDataMerge    dat dat            (dat -> r)
26
27 nominalDataFresh   set  dat  = callT $ NominalDataFresh   set  dat
28 nominalDataPermute perm dat  = callT $ NominalDataPermute perm dat
29 nominalDataMerge   dat1 dat2 = callT $ NominalDataMerge   dat1 dat2
30
31
32
33 data Ranked dat = Ranked { rank'rank :: Integer  ,
34                            rank'data :: dat
35                          }
36  deriving (Eq,Show,Read)
37
38
39 data UFCell set perm elem dat = Indirect (Susp set perm elem)
```

```
40                                  | Direct     dat
41    deriving (Eq,Show,Read)
42
43
44 unRankUFCell (Direct   (Ranked r d)) = Direct d
45 unRankUFCell (Indirect             s) = Indirect s
46
47 -- Calls for the arr_{uf} store
48
49 type UFStore set perm store elem dat m =
50        SContT (StateCall (store elem
51                     (UFCell set perm elem (Ranked dat))))
52       (SContT (StoreCall store elem) m)
53
54
55 ufDataInit :: dat -> UFCell set perm elem (Ranked dat)
56 ufDataInit dat = Direct (Ranked 0 dat)
57
58
59
60 ufStoreInit :: (Monad m) =>
61          dat
62       -> UFStore set perm store elem dat m ()
63
64 ufStoreInit dat = (clv 'p1 $ storeInit $ ufDataInit dat) >>= statePut
65
66
67 ufStoreGet :: (Monad m) =>
68          elem
69       -> UFStore set perm store elem dat m
70                (UFCell set perm elem (Ranked dat))
71
72 ufStoreGet        elem       =
73     stateGetsM (\store -> clv 'p1 $ storeGet store elem)
74
75 ufStorePut :: (Monad m) =>
76          elem
77       -> UFCell set perm elem (Ranked dat)
78       -> UFStore set perm store elem dat m ()
79
80 ufStorePut        elem dat =
```

```
81     stateModifyM (\store -> clv 'p1 $ storePut store elem dat)
82
83
84
85
86 ufStoreGets :: (Monad m) =>
87         elem
88     -> (UFCell set perm elem (Ranked dat) -> a)
89     -> UFStore set perm store elem dat m a
90
91 ufStoreGets       elem f    = ufStoreGet elem >>= return . f
92
93
94
95
96 ufStoreGetsM :: (Monad m) =>
97         elem
98     -> (UFCell set perm elem (Ranked dat)
99                -> UFStore set perm store elem dat m a)
100    -> UFStore set perm store elem dat m a
101
102 ufStoreGetsM      elem f    = ufStoreGet elem >>= f
103
104
105
106
107 ufStoreModify :: (Monad m) =>
108         elem
109     -> (UFCell set perm elem (Ranked dat)
110              -> UFCell set perm elem (Ranked dat))
111     -> UFStore set perm store elem dat m ()
112
113 ufStoreModify     elem f    =
114     ufStoreGet elem >>= (ufStorePut elem) . f
115
116
117
118 ufStoreModifyM :: (Monad m) =>
119         elem
120     -> (UFCell set perm elem (Ranked dat) ->
121              UFStore set perm store elem dat m
```

```
122                    (UFCell set perm elem (Ranked dat)))
123        -> UFStore set perm store elem dat m ()
124
125 ufStoreModifyM  elem f   =
126      ufStoreGet elem >>= f >>= (ufStorePut elem)
127
128
129
130 ufStoreFoldM :: (Monad m) =>
131          (a -> (elem, UFCell set perm elem dat)
132        -> SContT (StoreCall store elem) m a)
133        -> a
134        -> UFStore set perm store elem dat m a
135
136 ufStoreFoldM f a =
137     do store <- stateGet
138        let f' a (e,v) = f a (e, unRankUFCell v)
139        clv'p1 $ storeFoldM f' a store
140
141
142
143 findRanked :: (Monad m) =>
144            Susp set perm elem
145        -> UFStore set perm store elem dat
146          (SContT (PermCall perm)
147          (SContT (SetCall  set ) m))
148             (Susp set perm (elem, Ranked dat))
149
150
151 findRanked (Susp perm var) =
152   do vc <- ufStoreGet var
153       case vc of
154        Direct d    -> return $ Susp perm (var, d)
155
156        Indirect s  -> do Susp p (r,d) <- findRanked s
157                          ufStorePut var $ Indirect (Susp p r)
158                          p2 <- clv'p2 $ permCompose MutateNone
159                                                     perm
160                                                     p
161                          return $ Susp p2 (r,d)
162
```

```
163
164
165
166 find :: (Monad m) =>
167          Susp set perm elem
168      -> UFStore set perm store elem dat
169         (SContT (PermCall perm)
170         (SContT (SetCall  set) m))
171            (Susp set perm (elem, dat))
172
173 find s = do Susp p (v , (Ranked r d)) <- findRanked s
174              return $ Susp p (v,d)
175
176
177 ufDataModify :: (Monad m) =>
178      (dat
179       -> SContT (NominalDataCall set perm dat)
180          (UFStore set perm store elem dat
181          (SContT (PermCall perm)
182          (SContT (SetCall set) m)))
183             dat)
184      -> Susp set perm elem
185      -> SContT (NominalDataCall set perm dat)
186          (UFStore set perm store elem dat
187          (SContT (PermCall perm)
188          (SContT (SetCall set) m)))
189            ()
190
191 ufDataModify f s =
192    do Susp p (r, Ranked rk d) <- clv'p1 $ findRanked s
193       dp                      <- nominalDataPermute p d
194       d2                      <- f d
195       d3                      <- nominalDataPermute (permInverse p) d2
196       clv'p1 $ ufStorePut r $ Direct (Ranked rk d3)
197
198
199 {-
200    pr1 . r1 = pr2 . r2
201
202    A1 # r1 = n1  . r2
203       => p3^-1(A1) # r2 = p3^-1 n1 , p3 = pr1^-1 . pr2
```

```
204 -}
205
206
207 union   :: (Monad m, Eq elem) =>
208           Susp set perm elem
209        -> Susp set perm elem
210        -> SContT (NominalDataCall set perm dat)
211           (UFStore set perm store elem dat
212           (SContT (PermCall perm)
213           (SContT (SetCall set) m)))
214             ()
215
216 union s1 s2 =
217   do Susp p1 (r1 , Ranked rk1 d1) <- clv 'p1 $ findRanked s1
218      Susp p2 (r2 , Ranked rk2 d2) <- clv 'p1 $ findRanked s2
219      if r1 == r2
220       then do ds <- clv 'p3 $ permDS p1 p2
221               d3 <- nominalDataFresh ds d1
222               clv 'p1 $ ufStorePut r1 (Direct $ Ranked rk1 d3)
223       else case compare rk1 rk2 of
224              EQ -> do clv 'p1 $ ufStorePut r2
225                                  (Direct $ Ranked (rk2 + 1) d2)
226                       union (Susp p1 r1) (Susp p2 r2)
227              GT -> union (Susp p2 r2) (Susp p1 r1)
228              LT -> do p3 <- clv 'p3 $ permCompose MutateNone
229                                               (permInverse p1)
230                                                p2
231                       clv 'p1 $ ufStorePut r1 $ Indirect (Susp p3 r2)
232                       ufDataModify (\d -> nominalDataMerge d d1)
233                                    (Susp PermId r1)
234
235
236
237 unions :: (Monad m, Eq elem) => [Susp set perm elem]
238        -> SContT
239             (NominalDataCall set perm dat)
240             (UFStore set perm store elem dat
241             (SContT (PermCall perm)
242             (SContT (SetCall set) m)))
243                  (Susp set perm (elem, dat))
244
```

```
245 unions []               = error "Empty list of union finds"
246 unions [x]              = clv'p1 $ find x
247 unions (x : (y : l)) = do union x y
248                             unions (y : l)
249
250
251
252 runUFStoreP :: (Monad m, Ord elem) =>
253         dat
254     -> SContT (StateCall (StoreP elem
255                (UFCell set perm elem1 (Ranked dat))))
256        (SContT (StoreCall StoreP elem) m)   a
257     -> m a
258
259 runUFStoreP dat m =
260    runStoreP $ do store <- storeInit $ ufDataInit dat
261                   runStateCall store m
```

A.8 Unification

```
1  {-# LANGUAGE RankNTypes,
2               FlexibleInstances ,
3               MultiParamTypeClasses ,
4               NoMonomorphismRestriction ,
5               KindSignatures ,
6               GeneralizedNewtypeDeriving ,
7               ExistentialQuantification #-}
8
9  module Unification
10  where
11
12 import Call
13 import Naming
14 import Term
15 import UnionFind
16 import Store
17 import Error
18
19 import Control.Monad
20 import Control.Monad.Identity
21
22 import qualified Data.Set as Set
23 import qualified Data.Map as Map
24
25
26 -- SPECIALIZED UNION FIND TO UNIFICATION
27
28
29 data NominalDataUnif set perm = NominalDataUnif
30      {  ndu'occurs :: Int  ,
31         ndu'set    :: Set set  ,
32         ndu'terms  :: LazyTermSet set perm
33      }
34  deriving (Eq,Show,Read)
35
36
37 type UFStoreUnif set perm store m =
38      UFStore set perm store Var (NominalDataUnif set perm) m
39
```

```
40
41 nominalDataUnifFresh :: (Monad m) =>
42           Set set
43       -> NominalDataUnif set perm
44       -> SContT (SetCall set) m (NominalDataUnif set perm)
45
46 nominalDataUnifFresh s (NominalDataUnif occurs set terms) =
47    do s2 <- setUnion MutateNone s set
48        return $ NominalDataUnif occurs s2 terms
49
50
51 nominalDataUnifPermute :: (Monad m) =>
52           Perm set perm
53       -> NominalDataUnif set perm
54       -> SContT     (PermCall perm)
55             (SContT (SetCall set) m)
56                   (NominalDataUnif set perm)
57
58 nominalDataUnifPermute perm (NominalDataUnif occurs set terms) =
59    do s2 <- setPermute False perm set
60        return $ NominalDataUnif occurs s2 (LtsRemap perm terms)
61
62
63 nominalDataUnifMerge :: (Monad m) =>
64           NominalDataUnif set perm
65       -> NominalDataUnif set perm
66       -> SContT (SetCall set) m (NominalDataUnif set perm)
67
68 nominalDataUnifMerge  (NominalDataUnif occurs1 set1 terms1)
69                        (NominalDataUnif occurs2 set2 terms2) =
70    do s3 <- setUnion MutateNone set1 set2
71        return $ NominalDataUnif (occurs1 + occurs2)
72                                 s3
73                                 (ltsMerge terms1 terms2)
74
75
76
77 interpretNominalDataUnif :: (Monad m1, Monad m) =>
78             (SContT (PermCall perm)
79             (SContT (SetCall t1) m1) (NominalDataUnif t1 perm)
80         -> m (NominalDataUnif t1 perm))
```

```
81         -> NominalDataCall t1 perm (NominalDataUnif t1 perm) t (m b)
82         -> m b
83
84 interpretNominalDataUnif lv (NominalDataFresh    set   dat k) =
85    (lv $ clv'p1 $ nominalDataUnifFresh    set   dat  ) >>= k
86
87 interpretNominalDataUnif lv (NominalDataPermute perm dat k) =
88    (lv $             nominalDataUnifPermute perm dat  ) >>= k
89
90 interpretNominalDataUnif lv (NominalDataMerge dat1 dat2  k) =
91    (lv $ clv'p1 $ nominalDataUnifMerge    dat1 dat2) >>= k
92
93
94 findUnif = find
95
96
97
98 ufDataModifyUnif :: (Monad m1) =>
99      (NominalDataUnif set perm  ->
100             SContT    (NominalDataCall set
101                                          perm
102                                         (NominalDataUnif set perm)
103                        )
104               (UFStoreUnif set perm store
105               (SContT (PermCall perm)
106               (SContT (SetCall set) m1)))
107               (NominalDataUnif set perm)
108      )
109      -> Susp set perm Var
110      -> UFStoreUnif set perm store
111         (SContT (PermCall perm)
112         (SContT (SetCall set) m1))
113           ()
114
115 ufDataModifyUnif f s =
116      instanciateNoT (interpretNominalDataUnif clv'p2)
117                     (ufDataModify f s)
118
119
120 unionUnif :: (Monad m) =>
121         Susp set perm Var
```

```
122       -> Susp set perm Var
123       -> UFStoreUnif set perm store
124            (SContT (PermCall perm)
125            (SContT (SetCall set) m)) ()
126
127 unionUnif s1 s2 =
128        instanciateNoT (interpretNominalDataUnif clv 'p2)
129                       (union s1 s2)
130
131
132 nominalDataUnifNeutral :: NominalDataUnif set perm
133 nominalDataUnifNeutral = NominalDataUnif 0 SetEmpty LtsNil
134
135
136 unionsUnif :: (Monad m) =>
137           [Susp set perm Var]
138       -> UFStoreUnif set perm store
139          (SContT (PermCall perm)
140          (SContT (SetCall set) m))
141               (Susp set perm (Var, NominalDataUnif set perm))
142
143 unionsUnif l     = instanciateNoT (interpretNominalDataUnif clv 'p2)
144                                   (unions l)
145
146 nominalDataUnifOccurAdd :: (Monad m) =>
147            Var
148       ->   Int
149       ->   UFStoreUnif set perm store
150            (SContT (PermCall perm)
151            (SContT (SetCall set) m)) ()
152
153 nominalDataUnifOccurAdd v n = ufDataModifyUnif f (Susp PermId v)
154   where
155     f d = nominalDataMerge d (NominalDataUnif n SetEmpty LtsNil)
156
157
158 ufStoreInitUnif :: (Monad m) =>
159       UFStoreUnif set perm store m ()
160
161 ufStoreInitUnif = ufStoreInit nominalDataUnifNeutral
162
```

```
163
164 runUFStoreUnifP :: (Monad m, Ord elem) =>
165       SContT (StateCall (StoreP elem
166               (UFCell set1 perm1 elem1
167                    (Ranked (NominalDataUnif set perm)))))
168       (SContT (StoreCall StoreP elem) m)
169         a
170    -> m a
171
172 runUFStoreUnifP = runUFStoreP nominalDataUnifNeutral
173
174 -- THE UNIFICATION ALGORITHM
175
176
177 data Equation set perm = Equation
178       { eqSet    :: Set set ,
179         eqTerms :: [Term set perm]
180       }
181  deriving (Eq,Show,Ord,Read)
182
183
184 data OutputSolution set perm =
185       OutputSubst Var (Term set perm)
186     | OutputFresh Var (Set set)
187  deriving (Show,Read,Eq,Ord)
188
189
190 iterM :: (Monad m) => (t -> m a) -> [t] -> m ()
191 iterM f []       = return ()
192 iterM f (x : l) = do f x
193                      iterM f l
194
195
196 splitEq :: (Monad m) =>
197           [Term set perm]
198       ->  SContT (PermCall perm)
199           (SContT (SetCall set) m)
200                 ([Susp set perm Var], [Term set perm])
201
202 splitEq []       = return ([],[])
203 splitEq (t : l) = do Susp p u <- termHeadPerm t
```

```
204                        (vs,ts)  <- splitEq l
205                        case u of
206                         Leaf (Var v) -> return (((Susp p v) : vs) , ts)
207                         _            -> return (vs , (perm p u) : ts)
208
209
210 decompose :: (Monad m) =>
211          Equation set perm
212      -> SContT        (StackCall stack (Equation set perm))
213           (SContT      (OutputCall (OutputSolution set perm))
214           (UFStoreUnif set perm store
215           (SContT      (PermCall perm)
216           (SContT      (SetCall set)
217           (SContT      (ErrorCall [Char]) m))))) ()
218
219 decompose (Equation fs []     ) = return ()
220
221 decompose (Equation fs (t : q)) =
222   do u <- clv 'p4 $ permStepDown t
223      l <- mapM (clv 'p4 . permStepDown) q
224      case u of
225        Leaf (Var _)          ->
226            clv 'p6 $ raiseError "Error: head variables"
227
228        Leaf (Atm a)          ->
229            do val <- clv 'p5 $ setIsIn fs a
230               when val (clv 'p6 $ raiseError "Error: Freshness Cst")
231               iterM (clv 'p6 . (checkLeaf $ Atm a)) l
232
233        Leaf (Cst c)          ->
234            iterM (clv 'p6 . (checkLeaf $ Cst c)) l
235
236        Nominal (NPerm  _) _ ->
237             clv 'p6 $ raiseError "permStepDown did Wrong !"
238
239        Nominal (Abs     a) _ ->
240             do (atms,terms) <- clv 'p6 $ allAbs a (u : l)
241                let f set a = clv 'p5 $ setSet False a True set
242                fs1 <- foldM f fs atms
243                fs2 <- clv 'p5 $ setSet False a False fs1
244                stackPush (Equation fs2 terms)
```

```
245
246         Pair t1 t2             ->
247               do (l1 , l2) <- clv 'p6 $ allPair (u : l)
248                  stackPush (Equation fs l1)
249                  stackPush (Equation fs l2)
250
251  where
252    checkLeaf :: (Monad m) =>      Leaf
253                                -> Term set perm
254                                -> SContT (ErrorCall String) m ()
255
256    checkLeaf l1 (Leaf l2) = when (l1 /= l2)
257                                (raiseError "Not the same leaf!")
258    checkLeaf _ _           =      raiseError "Not the same leaf!"
259
260    allPair :: (Monad m) =>      [Term t t1]
261                              -> SContT (ErrorCall String)
262                                        m
263                                   ([Term t t1], [Term t t1])
264
265    allPair = foldM (\(ll,lr) t ->
266                          case t of
267                            Pair t1 t2 ->
268                               return ((t1 : ll) , (t2 : lr))
269
270                            _          ->
271                               raiseError "Not a Pair"
272                    ) ([],[])
273
274    allAbs :: (Monad m) =>      Atm
275                            -> [Term set perm]
276                            -> SContT (ErrorCall String)
277                               m
278                                 ([Atm], [Term set perm])
279
280    allAbs a l = mapM (checkAbs a) l >>= return . unzip
281
282    checkAbs :: (Monad m) =>      Atm
283                              -> Term t t1
284                              -> SContT (ErrorCall String)
285                                   m (Atm, Term t t1)
```

```
286
287     checkAbs a (Nominal (Abs b) u) =
288           return (b , (swap a b u))
289     checkAbs a _                    =
290           raiseError "Not␣un␣abstraction"
291
292
293
294
295 checkLastOccurence :: (Monad m) =>
296          Var
297      -> SContT   (StackCall stack (Equation set perm))
298           (SContT (OutputCall (OutputSolution set perm))
299           (UFStoreUnif set perm store
300           (SContT (PermCall perm)
301           (SContT (SetCall set) m)))) ()
302
303 checkLastOccurence v =
304   do Susp p (r , d) <- clv'p2 $ findUnif $ Susp PermId v
305      when (ndu'occurs d == 1)
306        (do terms <- clv'p4 $ flattenLTS $ ndu'terms d
307            case terms of
308              []     -> clv'p1 $ output (OutputFresh v (ndu'set d))
309
310              t : l -> do stackPush (Equation (ndu'set d) terms)
311                          clv'p1 $ output (OutputSubst v t)
312
313            let f _ = return nominalDataUnifNeutral
314            clv'p2 $ ufDataModifyUnif f (Susp PermId v)
315        )
316
317
318
319 storeEquation :: (Monad m) =>
320          Susp set perm Var
321      -> Equation set perm
322      -> SContT   (StackCall stack (Equation set perm))
323           (SContT (OutputCall (OutputSolution set perm))
324           (UFStoreUnif set perm store
325           (SContT (PermCall perm)
326           (SContT (SetCall set) m)))) ()
```

```
327
328 storeEquation (Susp p v) (Equation fs terms) =
329   do let i      =  permInverse p
330      ifs        <- clv'p4 $ setPermute False i fs
331      let ddiff =  NominalDataUnif 0 ifs (LtsRemap i (toLTS terms))
332      let f d    =  nominalDataMerge d ddiff
333      clv'p2 $ ufDataModifyUnif f (Susp p v)
334      checkLastOccurence v
335
336
337 occurCheck :: (Monad m) =>
338      UFStore
339           set
340           perm
341           store
342           elem
343           (NominalDataUnif set1 perm1)
344           (SContT (PermCall perm)
345           (SContT (SetCall  set )
346           (SContT (ErrorCall [Char]) m)))
347           ()
348
349 occurCheck = ufStoreFoldM f ()
350   where
351     f () (_ , Indirect _) =
352          return ()
353
354     f () (_ , Direct d  ) =
355          when (ndu'occurs d > 0)
356               (clv'p3 $ raiseError "OccurCheck␣Failed" )
357
358
359 solveLoop :: (Monad m) =>
360      SContT (StackCall stack (Equation set perm))
361     (SContT (OutputCall (OutputSolution set perm))
362     (UFStoreUnif set perm store
363     (SContT (PermCall perm)
364     (SContT (SetCall set)
365     (SContT (ErrorCall [Char]) m))))) ()
366
367 solveLoop =
```

```
368  do meq <- stackPop
369     case meq of
370      Nothing ->  clv'p2 $ occurCheck
371      Just eq ->  let Equation fs lt = eq in
372                  do  (susps , terms) <- clv'p4 $ splitEq lt
373                      if null susps
374                        then decompose     (Equation fs terms)
375                        else do Susp p (v,_) <- clv'p2 $ unionsUnif susps
376                                clv'p2 $ nominalDataUnifOccurAdd
377                                                    v
378                                                    (1 - (length susps))
379                                storeEquation (Susp p v)
380                                              (Equation fs terms)
381                      solveLoop
382
383
384
385 addOccurences :: (Monad m) =>
386          Term set perm
387      -> UFStoreUnif set perm store
388         (SContT (PermCall perm)
389         (SContT (SetCall set) m)) ()
390 addOccurences (Nominal _ t ) = addOccurences t
391 addOccurences (Pair t1 t2  ) = do addOccurences t1
392                                   addOccurences t2
393 addOccurences (Leaf (Var v)) = nominalDataUnifOccurAdd v 1
394 addOccurences _              = return ()
395
396
397
398
399
400 unifyGen :: (Monad m) =>
401          Term set perm
402      -> Term set perm
403      -> SContT (StackCall stack (Equation set perm))
404         (SContT (OutputCall (OutputSolution set perm))
405         (UFStoreUnif set perm store
406         (SContT (PermCall  perm   )
407         (SContT (SetCall   set    )
408         (SContT (ErrorCall [Char]) m)))))
```

```
409             ()
410
411 unifyGen s t = do clv 'p2 $ ufStoreInitUnif
412                   clv 'p2 $ addOccurences s
413                   clv 'p2 $ addOccurences t
414                   stackPush (Equation SetEmpty [s,t])
415                   solveLoop
416
417
418
419 unifyP ::
420          Term (Set.Set Atm) (Map.Map Atm Atm)
421       -> Term (Set.Set Atm) (Map.Map Atm Atm)
422       -> Either [Char] [OutputSolution (Set.Set Atm) (Map.Map Atm Atm)]
423
424 unifyP s t = runIdentity        $
425              runErrorCallErrorT $
426              runSetPersistent   $
427              runPermPersistent  $
428              runUFStoreUnifP    $
429              do (_,sol) <- runOutputP     $
430                            runStackStateT $
431                            unifyGen s t
432                 return sol
```

A.9 Environment Layer

```
1  {-# LANGUAGE RankNTypes,
2               FlexibleInstances,
3               MultiParamTypeClasses,
4               NoMonomorphismRestriction,
5               KindSignatures,
6               ExistentialQuantification #-}
7
8  module Env where
9
10 import Call
11 import Naming
12
13 import Control.Monad
14 import Control.Monad.State
15
16
17 data OpenClose = Open | Close
18  deriving (Eq,Ord)
19
20 data Side = SLeft | SRight
21  deriving (Eq,Ord)
22
23 data EnvAction set perm =
24        EnvCompose  Side (Perm set perm)
25     |  EnvSetFresh Atm   Bool
26
27
28 data EnvCall set perm (m :: * -> *) r =
29        EnvPerm                           (Perm set perm -> r)
30     |  EnvImage     Atm                 (Atm           -> r)
31     |  EnvImageInv  Atm                 (Atm           -> r)
32     | EnvIsFresh  Atm                   (Bool          -> r)
33     | EnvSet                            (Set set       -> r)
34     | EnvLocal  OpenClose               (()            -> r)
35     | EnvAction (EnvAction set perm)    (()            -> r)
36
37
38 envPerm              = callT $ EnvPerm
39
```

```
40 envSet                = callT $ EnvSet
41
42 envImage        a     = callT $ EnvImage      a
43
44 envImageInv     a     = callT $ EnvImageInv   a
45
46 envIsFresh      a     = callT $ EnvIsFresh    a
47
48 envComposeLeft  perm = callT $ EnvAction    (EnvCompose
49                                                    SLeft
50                                                    perm
51                                             )
52
53 envComposeRight perm = callT $ EnvAction    (EnvCompose
54                                                    SRight
55                                                    perm
56                                             )
57
58 envSetFresh     a v  = callT $ EnvAction    (EnvSetFresh a v)
59
60
61 envLocal lv m = do lv $ callT $ EnvLocal Open
62                    r <- m
63                    lv $ callT $ EnvLocal Close
64                    return r
65
66
67 -- THE INTERPRETATION FUNCTION
68
69 type StateEnv set perm =
70     ((Perm set perm) , Set set , [[EnvAction set perm]])
71
72 myget :: (Monad m) =>
73   StateT (StateEnv set perm) m (StateEnv set perm)
74 myget     = get
75
76 mygets :: (Monad m) => (StateEnv set perm -> a)
77                     -> StateT (StateEnv set perm) m a
78 mygets    = gets
79
80 myput :: (Monad m) =>       StateEnv set perm
```

```
81                              -> StateT (StateEnv set perm) m ()
82 myput      = put
83
84 mymodify :: (Monad m) => (StateEnv set perm -> StateEnv set perm)
85                            -> StateT (StateEnv set perm) m ()
86 mymodify = modify
87
88
89 liftStateEnvT :: (Monad m) =>      m a
90                                       -> StateT (StateEnv set perm) m a
91 liftStateEnvT = lift
92
93
94 getStatePerm     = mygets (\(p,_,_) -> p)
95 getStateSet      = mygets (\(_,s,_) -> s)
96 getStateUndo     = mygets (\(_,_,u) -> u)
97 setStatePerm p   = mymodify (\(_,s,u) -> (p,s,u))
98 setStateSet   s  = mymodify (\(p,_,u) -> (p,s,u))
99 setStateUndo u   = mymodify (\(p,s,_) -> (p,s,u))
100
101
102 modifyStatePerm f = do (p,s,u) <- myget
103                        p' <- f p
104                        myput (p',s,u)
105
106 modifyStateSet  f = do (p,s,u) <- myget
107                        s' <- f s
108                        myput (p,s',u)
109
110 modifyStateUndo f = do (p,s,u) <- myget
111                        u' <- f u
112                        myput (p,s,u')
113
114
115 clv'perm :: (Monad m) =>
116          m a
117       -> StateT (StateEnv set perm)
118                 (SContT theclass m) a
119
120 clv'perm = liftStateEnvT . clv'p1
121
```

```
122
123 clv'set :: (Monad m) =>
124          m a
125       -> StateT (StateEnv set perm)
126                (SContT theclass (SContT theclass1 m)) a
127
128 clv'set = liftStateEnvT . clv'p2
129
130
131 runEnvAction :: (Monad m) =>
132          EnvAction set perm
133       -> StateT (StateEnv set perm)
134            (SContT theclass
135            (SContT (PermCall perm)
136            (SContT (SetCall set) m)))
137               ()
138
139 runEnvAction (EnvCompose  SLeft  p  ) =
140        modifyStatePerm (\perm -> clv'perm $
141                                 permCompose MutateRight p perm
142                         )
143
144 runEnvAction (EnvCompose  SRight p  ) =
145        do modifyStatePerm (\perm -> clv'perm $
146                                  permCompose MutateLeft  perm p
147                           )
148           modifyStateSet  (\set  ->
149                                  do let p' = permInverse p
150                                     set' <- clv'perm $
151                                                 setPermute True p' set
152                                     return set'
153                           )
154
155 runEnvAction (EnvSetFresh        a v) =
156        modifyStateSet (\set   -> clv'set $
157                               setSet True a v set
158                       )
159
160
161
162 envOppositeAction :: (Monad m) =>
```

```
163       EnvAction set1 perm1
164   -> StateT (StateEnv set perm)
165         (SContT theclass
166         (SContT theclass1
167         (SContT (SetCall set) m)))
168                (EnvAction set1 perm1)
169
170 envOppositeAction (EnvCompose  side  p   ) =
171       do let p' = permInverse p
172          return $ EnvCompose side p'
173
174 envOppositeAction (EnvSetFresh a      _  ) =
175       do set <- getStateSet
176          v   <- clv'set $ setIsIn set a
177          return $ EnvSetFresh a v
178
179
180 interpretEnvT :: (Monad m) =>
181       EnvCall set perm t
182         (SContT theclass
183         (SContT (PermCall perm)
184         (SContT (SetCall set) m)) b)
185      -> StateT (StateEnv set perm)
186            (SContT theclass
187            (SContT (PermCall perm)
188            (SContT (SetCall set) m)))
189                  b
190
191 interpretEnvT (EnvPerm        k) =
192      do perm <- getStatePerm
193         liftStateEnvT $ k perm
194
195 interpretEnvT (EnvSet         k) =
196      do set  <- getStateSet
197         liftStateEnvT $ k set
198
199 interpretEnvT (EnvImage    a  k) =
200      do perm <- getStatePerm
201         a'   <- clv'perm $ permImage      perm a
202         liftStateEnvT $ k a'
203
```

```
204 interpretEnvT (EnvImageInv a  k) =
205     do perm <- getStatePerm
206        a'   <- clv'perm $ permImageInv  perm a
207        liftStateEnvT $ k a'
208
209 interpretEnvT (EnvIsFresh  a  k) =
210     do set  <- getStateSet
211        v    <- clv'set $ setIsIn       set  a
212        liftStateEnvT $ k v
213
214 interpretEnvT (EnvLocal Open  k) =
215     do modifyStateUndo (\l -> return ([] : l))
216        liftStateEnvT $ k ()
217
218 interpretEnvT (EnvLocal Close k) =
219     do undo  <- getStateUndo
220        (x,l) <- case undo of
221                     []      -> error "Undo list error"
222                     (x : l) -> return $ (x , l)
223        setStateUndo l
224        forM x runEnvAction
225        liftStateEnvT $ k ()
226
227 interpretEnvT (EnvAction act  k) =
228     do modifyStateUndo
229            (\l -> case l of
230                      []       -> error "Undo list error"
231                      (x : l2) -> do act' <- envOppositeAction act
232                                     return $ (act' : x) : l2
233            )
234        runEnvAction act
235        liftStateEnvT $ k ()
236
237
238 runEnv :: (Monad m) =>
239          SContT   (EnvCall set perm)
240           (SContT theclass
241           (SContT (PermCall perm)
242           (SContT (SetCall  set) m))) a
243       -> SContT   theclass
244           (SContT (PermCall perm)
```

```
245             (SContT (SetCall set) m)) a
246
247 runEnv c = do v <- runStateT (instanciateT interpretEnvT c)
248                                          (PermId,SetEmpty,[])
249                  return $ fst v
```

A.10 α-equivalence and Matching Algorithms

```
1  {-# LANGUAGE RankNTypes,
2               FlexibleInstances,
3               MultiParamTypeClasses,
4               NoMonomorphismRestriction,
5               KindSignatures,
6               ExistentialQuantification #-}
7
8  module Matching where
9
10 import Base
11 import Naming
12 import Call
13 import Term
14 import Env
15 import Error
16 import Store
17
18 import Control.Monad
19 import Control.Monad.Identity
20 import Control.Monad.State
21
22
23 import qualified Data.Map as Map
24 import qualified Data.Set as Set
25
26
27 runBasePersistent ::
28          SContT (PermCall (Map.Map Atm Atm))
29         (SContT (SetCall (Set.Set Atm))
30         (SContT (ErrorCall e) Identity))  a
31      -> Either e a
32 runBasePersistent = runIdentity .
33                     runErrorCallErrorT .
34                     runSetPersistent .
35                     runPermPersistent
36
37
38 -- Freshness Context Calls
39
```

```
40
41 data FreshContextCall set (m :: * -> *) r =
42       FreshConstraint  Var (Set set) (()              -> r)
43     | FreshContextRule Var           (Set set         -> r)
44     | FreshToList                    ([(Var,Set set)] -> r)
45
46
47 freshConstraint :: (Monad m) =>
48             Var
49         -> Set set
50         -> SContT (FreshContextCall set) m ()
51 freshConstraint v s = callT $ FreshConstraint v s
52
53 freshContextRule :: (Monad m) =>
54             Var
55         -> SContT (FreshContextCall set) m (Set set)
56 freshContextRule v = callT $ FreshContextRule v
57
58 freshToList :: (Monad m) =>
59             SContT (FreshContextCall set) m [(Var,Set set)]
60 freshToList        = callT   FreshToList
61
62
63
64 -- The algorithms
65
66
67 checkAtom :: (Monad m) =>
68           Atm
69       -> SContT (EnvCall set perm)
70          (SContT (FreshContextCall set)
71          (SContT (PermCall perm)
72          (SContT (SetCall set)
73          (SContT (ErrorCall [Char]) m))))
74             Atm
75
76 checkAtom a = do frs <- envIsFresh a
77                  if frs
78                   then clv 'p4 $ raiseError "Freshness error"
79                   else envImage a
80
```

```
81
82 -- ENVIRONMENT ELIMINATION PHASE
83
84 unEnvVar :: (Monad m) =>
85          Var
86       -> SContT (EnvCall set perm)
87          (SContT (FreshContextCall set) m) (Term set perm)
88 unEnvVar v = do p  <- envPerm
89                 s  <- envSet
90                 clv'p1 $ freshConstraint v s
91                 return $ perm p (var v)
92
93
94 -- CHECK TERM PHASE
95
96 data StreamCall a b (m :: * -> *) r = StreamStep a (b -> r)
97
98 streamStep :: (Monad m) => a -> SContT (StreamCall a b) m b
99 streamStep a = callT $ StreamStep a
100
101
102 checkTerm :: (Monad m) =>
103          Term set perm
104       -> SContT (StreamCall Var (Term set perm))
105          (SContT (EnvCall set perm)
106          (SContT (FreshContextCall set)
107          (SContT (PermCall perm)
108          (SContT (SetCall set)
109          (SContT (ErrorCall [Char]) m))))) (Term set perm)
110
111 checkTerm   (Nominal (NPerm  p) t) =
112          envLocal clv'p1 (do clv'p1 $ envComposeRight p
113                              checkTerm t
114                          )
115 checkTerm   (Nominal (Abs    a) t) =
116          do b <- clv'p1 $ envImage a
117             u <- envLocal clv'p1 (do clv'p1 $ envSetFresh a False
118                                      checkTerm t
119                                  )
120             return (Nominal (Abs b) u)
121 checkTerm   (Pair s t)              = do s' <- checkTerm s
```

```
122                                              t' <- checkTerm t
123                                              return (Pair s' t')
124 checkTerm    (Leaf (Atm a))          = do a2 <- clv'p1 $ checkAtom a
125                                              return $ atm a2
126 checkTerm    (Leaf (Var v))          = streamStep v
127 checkTerm t@(Leaf (Cst c))           = return t
128
129
130
131 -- DECOMPOSITION PHASE
132
133 decompose :: (Monad m) =>
134          Term set perm
135       -> Term set perm
136       -> SContT (StreamCall (Var, Term set perm) ())
137         (SContT (EnvCall set perm)
138         (SContT (FreshContextCall set)
139         (SContT (PermCall perm)
140         (SContT (SetCall set)
141         (SContT (ErrorCall [Char]) m)))))   ()
142
143 decompose (Nominal (NPerm p) t)                      u   =
144         envLocal clv'p1 (do clv'p1 $ envComposeLeft (permInverse p)
145                             decompose t u
146                         )
147
148 decompose  t                      (Nominal (NPerm p) u) =
149         envLocal clv'p1 (do clv'p1 $ envComposeRight p
150                             decompose t u
151                         )
152
153 decompose (Nominal (Abs  a) t) (Nominal (Abs  b) u) =
154         do a2 <- clv'p1 $ envImageInv a
155            b2 <- clv'p1 $ envImage    b
156            envLocal clv'p1 (do clv'p1 $ envSetFresh a2 True
157                                clv'p1 $ envSetFresh b  False
158                                clv'p1 $ envComposeLeft (permSwapping a b2)
159                                decompose t u
160                            )
161
162 decompose (Pair             s t) (Pair             u v) =
```

```
163        do decompose s u
164           decompose t v
165
166 decompose (Leaf (Cst        f)) (Leaf (Cst        g)) =
167        when (f /= g) (clv'p5 $ raiseError $ "Error : " ++
168                                              (show f)   ++
169                                              " /= "     ++
170                                              (show g)
171                              )
172
173 decompose (Leaf (Atm        a)) (Leaf (Atm        b)) =
174         do c <- clv'p1 $ checkAtom b
175            when (a /= c) (clv'p5 $ raiseError "Atoms not equals")
176
177 decompose (Leaf (Var        v))                   t   =
178         do s   <- clv'p2 $ freshContextRule v
179            p   <- clv'p1   envPerm
180            sp  <- clv'p3 $ setPermute False (permInverse p) s
181            l   <- clv'p4 $ setToList sp
182            mapM (\a -> clv'p1 $ envSetFresh a True) l
183            streamStep (v , t)
184
185 decompose                   _                     _   =
186         clv'p5 $ raiseError "Incompatible constructors"
187
188
189 core :: (Monad m) =>
190         Term set perm
191      -> Term set perm
192      -> SContT (StreamCall (Var, Term set perm) ())
193         (SContT (FreshContextCall set )
194         (SContT (PermCall         perm)
195         (SContT (SetCall          set )
196         (SContT (ErrorCall [Char]) m))))
197            ()
198
199 core s t = do (_,m) <- clv'p1 $ runEnv $ runStateT
200                          (instanciateT  interpDec
201                                         (decompose s t)
202                          )
203                          (return ()
```

```
204                               )
205                   m
206   where
207     streamCheckTerm u =
208            instanciateNoT (\(StreamStep v k) -> unEnvVar v >>= k)
209            (checkTerm u)
210
211     interpDec (StreamStep (v,t) k) =
212            do u <- liftStateT $ streamCheckTerm t
213               modify (\m -> m >> streamStep (v,u))
214               liftStateT $ k ()
215
216
217
218
219 -- ALPHA
220
221 alpha :: (Monad m) =>
222          Term set perm
223       -> Term set perm
224       -> SContT (FreshContextCall set)
225          (SContT (PermCall perm)
226          (SContT (SetCall  set)
227          (SContT (ErrorCall [Char]) m)))
228            ()
229 alpha s t = instanciateNoT interpretAlpha (core s t)
230   where
231     interpretAlpha (StreamStep (v , t) k) =
232          do Susp p u <- clv 'p1 $ termHeadPerm t
233             r <- case u of
234                    Leaf (Var w) ->
235                      if w == v
236                       then do set <- clv 'p2 $ permSupp p
237                                freshConstraint v set
238                       else clv 'p3 $ raiseError "No subtitutions in alpha"
239                    _    -> clv 'p3 $ raiseError "No subtitutions in alpha"
240             k r
241
242
243
244 match :: (Monad m) =>
```

```
245          Term set perm
246       -> Term set perm
247       -> SContT (FreshContextCall set)
248          (SContT (PermCall perm)
249          (SContT (SetCall set)
250          (SContT (ErrorCall [Char])
251          (SContT (StoreCall store Var) m))))
252             (Var -> Maybe (Term set perm))
253
254 match s t =
255       do i <- clv 'p4 $ storeInit Nothing
256          runStateCall i $ do instanciateT interpretMatch (core s t)
257                              stateGet >>= clv 'p5 . storeToFun
258
259   where
260     interpretMatch (StreamStep (v,t) k) =
261        do stateModifyM (\store ->
262              do mu    <- clv 'p5 $ storeGet store v
263                 case mu of
264                    Nothing ->
265                       clv 'p5 $ storePut store v (Just t)
266
267                    Just u  ->
268                       do clv 'p1 $ alpha u t
269                          if ((termSize u) > (termSize t))
270                           then (clv 'p5 $ storePut store v (Just u))
271                           else return store
272              )
273           clv 'p1 $ k ()
274
275
276
277
278
279
280
281 -- Solving And Checking Freshness
282
283
284 instanciateFreshContextSolve :: (Monad m) =>
285          SContT (FreshContextCall set)
```

```
286         (SContT (PermCall perm)
287         (SContT (SetCall  set)
288         (SContT (ErrorCall [Char])
289         (SContT (StoreCall store Var) m)))) a
290      -> SContT (StateCall (store Var (Set set)))
291         (SContT (PermCall perm)
292         (SContT (SetCall  set)
293         (SContT (ErrorCall [Char])
294         (SContT (StoreCall store Var) m)))) a
295
296 instanciateFreshContextSolve m =
297        instanciateT interpretFreshContextSolve m
298   where
299     interpretFreshContextSolve (FreshConstraint v s k) =
300         do stateModifyM (\store ->
301                   do vs <- clv'p4 $ storeGet store v
302                      s' <- clv'p2 $ setUnion MutateLeft vs s
303                      clv'p4 $ storePut store v s'
304                )
305             clv'p1 $ k ()
306
307     interpretFreshContextSolve (FreshContextRule v   k) =
308         clv'p1 $ k SetEmpty
309
310     interpretFreshContextSolve (FreshToList          k) =
311         do l <- stateGetsM (clv'p4 . storeToList)
312             clv'p1 $ k l
313
314
315
316 instanciateFreshContextCheck :: (Monad m) =>
317           store Var (Set set)
318      -> SContT (FreshContextCall set)
319         (SContT (PermCall perm)
320         (SContT (SetCall  set)
321         (SContT (ErrorCall [Char])
322         (SContT (StoreCall store Var) m)))) a
323      -> SContT (StateCall (store Var (Set set)))
324         (SContT (PermCall perm)
325         (SContT (SetCall  set)
326         (SContT (ErrorCall [Char])
```

```
327         (SContT (StoreCall store Var) m)))) a
328
329 instanciateFreshContextCheck storehypothesis m =
330         instanciateT interpretFreshContextCheck m
331   where
332     interpretFreshContextCheck (FreshConstraint v s k) =
333         do store <- stateGet
334            vs    <- clv 'p4 $ storeGet store v
335            b     <- clv 'p2 $ setIsSubSet s vs
336            when (not b) (clv 'p3 $ raiseError "Constraints not in context")
337            clv 'p1 $ k ()
338
339     interpretFreshContextCheck (FreshContextRule v   k) =
340         do s <- clv 'p4 $ storeGet storehypothesis v
341            clv 'p4 $ storePut storehypothesis v SetEmpty
342            clv 'p1 $ k s
343
344     interpretFreshContextCheck (FreshToList          k) =
345         do l <- stateGetsM (clv 'p4 . storeToList)
346            clv 'p1 $ k l
347
348
349
350
351 -- Alpha and Matching, Solving and checking
352
353
354 alphaSolve              s t =
355     instanciateFreshContextSolve            $ alpha s t
356
357 alphaCheck hypothesis s t =
358     instanciateFreshContextCheck hypothesis $ alpha s t
359
360
361 matchSolve              s t =
362     instanciateFreshContextSolve            $ match s t
363
364 matchCheck hypothesis s t =
365     instanciateFreshContextCheck hypothesis $ match s t
366
367
```

```
368 -- We assume that the freshness context is already in the store
369
370
371 data Rule set perm =
372      Rule { rule'context :: [(Var, Set set)] ,
373             rule'left    :: Term set perm ,
374             rule'right   :: Term set perm
375           }
376  deriving (Eq,Ord,Show,Read)
377
378
379 -- Rename the variable of a rule
380 freshRule ::
381            (Var -> Var)
382         -> Rule set perm
383         -> Rule set perm
384
385 freshRule renamevar (Rule c l r) =
386      Rule (map (\(v,s) -> (renamevar v , s)) c)
387           (substitute (Just . var . renamevar) l)
388           (substitute (Just . var . renamevar) r)
389
390
391 rewriteNoFresh :: (Monad m) =>
392          Rule set perm
393      -> Term set perm
394      -> SContT (StateCall (store Var (Set set)))
395         (SContT (PermCall perm)
396         (SContT (SetCall set)
397         (SContT (ErrorCall [Char])
398         (SContT (StoreCall store Var) m))))
399           (Term set perm)
400
401 rewriteNoFresh (Rule c l r) t =
402   do s <- clv'p4 $ storeFromList SetEmpty c
403      f <- matchCheck s l t
404      return $ substitute f r
405
406
407
408
```

```
409 rewriteFresh renamevar rule t =
410       rewriteNoFresh (freshRule renamevar rule) t
```

A.11 Zipper Layer

```
1  {-# LANGUAGE RankNTypes,
2        FlexibleInstances,
3        MultiParamTypeClasses,
4        NoMonomorphismRestriction,
5        KindSignatures,
6        ExistentialQuantification,
7        GeneralizedNewtypeDeriving #-}
8
9
10 module Zipper where
11
12 import Base
13 import Call
14 import Term
15
16 import Control.Monad.State
17
18
19 type Hole term = term -> term
20
21 fillHole :: (Hole term) -> term -> term
22 fillHole = id
23
24 type Context term = [Hole term]
25
26 data Zipper term  = Zipper
27   { zip'term     :: term ,
28     zip'context  :: Context term
29   }
30  deriving (Show)
31
32 zipMake :: term -> Zipper term
33 zipMake t = Zipper t []
34
35
36 data Dir dir  = Up | Down dir
37   deriving (Eq,Ord,Read,Show)
38
39
```

```
40 zipMove :: (Eq dir) =>
41          (term -> [(dir, (term, Hole term))])
42      -> Dir dir
43      -> Zipper term
44      -> Maybe (Zipper term)
45
46 zipMove _       Up        (Zipper _      []    ) =
47    Nothing
48
49 zipMove _       Up        (Zipper t (c : l)) =
50    return $ Zipper (fillHole c t) l
51
52 zipMove tdirs (Down d) (Zipper t     l     ) =
53    do (t', c) <- lookup d (tdirs t)
54       return $ Zipper t' (c : l)
55
56
57
58
59 zipUpdate :: term -> Zipper term -> Zipper term
60 zipUpdate u (Zipper _ l) = Zipper u l
61
62
63
64 zipDirs ::
65          (term -> [(dir, (term , Hole term))])
66      -> Zipper term -> [Dir dir]
67
68 zipDirs tdirs (Zipper t []) =        map (Down . fst) (tdirs t)
69 zipDirs tdirs (Zipper t _ ) = Up : (map (Down . fst) (tdirs t))
70
71
72
73 -- First Class Monadic Signature
74
75
76 data ZipCall term dir (m :: * -> *)   r =
77       ZipTerm                 (term       -> r)
78     | ZipUpdate term          (()         -> r)
79     | ZipDirs                 ([Dir dir] -> r)
80     | ZipMove    (Dir dir)    (Bool       -> r)
```

```
81
82
83 zipCallTerml      = callT   ZipTerm
84 zipCallUpdate t = callT $ ZipUpdate t
85 zipCallDir        = callT   ZipDirs
86 zipCallMove    d = callT $ ZipMove d
87
88
89
90
91 interpretZip :: (Monad m, Eq dir) =>
92          (term -> [(dir, (term, Hole term))])
93      -> ZipCall term dir n (m b)
94      -> StateT (Zipper term) m b
95
96 interpretZip tdirs = aux
97  where
98    aux (ZipTerm      k) =
99          do t <- gets zip'term
100            liftStateT $ k t
101
102   aux (ZipUpdate u k) =
103         do modify (\(Zipper _ c) -> Zipper u c)
104            liftStateT $ k ()
105
106   aux (ZipDirs      k) =
107         do zipper <- get
108            liftStateT $ k (zipDirs tdirs zipper)
109
110   aux (ZipMove d    k) =
111         do zipper <- get
112            r      <- case zipMove tdirs d zipper of
113                          Nothing -> return False
114                          Just z  -> do put z
115                                        return True
116            liftStateT $ k r
117
118
119
120 runZip :: (Monad m, Eq dir) =>
121         (term -> [(dir, (term, Hole term))])
```

```
122         -> term
123         -> SContT (ZipCall term dir) m a
124         -> m a
125
126 runZip tdirs t m = runStateT (instanciateT (interpretZip tdirs) m)
127                              (zipMake t)
128                    >>= return . fst
```

Appendix B

Objective CAML Code

B.1 Quadratic Nominal Unification

```
1  type atm = int
2  type cst = int
3  type var = int
4
5  (******************************************)
6  (* Helpers on Arrays *)
7
8
9  let mkArr n = Array.init n (fun x -> x)
10
11 let permuteArr f arr =
12  let arr2 = Array.copy arr in
13  (for i = 0 to (Array.length arr2) - 1
14     do arr2.(f i) <- arr.(i) done ;
15   arr2
16  )
17
18 let array'diff arr l =
19   let arr2 = Array.copy arr in
20   (List.iter (fun (i,e) -> arr2.(i) <- e) l ;
21    arr2
22   )
23
24 let maxAtoms = 30
```

```
25 let allAtoms = mkArr maxAtoms
26
27
28 (***********************************************)
29 (* Perms *)
30
31 type perm = Perm of int array
32 let unPerm (Perm p) = p
33
34 let idPerm = Perm allAtoms
35 let image   (Perm p) a = p.(a)
36 let inverse      p     = Perm (permuteArr (image p) allAtoms)
37
38 let compose (Perm p1) (Perm p2) =
39     Perm (Array.map (fun x -> p1.(p2.(x))) allAtoms)
40
41 let swap a b = function
42  | i when i = a -> b
43  | i when i = b -> a
44  | i            -> i
45
46 let swapping a b = Perm (Array.map (swap a b) allAtoms)
47 let swapPermL a b p = compose (swapping a b) p
48
49
50 (***********************************************************************)
51 (* FrsSet *)
52
53 type frsSet = FrsSet of bool array
54 let unFrsSet (FrsSet fs) = fs
55
56 let emptyFrsSet = FrsSet (Array.map (fun _ -> false) allAtoms)
57 let isInFrsSet (FrsSet fs) a = fs.(a)
58
59 (* permute p fs = p(fs) *)
60 let permute p (FrsSet fs) = FrsSet (permuteArr (image p) fs)
61
62 let combineFrsSet op (FrsSet fs1) (FrsSet fs2) =
63   FrsSet (Array.map (fun i -> op fs1.(i) fs2.(i)) allAtoms)
64 let unionFrsSet = combineFrsSet (or)
65
```

```
66
67 let frsSetSet b l (FrsSet fs) =
68     FrsSet (array'diff fs (List.map (fun i -> (i,b)) l))
69
70 let frsSetAdd = frsSetSet true
71 let frsSetDel = frsSetSet false
72
73 let dsPerm p1 p2 = FrsSet (Array.map
74       (fun a -> (image p1 a) <> (image p2 a)) allAtoms)
75
76
77 (*****************************************************************)
78 (* DAG *)
79
80
81 type node = { (* Data not changed by the algorithm *)
82                       head          : head                              ;
83               mutable fathers       : node list                         ;
84
85               (* Data changed by the algorithm *)
86               mutable deleted       : bool                              ;
87               mutable edges         : ((perm * node) * del_edge) list   ;
88               mutable pointer       : (perm * node) option              ;
89
90               (* Nominal data (changed also by the alogorithm) *)
91               mutable fresh_cond    : frsSet
92             }
93
94 and head = Atm  of atm
95          | Cst  of cst
96          | Var  of var
97          | Abs  of atm  * node
98          | Pair of node * node
99          | Perm of perm * node
100
101 and del_edge = { mutable edge_deleted : bool }
102
103
104 (****************************************************************)
105 (* Node management *)
106
```

```
107 let is_same_node (n1 : node) (n2 : node) = n1 == n2
108
109 let list_node_iter f = List.iter (function n -> if n.deleted then ()
110                                                  else f n
111                                   )
112
113 let addFrsCond r frs  = r.fresh_cond <- unionFrsSet r.fresh_cond frs
114 let addFrsAtms r b    = r.fresh_cond <- frsSetAdd b r.fresh_cond
115
116 (***************************************************************)
117 (* Edge management *)
118
119 let rec reduceHead r pi s = match (r.head , s.head) with
120  | (Perm (pi2,r1), _ ) -> reduceHead r1 (compose (inverse pi2) pi) s
121  | (_ , Perm (pi2,s2)) -> reduceHead r (compose pi pi2) s2
122  | (_ , _ )            -> (r,pi,s)
123
124
125 let add_undirected_edge r pi s =
126  let (r1, pi1, s1) = reduceHead r pi s in
127  let pi1inv    = inverse pi1
128  and del_egde  = { edge_deleted = false }
129  in  (r1.edges <- ((pi1   ,s1) , del_egde) :: r1.edges ;
130       s1.edges <- ((pi1inv,r1) , del_egde) :: s1.edges
131      )
132
133
134 let list_edges_iter f =
135    List.iter (function ((pi,n),d) -> if n.deleted or d.edge_deleted
136                                      then ()
137                                      else f ((pi,n),d)
138              )
139
140 let delete_edge (_,_,d) = d.edge_deleted <- true
141
142
143
144 (********************************************************)
145 (* DAG *)
146
147
```

```
148
149 type dag = { nonvar      : node list ;
150              variables   : node list ;
151              suspended   : node list
152            }
153
154
155 let reset_node n =
156  ( n.deleted      <- false      ;
157    n.edges        <- []         ;
158    n.pointer      <- None
159  )
160
161 let reset_dag d =
162  ( List.iter reset_node d.nonvar     ;
163    List.iter reset_node d.variables  ;
164    List.iter reset_node d.suspended
165  )
166
167
168 (*********************************************)
169 (* Substitution *)
170
171 let maxVars = 30
172 let sigma = Array.create maxVars None
173
174
175
176
177 (*********************************************)
178 (* Unification Algorithm *)
179
180
181
182 let push stack n = (stack := n :: !stack)
183 let pop  stack   = match !stack with
184  | []         -> failwith "Error: empty stack"
185  | (n :: st) -> (stack := st ; n)
186
187
188 let check_clash r pi s =
```

```
189  let (r2,pi2,s2) = reduceHead r pi s in
190  match (r2.head,s2.head) with
191  | (Var _        , _         )
192  | (_            , Var _     )
193  | (Abs (_,_)    , Abs (_,_) )
194  | (Pair (_,_)   , Pair (_,_) )                              -> ()
195  | (Atm a        , Atm b     ) when a  = image pi b
196                                 && not (isInFrsSet r.fresh_cond a)
197                                 && not (isInFrsSet s.fresh_cond b) -> ()
198  | (Cst c1       , Cst c2    ) when c1 = c2                   -> ()
199  | (_            , _         )                 -> failwith "FAIL-CLASH"
200
201
202 let rec propagate_unification_edges r pi s =
203  let (r2,pi2,s2) = reduceHead r pi s in
204  match (r2.head,s2.head) with
205  | (Var _           , _           )
206  | (_               , Var _       ) -> failwith "propagation␣failed:␣var"
207  | (Atm _           , Atm _       )
208  | (Cst _           , Cst _       ) -> ()
209  | (Pair (r1,r2)    , Pair (s1,s2) ) -> (add_undirected_edge r1 pi s1 ;
210                                          add_undirected_edge r2 pi s2 ;
211                                         )
212  | (Abs (a,r1)      , Abs  (b1,s1) ) -> let b2  = image pi b1
    in
213                                         let pi2 = compose (swapping a b2) pi in
214                                         (add_undirected_edge r1 pi2 s1 ;
215                                          addFrsAtms r [b2]
216                                         )
217  | (_               , _           ) -> assert false
218
219
220
221 let rec propagate_freshness_perm fc r = match r.head with
222  | Perm (p,r1) -> propagate_freshness_perm (permute (inverse p) fc) r1
223  | _           -> addFrsCond r fc
224
225
226
227 let propagate_freshness r =
228   let rec aux fc r = match r.head with
```

```
229      | Cst _
230      | Atm _ -> ()
231      | Var x -> print_string "r.fresh_cond # x"
232      | Pair (r1,r2) -> (propagate_freshness_perm fc r1;
233                         propagate_freshness_perm fc r2
234                        )
235      | Abs (a,r1)   -> propagate_freshness_perm (frsSetDel [a] fc) r1
236      | Perm (p,r1)  -> aux (permute (inverse p) fc) r1
237   in aux r.fresh_cond r
238
239
240 let rec do_fathers s =
241     list_node_iter (fun t -> match t.head with
242                               | Perm (_,_) -> do_fathers t
243                                | _         -> finish t
244                    )
245                    s.fathers
246
247 and finish r =
248  let stack = ref [] in
249  (push stack (idPerm,r) ;
250   while !stack <> [] do
251       let (pi,s) = pop stack in
252       ((match s.pointer with
253          | Some (pi2,r2) when is_same_node r2 r -> ()
254          | Some _     -> failwith "FAIL-LOOP"
255          | None       -> s.pointer <- Some ((inverse pi),r)
256        ) ;
257        do_fathers s ;
258        check_clash r pi s;
259        (match s.head with
260          | Var x          -> sigma.(x) <- Some ((inverse pi),r)
261          | _              -> propagate_unification_edges r pi s
262        ) ;
263        list_edges_iter (function (e,d) ->
264                            push stack e ;
265                            d.edge_deleted <- true
266                        ) s.edges;
267        if is_same_node r s
268         then addFrsCond r (dsPerm idPerm pi)
269         else (addFrsCond r (permute (inverse pi) s.fresh_cond) ;
```

```
270              s.deleted <- true
271            )
272     ) ;
273     propagate_freshness r ;
274     r.deleted <- true
275   done
276  )
277
278
279
280
281 (**********************************************)
282 (* Main function *)
283
284
285 (* a dag where u = v *)
286 let unifiy dag s t =
287  (reset_dag dag ;
288   add_undirected_edge s idPerm t ;
289   list_node_iter finish dag.nonvar ;
290   list_node_iter finish dag.variables
291  )
```