Proof Search for Minimal Logic in Haskell

Nik Sultana[1]
Mathematical Institute,
University of Munich,
Germany

[1]nik.sultana@yahoo.com

Abstract

This report describes a Haskell implementation of Schwichtenberg's proof search algorithm. Implementations of the algorithm's dependencies are also described: Wand's type inference algorithm, Normalisation by Evaluation, and Pattern Unification.

Contents

1 Introduction **3**
1.1 Running the proof system . . . 3
1.2 Notation . . . 3

2 Terms **4**
2.1 Representation . . . 4
2.2 Class of terms . . . 4
2.2.1 Instantiations . . . 5
2.3 Translation . . . 6
2.4 Substitution . . . 6
Notes . . . 7

3 Types **8**
3.1 Type definitions . . . 8
3.2 Constraint generation . . . 9
3.3 Constraint solution . . . 10
3.3.1 Auxilliary functions . . . 10
3.4 Main function . . . 11
3.5 Examples . . . 11

4 Normalisation by Evaluation **12**
4.1 Semantic values . . . 12
4.2 Evaluation function . . . 12
4.2.1 Context . . . 13
4.3 Reflect and Reify . . . 13
4.4 Main function . . . 13
4.5 Examples . . . 14
Notes . . . 14

5 Pattern unification **15**
5.1 Quantifier prefix . . . 15
5.2 Q-Terms . . . 15
5.3 Patterns . . . 16
5.4 Pattern unification . . . 16
5.4.1 Pruning . . . 18
5.4.2 Auxiliary functions . . . 19
5.5 Main function . . . 20
5.6 Examples . . . 20

Notes . . . 21

6 Proof system **22**

6.1 Supporting definitions . . . 22

6.1.1 Lifting combinators . . . 23

6.1.2 Syntactic checks . . . 23

6.1.3 Harrop normal form . . . 23

6.2 Theorem prover . . . 24

6.2.1 Strip . . . 26

6.2.2 Resolve . . . 27

6.3 Auxilliary functions . . . 30

6.4 Main function . . . 30

7 Conclusion **31**

Acknowledgements **31**

A Examples **32**

A.1 Definitions . . . 32

A.2 Example 1 . . . 32

A.3 Example 2 . . . 33

A.4 Example 3 . . . 35

B Supporting functions **40**

Bibliography **41**

Index **42**

Chapter 1

Introduction

The theorem prover described in this report accepts formulas in a decidable fragment of minimal quantifier logic and, if they are theorems, returns corresponding proof objects as terms in the simply-typed λ-calculus. It implements the algorithm described by Schwichtenberg (2004) based on a restricted higher-order unification described by Miller (1991). This proof search algorithm has been used in the MinLog[1] proof assistant. The algorithm's definition assumes terms to be in long normal form; normalisation is carried out here through 'normalisation by evaluation' (Berger & Schwichtenberg 1991). This last algorithm includes type-indexed functions; types are constructed by means of the type reconstruction algorithm described by Wand (1987). The report's structure mirrors the dependencies of the components making up the proof system. These have been implemented in Haskell (Jones 2003) using the so-called 'Glasgow extensions'.

1.1 Running the proof system

The prover is intended to be run in Haskell's top-level. The code presented here has been tested using GHCi version 6.4.1. Unimportant parts of the source code have been omitted from this report, but the source file of this report also doubles as the full Haskell source code of the implementation in literate programming style. Not all the definitions in this report will immediately appear to be Haskell definitions; this is because the source code presented here has been formatted using the lhs2TEX[2] tool for improved readability.

1.2 Notation

This report is generated from the source Haskell code but has been carefully formatted to make the notation more widely recognisable. The symbol $=$ denotes definition, and the equality test is denoted by the $\equiv$ symbol. The arrow $\rightarrow$ is used both in type signatures and in meta (Haskell) level λ-abstractions. A possibly-empty list of items $\vec{x}$ is typed $[\tau]$ for some type τ, and symbols $[]$ and infix $:$ denote nil and cons respectively. Variables v, x, y, z range over variables, r, s, t over terms – or Q-terms, or patterns: the context will render this clear. Variable Q ranges over quantifier prefixes, n over integers, and variables ρ and σ range over substitutions.

The distinction between meta and object levels in terms of notation is as follows: λ denotes meta-abstraction; object-level abstraction is denoted by $\lambda\!\!\lambda$. Object-level application is denoted by $\bullet$, while at meta-level it is denoted by either \$ or juxtaposition. The symbol $\cdot$ indicates an empty place in a partially-applied function.

[1]`http://www.minlog-system.de/`

[2]`http://people.cs.uu.nl/andres/lhs2tex/`

Chapter 2

Terms

This chapter will describe the implementation of λ-term representations and some standard definitions over them. These term representations are inter-translatable and will be instantiated in a typeclass called *Terms* in §2.2. In the final chapter formulas and sequents too will be instantiated in this class to facilitate their handling. The definitions in this chapter lay the foundations for the chapters that follow, and it concludes with a definition of substitutions and their composition.

2.1 Representation

Two representations for expressions will be defined, both of which use concrete names for variables. At times it is more convenient to use one representation over the other. The first definition consists of the standard notation for λ-expressions.

```
data Exp
  = String           -- variables
  | λ\String Exp     -- abstraction
  | Exp • Exp        -- application
```

The second definition uses *functor-arguments* notation – that is, the operands in nested applications are collected into a list of operands, as are the variable names in nested abstractions. The symbols used above to denote object-level abstraction and application are overloaded, but the intended representation should be clear from the context of use.

```
data FAE
  = String
  | λ\[String] FAE
  | FAE • [FAE]
```

2.2 Class of terms

Irrespective of the representation used for terms – and assuming concrete variable naming – one expects certain operations to be definable over terms. These operations are collected to define the *class* of terms, and the two representations of terms defined above are made instances of this class. In the chapters that follow additional instances of this class will be defined, such as Q-terms described in Chapter 5.

```
class $Term\ a$ where
  $\mathsf{FV} :: a \to [String]$
  $\mathsf{BV} :: a \to [String]$
  $\cdot\dagger :: a \to a$   -- read "compact"
  $\cdot\ddagger :: a \to a$   -- read "compact1"
  $\cdot\Downarrow :: a \to a$   -- read "normalise"
```

The role of *compact* is to standardise the presentation of terms in representations, such as *FAE*, in which a normal form might be expressed in various ways – this contrasts with *Exp* values for which compacting behaves like the identity function. Compacting acts on the dominant constructor in an expression, and subexpressions are subsequently handled by *compact1*. The expected behaviour of the other functions is standard. Definitions can now be made over the whole class due to the interface shared by its members, irrespective of the precise details of how the interface is implemented.

```
$isClosed :: Term\ a \Rightarrow a \to Bool$
$isClosed\ t = (\mathsf{FV}\ t) \equiv [\,]$
```

2.2.1 Instantiations

In this section the representations of terms that were defined earlier are made instances of the class of terms. The instantiation of *Exp* is straightforward and will not be given explicitly here. For this representation, *compact* behaves like the identity function, and *normalise* invokes the normalisation-by-evaluation function that will be described in Chapter 4.
In the instantiation of *FAE*, *compact* has more interesting behaviour and standardises the presentation of nested abstractions and applications. This is quite different from normalising a term since *compact* acts on a more superficial level. Functions defined over this representation of terms may depend on their input terms being in a compact form. In this instantiation *normalise* invokes the normalisation function defined for *Exp* terms, then transforms terms between representations by using the homomorphisms defined in the next section.

```
instance $Term\ FAE$ where
  $\mathsf{FV} = nub \circ \mathsf{FV}_{\mathrm{FAE}}$
  $\mathsf{BV} = nub \circ \mathsf{BV}_{\mathrm{FAE}}$
  $x\dagger = x \bullet [\,]$
  $t\dagger = t\ddagger$
  $x\ddagger = x$
  $(\lambda\!\!\lambda x\ r)\ddagger =$
    case $x$ of
      $[\,] \to r\ddagger$
      $vs \to$ case $r$ of
          $v \to \lambda\!\!\lambda vs\ v$
          $\lambda\!\!\lambda v\ r' \to (\lambda\!\!\lambda(vs +\!\!+ v)\ r')\ddagger$
          $r' \bullet s' \to \lambda\!\!\lambda x\ ((r'\ddagger) \bullet ((\cdot\ddagger)\ \overrightarrow{s'}))$
  $(r \bullet s)\ddagger =$
    case $s$ of
      $[\,] \to r\ddagger$
      $\_ \to$ case $r$ of
          $r' \bullet s' \to (r' \bullet (s' +\!\!+ s))\ddagger$
          $\_ \to (r\ddagger) \bullet ((\cdot\ddagger)\ \overrightarrow{s})$
```

```
$t \Downarrow =$
  let $t' = normalisNbE\ (\overline{t}^{\text{Exp}})$
  in case $t'$ of
    $Nothing \rightarrow t$
    $Just\ t'' \rightarrow (\overline{t''}^{\text{FAE}})\ \dagger$
```

2.3 Translation

The isomorphism between the representations defined for λ-terms is witnessed by the functions defined next.

```
$\overline{x}^{\text{FAE}} = x$
$\overline{(\lambda\!\!\lambda x\ r)}^{\text{FAE}} = \lambda\!\!\lambda[x]\ (\overline{r}^{\text{FAE}})$
$\overline{(r \bullet s)}^{\text{FAE}} = (\overline{r}^{\text{FAE}}) \bullet [\overline{s}^{\text{FAE}}]$
```

The second function works in the opposite direction.

```
$\overline{x}^{\text{Exp}} = x$
$\overline{(\lambda\!\!\lambda x\ r)}^{\text{Exp}} =$
  let $mkAbs\ [\,]\ r = r$
      $mkAbs\ (x : xs)\ r = \lambda\!\!\lambda x\ (mkAbs\ xs\ r)$
  in $mkAbs\ x\ (\overline{r}^{\text{Exp}})$
$\overline{(r \bullet s)}^{\text{Exp}} =$
  let $mkApp\ r\ [\,] = r$
      $mkApp\ r\ (s : ss) = mkApp\ (r \bullet (\overline{s}^{\text{Exp}}))\ ss$
  in $mkApp\ (\overline{r}^{\text{Exp}})\ s$
```

2.4 Substitution

This section defines substitution over *Exp*-values. The equivalent function over *FAE* is definable using the homomorphisms from the previous section. Some operations seemed more natural to define over *Exp* than over *FAE*, and vice versa. For this reason, normalisation (Chapter 4) is defined over *Exp* and unification (§ 5.4) over *FAE*. It is useful to assume that, after substitution, terms are brought into long normal form – this is effected by the functions that invoke the substitution. The following definition formalises that a value of type *Subst* is the graph of an *Exp*-substitution function.

```
type $Subst = [(String, Exp)]$
```

The next definition interprets a substitution into function over terms in order to facilitate its application.

```
$s\{v \mapsto r\} =$
  case $s$ of
    $x \rightarrow$
      if $x \equiv v$
      then $r$
      else $x$
    $\lambda\!\!\lambda x\ t \rightarrow$
      if $x \not\equiv v$
      then $\lambda\!\!\lambda x\ (t\{v \mapsto r\})$
      else $\lambda\!\!\lambda x\ t$
    $s \bullet t \rightarrow (s\{v \mapsto r\}) \bullet (t\{v \mapsto r\})$
```

Function *mkSubst* lifts the previous definition on single substitutions to lists of substitutions – it functionalises values of *Subst*. One could expect that *mkSubst* will appear partially-applied in definitions.

```
    $mkSubst\ [\,]\ r = r$
    $mkSubst\ ((v, r) : ss)\ t = mkSubst\ ss\ (t\{v \mapsto r\})$
```

Composition

The composition of substitutions σ_1 and σ_2, denoted by $\sigma_1 \fatsemi \sigma_2$, involves the following steps:

- First, *dropExtra* trims away from σ_2 any maps that are also defined in σ_1;

```
    $dropExtra :: Subst \to Subst \to Subst$
    $dropExtra\ [\,]\ \sigma_2 = \sigma_2$
    $dropExtra\ \sigma_1\ [\,] = \sigma_1$
    $dropExtra\ (x : xs)\ (y : ys) =$
      if $fst\ x \equiv fst\ y$
      then $dropExtra\ xs\ (dropExtra\ (x : xs)\ ys)$
      else $dropExtra\ xs\ (y : (dropExtra\ (x : xs)\ ys))$
```

- Substitution σ_1 is then applied to the value-terms of σ_2. This makes use of the function *mkSubst* defined earlier. The definition of composition follows:

```
    $\sigma_1 \fatsemi [\,] = \sigma_1$
    $[\,] \fatsemi \sigma_2 = \sigma_2$
    $\sigma_1 \fatsemi \sigma_2 = remSelfMap\ [(x, mkSubst\ \sigma_1\ r) \mid (x, r) \leftarrow (dropExtra\ \sigma_1\ \sigma_2)]$
```

- The last line of the previous definition refers to a function *remSelfMap*. This function removes maps of the sort $x \mapsto x$ from a substitution.

```
    $remSelfMap :: Subst \to Subst$
    $remSelfMap\ [\,] = [\,]$
    $remSelfMap\ (x : xs) =$
      case $x$ of
        $(v, v') \to$
          if $v \equiv v'$
          then $remSelfMap\ xs$
          else $rest$
        $otherwise \to rest$
      where $rest = x : (remSelfMap\ xs)$
```

Notes

It would benefit the implementation to have separate optimised normalisation functions for each representation of λ-terms, or perhaps use a single representation throughout. Against the backdrop of the usual contention between readability and performance, one would expect that this would diminish readability.

Some definitions described in this chapter could be made more generic: for instance, in the representation types described in §2.1, the type of names could be made a parameter rather than fixed as a string. Similarly, the definition of *Subst* could specify its right components to be values of some type *a* in the class *Term*. In addition, one could make *Subst* into an ADT to restrict the formation of substitutions – to ensure, for example, that the graphs are functoid.

Chapter 3

Types

This chapter describes an implementation of the type reconstruction algorithm described by Wand (1987). The algorithm has been modified to work with open terms: this is done by assigning distinct free variables distinct type variables. This assignment corresponds with the principle that in the absence of constraints a variable can have any type.

Wand's algorithm operates in two phases: the first phase involves traversing a term to generate typing constraints; these constraints are then solved using first-order unification.

3.1 Type definitions

The language of types consists of individuals and functions, formalised by the following definition. The overall system will only handle terms typable in this class.

$$
\begin{aligned}
&\textbf{data}\ \tau\\
&\quad = O\\
&\quad \mid \tau \Rightarrow \tau
\end{aligned}
$$

The type reconstruction algorithm uses the following types of intermediate values:

- *TVariable* is the denumerable source of type variables.
- Type τ' is the general analogue to τ, or *type schemes.* The symbols τ' and τ are used to both represent types and range over their values.

  $$
  \begin{aligned}
  &\textbf{data}\ \tau'\\
  &\quad = \tau' \overset{\circ}{\underset{\circ}{\Rightarrow}} \tau'\\
  &\quad \mid \mathit{TVariable}
  \end{aligned}
  $$

- *TEqn* formalises equations between type schemes. A list of *TEqn* forms the type of unification problems: a *unification problem* is a set of equations.

  $\textbf{type}\ \mathit{TEqn} = (\tau', \tau')$

- Γ is a typing context: a finite map from term variables to types. The symbol Γ is used to denote both the type of contexts and also their values.

  $\textbf{type}\ \Gamma = (\mathit{String} \to \tau')$

- *Goal* values are triples consisting of a typing context, an expression that needs to be typed, and the type calculated up to that point.

  type $Goal = (\Gamma, Exp, \tau')$

- *Subs* are substitution functions defined over type schemes.

  type $Subs = (\tau' \to \tau')$

3.2 Constraint generation

The implementation of the constraint-generation phase follows closely the definition given by Wand (1987): it is organised into an algorithmic *skeleton* that invokes the more specific *actionTable* function. The algorithm has been modified to work with open terms in the following way: the function *undefInit* generates an initial typing context by mapping distinct free term variables to distinct type variables, and also provides the function *skeleton* with the seed value for fresh type variable names.

```
undefInit :: Exp → (TVariable, Γ)
undefInit r =
  let fvs = (nub ∘ FV_Exp) r
      fvl = ((toInteger (length fvs)) :: TVariable)
      fvl′ = (λx → $\overset{\circ}{x}$) $\overrightarrow{(countUp\ fvl)}$
      undef = mapp fvs fvl′   -- build a map from free variable names to fresh types
          -- as far as the type-checker is concerned these would be undefined.
  in (fvl, undef)
```

The next two functions together generate constraints to be solved by unification. Function *skeleton* iterates the application of *actionTable* to the initial goal and produces both a unification problem and also the fresh variable name it started with. The latter will be used by the function *typeOf*, defined further down.

```
skeleton :: Exp → ([TEqn], TVariable)
skeleton r =
  let loop :: ([TEqn], [Goal], TVariable) → ([TEqn], [Goal], TVariable)
      loop (l, [ ], mt) = (l, [ ], mt)
      loop (l, (g : gs), mt) = loop (l′ ++ l, gs ++ g′, mt′)
        where (l′, g′, mt′) = actionTable g l mt
      (eqns, _, _) = loop init   -- extract the unification problem
  in (eqns, initv)
  where
      (initv, undef) = undefInit r   -- produces initial variable and also initial typing context
      g = (undef, r, $\overset{\circ}{initv}$)   -- initial goal; use initial typevar ‘init‘
      init = ([ ], [g], initv + 1)   -- (init + 1) is the next fresh variable
```

Function *actionTable* refines goals and, in doing so, produces unification problems.

```
actionTable :: Goal → [TEqn] → TVariable → ([TEqn], [Goal], TVariable)
actionTable (Γ, x, τ′) l mt = ([(τ′, Γ x)], [ ], mt)
actionTable (Γ, r • s, τ′) l mt = ([ ], [g1, g2], mt + 1)   -- adds two new goals
  where
    g1 = (Γ, r, $\overset{\circ}{mt} \overset{\circ}{\Rightarrow} \tau'$)
  $g2 = (\Gamma, s, \overset{\circ}{mt})$
$actionTable\ (\Gamma, \lambda\!\!\lambda v\ \ r, \tau')\ l\ mt = ([(\tau', \overset{\circ}{n_1} \Rightarrow \overset{\circ}{n_2})], [(\Gamma', r, \overset{\circ}{n_2})], mt + 2)$   -- refines the goal
  where
    $n_1 = mt$
    $n_2 = mt + 1$
    $\Gamma' = \lambda x \rightarrow$ if $(x \equiv v)$ then $\overset{\circ}{n_1}$ else $(\Gamma\ x)$   -- extends the typing context
```

3.3 Constraint solution

The type inference process concludes with the generation of a principal type for the term, if one exists, by solving the constraints generated during the first phase. A solution to a unification problem is a substitution σ such that, for each equation $r \overset{?}{=} s$ in the problem, it is the case that $\sigma r \equiv \sigma s$.

```
$foUnifn :: [\,TEqn\,] \rightarrow Maybe\ Subs$
$foUnifn\ p =$
  case $p$ of
    $[\,] \rightarrow Just\ id$
    $((\tau'_1 \overset{\circ}{\Rightarrow} \tau'_2, \overset{\circ}{n}) : es) \rightarrow foUnifn\ ((\overset{\circ}{n}, \tau'_1 \overset{\circ}{\Rightarrow} \tau'_2) : es)$
    $((\tau'_1 \overset{\circ}{\Rightarrow} \tau'_2, \tau''_1 \overset{\circ}{\Rightarrow} \tau''_2) : es) \rightarrow foUnifn\ ((\tau'_1, \tau''_1) : (\tau'_2, \tau''_2) : es)$
    $(p@(\overset{\circ}{n}, \tau'_1 \overset{\circ}{\Rightarrow} \tau'_2) : es) \rightarrow$
      if $occursCheck\ n\ (snd\ p)$
      then $Nothing$
      else $composeF\ p\ n\ es$
    $(p@(\overset{\circ}{n_1}, \overset{\circ}{n_2}) : es) \rightarrow$
      if $n_1 \equiv n_2$
      then $foUnifn\ es$
      else $composeF\ p\ n_1\ es$
    where $composeF\ p\ n\ es =$
      let $f = (snd\ p)\{n \mapsto \cdot\}$
          $g\ (p1, p2) = (f\ p1, f\ p2)$   -- to map over list of pairs
          $rest = foUnifn\ (g\ \overrightarrow{es})$
      in case $rest$ of
      $Nothing \rightarrow Nothing$
      $Just\ s \rightarrow Just\ (s \circ f)$
```

3.3.1 Auxilliary functions

Function *occursCheck* is an elementary check made during unification, *subst* constructs substitutions over type constraints, and *leastInstance* coerces type schemes into simple types – thus rendering the types, which are calculated by this algorithm, usable by the implementation of normalisation-by-evaluation (Chapter 4).

```
$occursCheck :: TVariable \rightarrow \tau' \rightarrow Bool$
$occursCheck\ i\ \ \overset{\circ}{n} = i \equiv n$
$occursCheck\ i\ (\tau'_1 \overset{\circ}{\Rightarrow} \tau'_2) = (occursCheck\ i\ \tau'_1) \vee (occursCheck\ i\ \tau'_2)$
```

$$\tau'\{i \mapsto \overset{\circ}{n}\} = \mathbf{if}\ n \equiv i\ \mathbf{then}\ \tau'\ \mathbf{else}\ \overset{\circ}{n}$$
$$\tau'\{i \mapsto (\tau'_1 \overset{\circ}{\Rightarrow} \tau'_2)\} = (\tau'\{i \mapsto \tau'_1\}) \overset{\circ}{\Rightarrow} (\tau'\{i \mapsto \tau'_2\})$$

$$\mathit{leastInstance} :: \tau' \to \tau$$
$$\mathit{leastInstance}\ \overset{\circ}{n} = O$$
$$\mathit{leastInstance}\ (\mathit{t1} \overset{\circ}{\Rightarrow} \mathit{t2}) = (\mathit{leastInstance}\ \mathit{t1}) \Rightarrow (\mathit{leastInstance}\ \mathit{t2})$$

3.4 Main function

Function *typeOf* is the principal export of this module. It first obtains a unification problem and an initial variable from *skeleton*, solves the problem to obtain a substitution function – if one exists – using *foUnifn*, and applies this substitution to the initial variable.

```
$\mathit{typeOf}\ r =$
  $\mathbf{let}\ (\mathit{eqns}, \mathit{init}) = \mathit{skeleton}\ r$
  $\mathbf{in\ case}\ (\mathit{foUnifn}\ \mathit{eqns})\ \mathbf{of}$
    $\mathit{Nothing} \to \mathit{Nothing}$
    $\mathit{Just}\ \sigma \to \mathit{Just}\ (\sigma\ \overset{\circ}{\mathit{init}})$
```

3.5 Examples

$$\mathit{typeOf}\ (\lambda\!\!\lambda\texttt{x}\ (\texttt{x} \bullet \texttt{x}))$$
$$\rightsquigarrow \mathit{Nothing}$$
$$\mathit{typeOf}\ (\lambda\!\!\lambda\texttt{f}\ (\lambda\!\!\lambda\texttt{x}\ (\texttt{f} \bullet \texttt{x})))$$
$$\rightsquigarrow \mathit{Just}\ ((\overset{\circ}{3} \overset{\circ}{\Rightarrow} \overset{\circ}{4}) \overset{\circ}{\Rightarrow} (\overset{\circ}{3} \overset{\circ}{\Rightarrow} \overset{\circ}{4}))$$
$$\mathit{typeOf}\ \texttt{x}$$
$$\rightsquigarrow \mathit{Just}\ \overset{\circ}{0}$$
$$\mathit{typeOf}\ (\texttt{x} \bullet \texttt{y})$$
$$\rightsquigarrow \mathit{Just}\ \overset{\circ}{2}$$
$$\mathit{typeOf}\ (\lambda\!\!\lambda\texttt{x}\ \texttt{y})$$
$$\rightsquigarrow \mathit{Just}\ (\overset{\circ}{2} \overset{\circ}{\Rightarrow} \overset{\circ}{0})$$

Chapter 4

Normalisation by Evaluation

In outline, NbE works by first evaluating a term then reifying the resulting value into a term that is β-normal and η-long. The proof search process depends on unification, which is described in the next chapter, and which operates on normal terms. The implementation described below closely follows the description on Wikipedia[1], except that it has been modified to work with open terms.

4.1 Semantic values

The meaning of terms consists in functions between semantic values and constants of the individual type: these are named *LAM* and *SYN* respectively. Since we must also cater for open terms, and there is nothing we can assume about free variables, free variables are treated as syntactical thunks in the space of semantic values. These thunks are constructed by *FREE*. As a result of having these additional values, we also need to represent particular applications explicitly in the semantic space: because applying an open term to a – possibly open – term cannot otherwise yield a semantic value. The constructor for combinations is *APPLY* and is parametrised by two semantic values and the type of the operand: the type needs to be stored in order to be used by the type-indexed function *reify* when generating a term from a semantic value. This will be elaborated further below.

```
data Sem
  = SYN Exp
  | LAM (Sem → Sem)
  | APPLY Sem Sem τ   -- occurs when the operator is a free variable,
      -- whereupon a LAM cannot be constructed.
  | FREE Exp
```

4.2 Evaluation function

The function defined next interprets terms into values of *Sem*. This function is called at the start of the NbE process – cf. the function *nbe* in §4.4. As is normally the case, this function is parametrised by a context, as well as the term to be evaluated.

[1]http://en.wikipedia.org/wiki/Normalisation_by_evaluation

```
$[\![x]\!]_g = lookup\_ctx\ g\ x$
$[\![\lambda\!\!\lambda x\ r]\!]_g = LAM\ (\lambda y \rightarrow [\![r]\!]_{(add\_ctx\ g\ x\ y)})$  -- functions built by threading
    -- the arguments through the meaning.
$[\![r \bullet s]\!]_g =$
  case $[\![r]\!]_g$ of
    $LAM\ w \rightarrow w\ ([\![s]\!]_g)$
    $FREE\ t \rightarrow APPLY\ (FREE\ t)\ ([\![s]\!]_g)\ (leastInstance\ \tau')$  -- this caters for open terms
      where $(Just\ \tau') = typeOf\ s$  -- note type info kept in APPLY thunks.
  -- the following line nests two APPLYs since the first cannot be evaluated
  -- functionally (as in the case of (LAM s) above).
    $r'@(APPLY\ \_\ \_\ \_) \rightarrow APPLY\ r'\ ([\![s]\!]_g)\ (leastInstance\ \tau')$
      where $(Just\ \tau') = typeOf\ s$
```

4.2.1 Context

The previous function is parametrised by a context mapping variable names to semantic values. As with substitutions, defined in §2.4, contexts are represented by their graphs. The implementation is straightforward and the details will be omitted, save that the functions *lookup_ctx* and *add_ctx* are defined over the type of contexts.

4.3 Reflect and Reify

Functions *reflect* and *reify* – denoted by $\uparrow^x_t$ and $\downarrow^x_t$ – are type-indexed functions whose other parameters are the last-used variable x, from which to generate fresh variables, and expressions and semantic values respectively.

```
$\uparrow^x_O\ r = SYN\ r$
$\uparrow^x_{(\tau \Rightarrow \tau_2)}\ r = LAM\ (\lambda w \rightarrow \uparrow^x_{\tau_2}\ (r \bullet (\downarrow^x_\tau\ w)))$
```

The function *reify* has a slightly more complex definition than *reflect* because of its sensitivity to open terms: if we were to restrict our attention to closed terms then the first two clauses of *reify* would have been sufficient. Note that $\sharp_x$ produces the next fresh variable after accepting the current fresh variable x.

```
$\downarrow^x_O\ (SYN\ r) = r$
$\downarrow^x_{(\tau \Rightarrow \tau_2)}\ (LAM\ r) =$
  let $x' = \sharp_x$
      $x'' = \sharp_{x'}$
  in $\lambda\!\!\lambda x'\ (\downarrow^{x''}_{\tau_2}\ (r\ \$\ (\uparrow^{x'}_\tau\ x')))$
$\downarrow^x_\tau\ (APPLY\ r\ s\ \tau_2) =$
  let $x' = \sharp_x$
  in $(\downarrow^x_{(\tau_2 \Rightarrow \tau)}\ r) \bullet (\downarrow^{x'}_{\tau_2}\ s)$
$\downarrow^x_\tau\ (FREE\ r) = r$  -- behaves like SYN
```

4.4 Main function

The function *nbe* combines the above definitions into an NbE implementation: starting with “a” as the first fresh variable, the meaning of expression e of type t is reified.

```
$nbe\ \tau\ t = \downarrow^{\mathtt{a}}_\tau\ ([\![t]\!]_{empty\_ctx})$
```

The principal export of this module is the function *normalisNbE*, which combines the functionality of Wand's algorithm and *nbe*.

```
normalisNbE t =
  case (typeOf t) of
    Nothing → Nothing
    Just τ′ → Just (nbe (leastInstance τ′) t)
```

4.5 Examples

The next set of definitions are drawn from the Wikipedia article cited earlier too.

```
k = λx (λy x)
s = λx (λy (λz ((x • z) • (y • z))))
skk = (s • k) • k
testrun = nbe (O ⇒ O) skk
```

Some results of the NbE implementation are presented next.

```
testrun
 ⇝ λa′ a′

normalisNbE (F • skk)
 ⇝ Just (F • (λa′ a′))

normalisNbE (skk • F)
 ⇝ Just F

normalisNbE (λx (skk • F))
 ⇝ Just (λa′ F)
```

Notes

The implementation would benefit, in terms of clarity, from a ‘cleaner’ handling of fresh variables. It can be rendered more generic by making abstract the type of names, and turning *Ctx* into an ADT.

Chapter 5

Pattern unification

The proof search algorithm that will be described in the next chapter translates proof problems into unification problems – more specifically, the unification of *patterns*. Patterns are instances of a restricted class of λ-terms within a broader class of *Q-terms*. A Q-term is a pair consisting of a quantifier prefix Q and a term whose free variables are captured by the quantifier prefix. A *quantifier prefix* is an instance of a *prefix class*, as used in logic to classify formulas: examples of well-known prefixes include the Bernays-Schönfinkel class $\exists^*\forall^*$ and the Ackermann class $\exists^*\forall\exists^*$, both for FOL. The restricted class from which patterns are drawn is the so-called L_λ class: the quantifier prefix for this class is $\forall^*\exists^*\forall^*$.

Key notions will be formalised next, such as Q-terms and patterns, following which the unification algorithm over patterns is defined. The chapter concludes with applications of the algorithm to some examples. The principal references for this chapter are the articles by Schwichtenberg (2004), Miller (1991) and Nipkow (1993). In particular, some of the examples at the end of the chapter were drawn from Nipkow's article.

5.1 Quantifier prefix

Quantifier prefixes will be restricted to the class $\forall^*\exists^*\forall^*$: the variables bound in each segment are called *signature*, *flexible*, and *forbidden* variables respectively. A quantifier prefix is implemented here as an abstract datatype. This ensures that it can be built and queried using a restricted set of functions and, consequently, that prefixes are restricted to the class of interest. The precise details of the ADT's implementation will be omitted and the behaviour of the interface functions will be summarised instead: nullary *empty* yields a prefix devoid of bound variables; taking Q to range over prefixes and x over variables, $Q^+_L x$, $Q^+_\forall x$, and $Q^+_\exists x$ denote adding x to the signature, flexible, and forbidden variables respectively; $Q^-_L x$, $Q^-_\forall x$, and $Q^-_\exists x$ denote removing x from being a signature, flexible, and forbidden variable respectively; and finally Q_L, $Q_\exists$, $Q_\forall$ return Q's signature, flexible and forbidden variables respectively as a list.

5.2 Q-Terms

A Q-term is formalised as a pair consisting of a quantifier prefix and a term. In unification one tends to work with terms represented in functor-arguments notation, which was defined in §2.1. Q-terms are instantiated in the class *Term*, which was defined in §2.2, as shown below.

```
**instance** *Term QTerm* **where**
  $\mathsf{FV}\ (q, t) = \mathsf{FV}\ t \setminus\setminus (q_L \mathbin{+\!\!+} q_\exists \mathbin{+\!\!+} q_\forall)$
  $\mathsf{BV}\ (q, t) = nub\ \$\ \mathsf{BV}\ t \mathbin{+\!\!+} q_L \mathbin{+\!\!+} q_\exists \mathbin{+\!\!+} q_\forall$
  $(q, t)\dagger = (q, t\dagger)$
```

$(q, t)\ddagger = (q, t\ddagger)$
$(q, t) \Downarrow = (q, t \Downarrow)$

5.3 Patterns

Patterns are Q-terms for which the quantifier prefix is $\forall^*\exists^*\forall^*$ and whose terms are restricted as follows:

- if u is forbidden in Q and $\vec{r}$ are Q-terms, then $u \bullet \vec{r}$ is a Q-term;
- if Y is flexible in Q and $\vec{z}$ are distinct variables forbidden in Q, then $Y \bullet \vec{z}$ is a Q-term;
- if r is a $\mathrm{Q}\overset{+}{\forall} z$-term then $\lambda\!\!\lambda z\ r$ is a Q-term.

Also note that patterns are higher-order: flexible variables may be function variables. The function *isPattern* detects whether a given Q-term is a pattern – or equivalently, an element in L_λ. In order to be checked, a Q-term must be in compact form: this ensures uniform treatment. The checking function sees to normalising a Q-term before applying tests to check whether it is a pattern. It disregards the quantifier prefix and focuses on the structure of terms, since by the definition of Q-terms the quantifier prefix is restricted to a particular class.

```
$isPattern :: QTerm \rightarrow Bool$
$isPattern\ (q, z) = z \in (q_L + q_\exists + q_\forall)$
$isPattern\ (q, \lambda\!\!\lambda z\ r) = isPattern\ (foldl\ (\cdot\overset{+}{\forall})\ q\ z, r)$
$isPattern\ (q, u \bullet r) =$
  if $u \in q_L \lor u \in q_\forall$
  then let $r' = (\lambda x \rightarrow (q, x))\ \vec{r}$   -- turn each term in “r” into a Q-term
    in $foldr\ (\land)\ True\ (isPattern\ \overrightarrow{r'})$   -- check that operands are patterns
  else if $u \in q_\exists$
    then let
          -- forbd tests that each element in list is a forbidden var.
      $forbd\ l = foldR\ test\ True\ l$
                where $test\ z\ xs\ next =$
                  case $(\overline{z}^{\mathrm{Exp}}) \Downarrow$ of
                    $\lambda\!\!\lambda z1\ (z'' \bullet z2) \rightarrow$
                      if $z1 \equiv z2$
                      then $z'' \in q_\forall \land next$
                      else $False$
                    $otherwise \rightarrow False$
      in $distinct\ r \land forbd\ r$
    else $False$   -- since “u” does not appear in quantifier prefix
$isPattern\ \_ = False$
```

5.4 Pattern unification

The Q-prefix carries over to unification problems too: a *Q-unification problem* is a unification problem in the context of quantifier prefix Q and is formalised as follows:

```
type $UProblem = (QPrefix, Eqns\ FAE)$
type $Eqns\ a = [(a, a)]$
```

Values of *UProblem* could also be specified to be a list of Q-terms, but it is more convenient to work with the definition given above since it reflects the expectation that all the Q-terms share the same prefix.

It has been proved, in the articles cited earlier, that the unification problem for L_λ is decidable, yields most general unifiers, and that its quantifier prefix class is closed under unification. The type *UProblem* is made an instance of *Term*; the details are omitted since the instantiation is straightforward.

A unification problem will be solved iteratively by a function of type $UProblem \rightharpoonup (UProblem, Subst)$. Solving the problem involves refining the problem and accumulating a substitution function. This process terminates either when the problem is trivial – that is, all its equations are identities – or when a solution cannot be found. If the problem is solvable, the substitutions of sub-problems are composed to yield a unifier for the problem we started with. A successful solution process could be illustrated as shown below, where U_0 denotes the original Q-unification problem, and U_i the ith refinement.

$$U_0 \longrightarrow_{\rho_1} U_1 \longrightarrow_{\rho_2} U_2 \longrightarrow_{\rho_3} \cdots \longrightarrow_{\rho_{n-1}} U_{n-1} \longrightarrow_{\varepsilon} U_n$$

Let ρ_n be ε. The solution, denoted by ϕ, to U_0 is then $(\rho_1 \fatsemi \cdots \fatsemi \rho_n) \restriction Q_\exists$. Given the iterative nature of the algorithm ϕ can be expressed as $(\rho_1 \fatsemi \phi') \restriction Q_\exists$ where ϕ' is the solution to U_1. The pattern unification algorithm is defined next by cases on the shape of its input.

```
$unify' :: UProblem \to Maybe\ (UProblem, Subst)$
$unify'\ p@(q, [\,]) = Just\ (p, [\,])$   -- case: trivial
$unify'\ p@(q, (e@(r, s) : es)) =$
  **if** $r \equiv s$   -- case: identity
  **then** $Just\ ((q, es), [\,])$
  **else case** $(r, s)$ **of**
    $(\lambda x\ r', \lambda y\ s') \to$
      **if** $x \not\equiv y$   -- case: xi
      **then** $Nothing$
      **else** $Just\ ((q', ((r', s') : es)), [\,])$
        **where** $q' = foldl\ (\cdot\overset{+}{\underset{\forall}{}})\ q\ x$
    $(f \bullet r', g \bullet s') \to$
      **if** $(f \in q_L \land g \in q_L) \lor (f \in q_\forall \land g \in q_\forall)$   -- case: rigid-rigid
      **then**
        **if** $f \not\equiv g$
        **then** $Nothing$
        **else**
          **let** $r's' = zip\ r'\ s'$
          **in** $Just\ ((q, (r's' +\!\!+\ es)), [\,])$
      **else**
        **if** $(f \in q_L \lor f \in q_\forall) \land g \in q_\exists$   -- case: rigid-flex
        **then** $Just\ ((q, ((swap\ e) : es)), [\,])$
        **else** $unify''\ p$   -- hand over to handle other cases.
    $otherwise \to unify''\ p$

$unify''\ (q, ((f \bullet r', g \bullet s') : es)) =$
  **if** $f \in q_\exists \land g \in q_\exists$   -- case: flex-flex
  **then**
    **if** $f \equiv g$
    **then**
      **let** $f' = \sharp_f(q_\exists)$
          $q' \ = (q_\exists^{-}\ f)_\exists^{+}\ f'$
          $\rho = [(f, \overline{(\lambda(\ulcorner r' \urcorner)\ (f' \bullet w))}^{\mathrm{Exp}})]$
          $w \ = r' \Cap s'$
      **in** $Just\ ((q', mapPair\ \overline{\rho}^{\mathrm{fun}}\ es), \rho)$
```

```
        **else**   -- with different heads
          **let** $f' = \sharp_f(q_\exists)$
            $q' \;= ((q_\exists^- \ f)_\exists^- \ g)_\exists^+ \ f'$
            $\rho = [(f, \overline{(\lambda(\ulcorner r' \urcorner) \ (f' \bullet w))}^{\mathrm{Exp}}), (g, \overline{(\lambda(\ulcorner s' \urcorner) \ (f' \bullet w))}^{\mathrm{Exp}})]$
            $w \;= r' \cap s'$
          **in** $\mathit{Just}\ ((q', \mathit{mapPair}\ \overline{\rho}^{\mathrm{fun}}\ es), \rho)$
      **else** $\mathit{unify}'''\ (q, ((f \bullet r', g \bullet s') : es))$
    $\mathit{unify}''\ p = \mathit{unify}'''\ p$
    $\mathit{unify}'''\ (q, ((f \bullet r', t) : es)) =$
      **if** $\neg\ (f \in q_\exists)$   -- case: flex-rigid
      **then** $\mathit{Nothing}$
      **else**
        **if** $f \subseteq_\Lambda t$                    -- occurs check
        **then** $\mathit{Nothing}$
        **else**
          **if** $\mathit{shouldPrune}\ q\ r'\ t$   -- pruning necessary
          **then case** $\mathit{canPrune}\ (q, t)\ [\,]$ **of**
            $\mathit{Nothing}$              $\rightarrow \mathit{Nothing}$   -- pruning failed
            $\mathit{Just}\ ([(v, e)], v') \rightarrow$
              **let** $q' = (q_\exists^-\ v)_\exists^+\ v'$
                $\rho = [(v, e)]$
              **in** $\mathit{Just}\ ((q', \mathit{mapPair}\ \overline{\rho}^{\mathrm{fun}}\ es), \rho)$
          **else**   -- explicit definition
            **let** $q' = q_\exists^-\ f$
              $\rho = [(f, \overline{(\lambda(\ulcorner r' \urcorner)\ t)}^{\mathrm{Exp}})]$
            **in** $\mathit{Just}\ ((q', \mathit{mapPair}\ \overline{\rho}^{\mathrm{fun}}\ es), \rho)$
    $\mathit{unify}'''\ \_ = \mathit{Nothing}$
```

5.4.1 Pruning

The flex-rigid case in the unification process may involve pruning away free – actually, *loose* – forbidden variables if they appear on only one side of an equation. Function *shouldPrune* tests whether pruning is needed, and *canPrune* attempts to prune. If the former is true but the latter fails then the unification process fails.

```
    $\mathit{shouldPrune} :: \mathit{QPrefix} \rightarrow [\mathit{FAE}] \rightarrow \mathit{FAE} \rightarrow \mathit{Bool}$
    $\mathit{shouldPrune}\ q\ l\ t = (\mathit{shouldPrune}'\ q\ l\ t) \not\equiv [\,]$   -- wraps around the next function.

    $\mathit{shouldPrune}' :: \mathit{QPrefix} \rightarrow [\mathit{FAE}] \rightarrow \mathit{FAE} \rightarrow [\mathit{String}]$   -- returns list of forbidden variables
    $\mathit{shouldPrune}'\ q\ l\ t =$ **let** $\mathit{frees} = \mathsf{FV}\ t$                    -- free in "t" that should be pruned.
                           $\mathit{forbs} = [x \mid x \leftarrow \mathit{frees}, x \in q_\forall]$
                           $\mathit{frees}' = \bigcup \mathsf{FV}\ \overrightarrow{l}$
                       **in** $\mathit{forbs} \setminus\setminus \mathit{frees}'$
```

The arguments expected by *shouldPrune'* are: the quantifier prefix – from which the forbidden variables are drawn, a list of operands from the left-hand-side of the (flex-rigid) equation, and the term on the right-hand-side. The function returns the forbidden variables that occur loose on the right-hand-side but not on the left. Pruning will act to remove this disparity, but this might not always be possible. The function described next accepts a term to prune and a list of externally-bound variables and, if pruning is possible, will return a substitution that will eliminate the disparity in loose forbidden variables. It also returns the next fresh variable name to be used by the rest of the unification process. The most interesting clause in this definition concerns redexes; the other cases serve to propagate the search for places where to apply pruning. Finally, function *propagPrune* propagates the search for subterms to prune through the operand-list.

```
canPrune :: QTerm → [String] → Maybe (Subst, String)
  -- look for a applicative term having shape X w1s z w2s
canPrune (q, ⅄vs e) bs = canPrune (q, e) (vs ++ bs)
canPrune (q, v • l) bs =
  if v ∈ $q_\exists$
  then   -- most important clause
    let isFreeForbIn _ [] = Nothing
      isFreeForbIn pre (z : zs) =
        if z ∈ $q_\forall$ ∧ ¬ (z ∈ bs)   -- “z” is forbidden and not bound.
        then Just (reverse pre, z, zs)
        else isFreeForbIn (z : pre) zs
        -- now search for an unbound forbidden variable in arguments to flexible “v”
      w1s_z_w2s = isFreeForbIn [] l
    in case w1s_z_w2s of
      Nothing → Nothing
      Just (w1s, z, w2s) →
        let v′ = $\sharp_v(q_\exists)$   -- generate fresh variable in “q” from “v”
          sub = ⅄(⌜(w1s ++ (z : w2s))⌝) (v′ • (w1s ++ w2s))
        in Just ([(v, $\overline{sub}^{\text{Exp}}$)], v′)   -- build part of pruning step; this is
              -- the productive part of this function,
              -- the rest is mostly searching.
  else propagPrune q l bs   -- propagate search to l
canPrune (q, e1 • e2s) bs =
  let p1 = canPrune (q, e1) bs   -- propagate the search
    p2 = propagPrune q e2s bs
  in case p1 of
    Nothing → p2
    ans   → ans
canPrune (q, v) bs = Nothing   -- there’s no flex variable at head
propagPrune q [] bs = Nothing   -- propagate the search for stuff to prune
propagPrune q (x : xs) bs =
  case (canPrune (q, x) bs) of
    Nothing → propagPrune q xs bs
    p → p
```

5.4.2 Auxiliary functions

The precise details of the following definitions are omitted since their implementations are straightforward. In the interest of readability the notation used in this report conceals the constructors used in *Exp* and *FAE* (see §2.1) for the formation of variables. Notwithstanding the absence of explicit markings, a distinction must be drawn between variables as terms and their names. Taking xs to range over lists of variables, $\ulcorner xs \urcorner$ will denote the list of names associated with each variable in xs – that is, it acts elementwise on xs to project a variable's name. Taking ns to range over lists of variable names, then $\ulcorner\ulcorner ns \urcorner\urcorner$ denotes the list of variables bearing those names. The symbol $\subseteq_\Lambda$ will be used to denote the subterm relation, and the function $\sharp_{ns} n$ produces the variable name closest to n which is fresh relative to the elements in ns. Finally, $\overline{\rho}^{\text{fun}}$ denotes the function produced from the graph ρ.

5.5 Main function

Function *unify* combines the behaviour of earlier definitions and iterates the unification algorithm until it yields a solution or fails to find one.

```
unify :: UProblem → Maybe (QPrefix, Subst)
unify p@(q, e) =
  case (unify′ (p ⇓)) of    -- note: “normalise” compacts its result.
    Nothing → Nothing
    Just (p′, ρ) →
      if p ≡ p′
      then Just (fst p′, ρ)
      else case (unify p′) of
        Nothing → Nothing
        Just (q′, φ′) → Just (q′, (ρ ⨾ φ′) ↾ ($q_∃$))
```

5.6 Examples

Some of the following examples are drawn from the article by Nipkow (1993). We will start with elementary tests and proceed to test more of the functionality implemented above.

```
qpref = foldl ($\cdot^{+}_{\exists}$) empty [Y, X]
test1 = unify (qpref, [(X, Y)])
test1′ = unify (qpref, [(X • [ ], Y • [ ])])
test1″ = unify (qpref, [(λλ[ ] X, λλ[ ] Y)])
```

The evaluation of *test1*, *test1'*, and *test1"* yields the same value:

```
Just (
Signatures : (none)
Flexibles   : X′
Forbiddens : (none)
, [(X, X′), (Y, X′)])
```

The pattern-checking function is tested next.

```
p1 = (q, (λλ[x] (F • [(λλ[z] (x • [z]))])))
  where q = empty$^{+}_{\exists}$ F
p1short = (q, (λλ[x] (F • [x])))
  where q = empty$^{+}_{\exists}$ F
np4 = (q, (λλ[x] (G • [H])))   -- not a pattern
  where q = (foldl) $\cdot^{+}_{\exists}$ empty [G, H]
p3 = (q, (λλ[x] (c • [x])))
  where q = empty$^{+}_{L}$ c
np5 = (q, (λλ[x] (c • [Z])))   -- not a pattern
  where q = empty$^{+}_{L}$ c
p4 = (q, (λλ[x, y] (F • [x, y])))
  where q = empty$^{+}_{\exists}$ F
np1 = (q, (F • [c]))
  where q = (empty$^{+}_{\exists}$ F)$^{+}_{L}$ c
np2 = (q, (λλ[x] (F • [x, x])))
  where q = empty$^{+}_{\exists}$ F
```

$np3 = (q, (\lambda\!\!\lambda[\mathtt{x}]\ (\mathtt{F} \bullet [(\mathtt{F} \bullet [\mathtt{x}])])))$
  where $q = empty^{+}_{\exists}\ \mathtt{F}$

The previous definitions whose names do not begin in *np* are patterns. More interesting behaviour is tested next. Definition *pr1* is not a pattern unification problem, and as a result the algorithm misbehaves since it was not presented with valid input.

$pr1 = (q, [(snd\ p1, snd\ np4)])$ -- this is NOT a pattern unification problem
  where $q = foldl\ (\cdot^{+}_{\exists})\ empty\ [\mathtt{F}, \mathtt{G}, \mathtt{H}]$

The next attempts are all successful and test different parts of the algorithm.

$pr2 = ((q, [(\lambda\!\!\lambda[\mathtt{x}, \mathtt{y}]\ (\mathtt{F} \bullet [\mathtt{x}]), \lambda\!\!\lambda[\mathtt{x}, \mathtt{y}]\ (\mathtt{c} \bullet [\mathtt{G} \bullet [\mathtt{y}, \mathtt{x}]]))]) :: UProblem)\ \dagger$ -- flex-rigid
  where $q = (foldl\ (\cdot^{+}_{\exists})\ empty\ [\mathtt{F}, \mathtt{G}])^{+}_{L}\ \mathtt{c}$

unify pr2
$\rightsquigarrow$ *Just* (
Signatures : *c*
Flexibles : *G′ and F*
Forbiddens : *a‴ and a′*
, $[(\mathtt{G}, \lambda\!\!\lambda\mathtt{a}'''\ (\lambda\!\!\lambda\mathtt{a}'\ (\mathtt{G}' \bullet \mathtt{a}')))])$

$pr3 = ((q, [(\lambda\!\!\lambda[\mathtt{x}, \mathtt{y}]\ (\mathtt{F} \bullet [\mathtt{x}]), \lambda\!\!\lambda[\mathtt{x}, \mathtt{y}]\ (\mathtt{G} \bullet [\mathtt{y}, \mathtt{x}]))]) :: UProblem)\ \dagger$ -- flex-flex
  where $q = foldl\ (\cdot^{+}_{\exists})\ empty\ [\mathtt{F}, \mathtt{G}]$

unify pr3
$\rightsquigarrow$ *Just* (
Signatures : (*none*)
Flexibles : *F′*
Forbiddens : *a‴ and a′*
, $[(\mathtt{F}, \lambda\!\!\lambda\mathtt{a}'\ (\mathtt{F}' \bullet \mathtt{a}')), (\mathtt{G}, \lambda\!\!\lambda\mathtt{a}'''\ (\lambda\!\!\lambda\mathtt{a}'\ (\mathtt{F}' \bullet \mathtt{a}')))])$

$pr4 = (q, [(\mathtt{x}, \mathtt{X} \bullet [\mathtt{x}])])\ \dagger :: UProblem$
  where $q = (empty^{+}_{\exists}\ \mathtt{X})^{+}_{\forall}\ \mathtt{x}$

unify pr4
$\rightsquigarrow$ *Just* (
Signatures : (*none*)
Flexibles : (*none*)
Forbiddens : *x*
, $[(\mathtt{X}, \lambda\!\!\lambda\mathtt{x}\ \mathtt{x})])$

Notes

In §5.3 the combinator *foldR* was used; curiously an equivalent of this general combinator could not be found in the standard libraries.

Chapter 6

Proof system

An automatic theorem prover can now be built by combining the components from earlier chapters.

6.1 Supporting definitions

The language of formulas will be described next and shown to be term-like. As in the previous chapter, formulas will then be paired with a $\forall^*\exists^*\forall^*$ quantifier prefix to to yield *Q-formulas.* The theorem prover works over a fragment of Q-formulas formed by *clause* and *goal* Q-formulas; this fragment is characterised in §6.1.2.

```
data Formula
    = Formula ⟶ Formula    -- implication
    | ∀[String] Formula     -- univ.quantification
    | String⟨[FAE]⟩          -- predicate
instance Term Formula where
    $\mathsf{FV}\ (\_\langle f\rangle) = \bigcup \mathsf{FV}\ \overrightarrow{f}$
    $\mathsf{FV}\ (\forall xs\ f) = (\mathsf{FV}\ f) \setminus\setminus xs$
    $\mathsf{FV}\ (f1 \longrightarrow f2) = (\mathsf{FV}\ f1) +\!\!+ (\mathsf{FV}\ f2)$

    $\mathsf{BV}\ (\_\langle f\rangle) = \bigcup \mathsf{BV}\ \overrightarrow{f}$
    $\mathsf{BV}\ (\forall xs\ f) = (\mathsf{BV}\ f) +\!\!+ xs$
    $\mathsf{BV}\ (f1 \longrightarrow f2) = (\mathsf{BV}\ f1) +\!\!+ (\mathsf{BV}\ f2)$

    $(n\langle f\rangle)\,\dagger = n\langle(\cdot\dagger)\ \overrightarrow{f}\rangle$
    $(\forall xs\ f)\,\dagger = \forall xs\ (f\,\dagger)$
    $(f1 \longrightarrow f2)\,\dagger = (f1\,\dagger) \longrightarrow (f2\,\dagger)$

    $(n\langle f\rangle)\,\ddagger = n\langle(\cdot\ddagger)\ \overrightarrow{f}\rangle$
    $(\forall xs\ f)\,\ddagger = \forall xs\ (f\,\ddagger)$
    $(f1 \longrightarrow f2)\,\ddagger = (f1\,\ddagger) \longrightarrow (f2\,\ddagger)$

    $(n\langle f\rangle)\Downarrow = n\langle(\cdot \Downarrow)\ \overrightarrow{f}\rangle$
    $(\forall xs\ f)\Downarrow = \forall xs\ (f \Downarrow)$
    $(f1 \longrightarrow f2)\Downarrow = (f1 \Downarrow) \longrightarrow (f2 \Downarrow)$
type QFormula = (QPrefix, Formula)
```

The theorem prover will operate on sequents, which will be defined next. The antecedent formulas in a sequent are accompanied by a name and a number: the name serves to identify the assumption variable associated with the assumed formula, and the number is the multiplicity of the assumption – the maximum number of times the assumption may be used – and serves to control the complexity of resulting proofs. The type of sequents are instantiated in the class of terms as shown below.

```
data Sequent = [(String, Formula, Int)] ⟹ Formula

instance Term Sequent where
  FV (p ⟹ f) = let pForms = (λ(_, y, _) → y) $\overrightarrow{p}$
               in nub ((⋃ FV $\overrightarrow{pForms}$) ++ FV f)
  BV (p ⟹ f) = let pForms = (λ(_, y, _) → y) $\overrightarrow{p}$
               in nub ((⋃ BV $\overrightarrow{pForms}$) ++ BV f)
  (p ⟹ f) †  = (mapTriple2 (·†) p) ⟹ (f†)
  (p ⟹ f) ‡  = (mapTriple2 (·‡) p) ⟹ (f‡)
  (p ⟹ f) ⇓  = (mapTriple2 (· ⇓) p) ⟹ (f ⇓)
```

If a formula in this fragment is provable, the theorem prover will return a proof witnessing this fact as a term.

```
type Proof = Exp
```

6.1.1 Lifting combinators

In the previous chapter, *FAE* was favoured over *Exp* as a representation for λ-terms. In this chapter both will be used, albeit in different scopes: *Exp* is used to represent proofs, and *FAE* is more wieldy to represent arguments to predicates. The function *liftExpForm* lifts functions over terms – in *Exp* notation – to be propagated to terms within formulas. This is used, for example, to lift substitutions on terms to operate on terms within formulas. Similarly, *liftFrmtoSeq* lifts functions defined over formulas to operate on sequents.

6.1.2 Syntactic checks

The definitions in this section characterise clause and goal Q-formulas, and Q-sequents. An informal description will be given in place of precise definitions here, which are adapted from Schwichtenberg (2004).

- If $\vec{r}$ are Q-terms then $P\vec{r}$ is both a Q-clause and Q-goal.
- If D is a Q-clause and G a Q-goal, then $D \longrightarrow G$ is a Q-goal.
- If G is a Q-goal and D a Q-clause, then $G \longrightarrow D$ is a Q-clause.
- If G is a $\mathrm{Q}_\forall^+ x$-goal, then $\forall x G$ is a Q-goal.
- If $D\{y \mapsto Y \bullet Q_\forall\}$ is a $\mathrm{Q}_\exists^+ Y$-clause then $\forall y D$ is a Q-goal.

6.1.3 Harrop normal form

For the purposes of resolution it is useful to represent antecedent formulas as $\forall \vec{x}.\vec{f} \longrightarrow g$, called an *elab-list* in the Minlog implementation and *Harrop normal form* (HNF) in Isabelle (Berghofer 2003, §2.3.1). This representation induces a structural uniformity in formulas, as a result of which checking whether the formula's head (conclusion) and the sequent's conclusion are resolvable. If the conclusions can be resolved then $\vec{f}$ are added as subgoals to the proof process. Formulas in HNF are formalised as values of the type *Reformula* below. The functions *reformulate* and *unreformulate* are defined to handle conversions of formulas to and from values of *Reformula* respectively; their definitions are straightforward and are omitted here.

```
type Reformula = ([String], [Formula], Formula)
```

6.2 Theorem prover

The proof system consists of two inference rules: *strip* and *resolve*. The proving machinery applies these automatically by iterating a process that checks the shape of the formula then applies one of these two rules, and fails if the inference step fails. Function *prove* accepts a Q-sequent and if the search is successful it returns the final Q-sequent, a proof and the overall substitution function computed.

```
prove :: Int → QSequent → IO (Maybe (QPrefix, Subst, Proof, Sequent))
prove cnt (q, s) = do
  putStr (=== Step ++ (show cnt) ++ \n Goal : ++ (show s) ++ \n)
  case s of
    p ⟹ ∀xs f →
      case (strip cnt (q, s)) of
        Nothing → do
          putStr (Failed to strip variable(s).\n)
          return Nothing
        Just (q′, s′, pf, ρ) → do
          putStr (Stripped variable(s).\n)
          continuance ← prove (cnt + 1) (q′, s′)
          case continuance of    -- do rest of the proof
            Nothing → do
              putStr (Failed step ++ (show cnt) ++ .\n)
              return Nothing
            Just (q2, ϕ′, pf_rest, s″) →
              let
                  $\phi = (\rho \mathbin{⨾} \phi') \upharpoonright (q'_\exists)$
                  goal = appSubstSequ $\phi$ s″
              in do
                  putStr (Completed step ++ (show cnt) ++ .\n)
                  return $ Just (q2, $\phi$, $\overline{\overline{\phi}^{\text{fun}}\ (\overline{(pf\ pf\_rest)}^{\text{FAE}})}^{\text{Exp}}$, goal)
    p ⟹ (f ⟶ f′) →
      case (strip cnt (q, s)) of
        Nothing → do
          putStr (Failed to strip implication.\n)
          return Nothing
        Just (q′, s′, pf, ρ) → do
          putStr (Stripped implication.\n)
          continuance ← prove (cnt + 1) (q′, s′)
          case continuance of    -- do rest of the proof
            Nothing → do
              putStr (Failed step ++ (show cnt) ++ .\n)
              return Nothing
            Just (q2, ϕ′, pf_rest, s″) →
              let
                  $\phi = (\rho \mathbin{⨾} \phi') \upharpoonright (q'_\exists)$
                  goal = appSubstSequ $\phi$ s″
              in do
                  putStr (Completed step ++ (show cnt) ++ .\n)
                  return $ Just (q2, $\phi$, $\overline{\overline{\phi}^{\text{fun}}\ (\overline{(pf\ pf\_rest)}^{\text{FAE}})}^{\text{Exp}}$, goal)
```

```
$p \Longrightarrow n\langle ts\rangle \rightarrow$
  case ($resolve\ cnt\ (q, s)$) of
    $[\,] \rightarrow$ do
      $putStr$ (Failed to resolve.\n)
      $return\ Nothing$
    $list \rightarrow$
      let
          $ploop\ [\,] = return\ [\,]$   -- iterate through possible paths until reach one that leads to the proof.
          $ploop\ (x@(q, ss, pf, \_, \_) : xs) =$
              let
                $loop\ [\,]\ \_ = return\ [\,]$   -- no subgoals
                $loop\ (x : xs)\ q =$ do
                  $continuance \leftarrow prove\ (cnt + 1)\ (q, x)$
                  case $continuance$ of    -- do rest of the proof
                    $Nothing \rightarrow$
                      $return\ [Nothing]$   -- abort the proof attempt, disregard other subgoals.
                    $ans@(Just\ (q', \rho, pf', \_)) \rightarrow$
                      let
                        $xs' = (appSubstSequ\ \rho)\ \overrightarrow{xs}$
                      in if $xs \equiv [\,]$
                        then $return\ [ans]$
                        else do
                          $rest \leftarrow loop\ xs'\ q'$
                          $return\ (ans : rest)$
                $theRest\ y =$ if $\neg\ (Nothing \in y)$
                        then $return\ [(x, y)]$   -- don't attempt the other paths,
                                                  -- return the first that succeeds.
                        else $ploop\ xs$
              in do
                $thisGoal \leftarrow loop\ ss\ q$
                $theRest\ thisGoal$
          $attempt = ploop\ list$
      in do
          $putStr$ (Resolvable with $+\!\!+$ ($andList$ (($\lambda(\_, \_, \_, \_, u) \rightarrow u$) $\overrightarrow{list}$)) $+\!\!+$ \n)
            -- try each resolvant in turn, until we reach one for which all subgoals are provable.
          $attemptResult \leftarrow attempt$
          case $attemptResult$ of
            $[\,] \rightarrow$ do
              $putStr$ (Failed step $+\!\!+$ ($show\ cnt$) $+\!\!+$ .\n)
              $return\ Nothing$   -- none of the proof attempts succeeded.
            $\_ \rightarrow$
              let
                $focus = head\ attemptResult$
                $(q, ss, pf, subs, \_) = fst\ focus$   -- original goal information
                $answer\ x =$
                  let
                    $result = catMaybes\ x$   -- focus on first successul attempt
                  in
```

```
        if (result ≡ [])
        then
          if (ss ≢ [])
          then error Goal mismatch!\n
          else   -- apply preproof to empty subproof list
            Just (q, subs, pf [], appSubstSequ subs s)
        else
          let
            finalStep = last result
              -- finalStep contains the final state and final substitution
              -- that we're to use. Now concentrate on building proof term.
            $(q', \phi', \_, p' \Longrightarrow \_) = finalStep$
            $proofs = (\lambda(\_, \_, pf\_i, \_) \rightarrow pf\_i)\ \overrightarrow{result}$   -- project out the list of proofs
            $\phi = subs \restriction (q'_{\exists}) ⨾ \phi'$
          in $Just\ (q', \phi, \overline{(\overline{\phi}^{\text{fun}}\ (\overline{(pf\ \ proofs)}^{\text{FAE}}))}^{\text{Exp}}, appSubstSequ\ \phi\ (p' \Longrightarrow (n\langle ts\rangle)))$
  in do
    putStr (Completed step ++ (show cnt) ++ .\n)
    return (answer (snd focus))
```

6.2.1 Strip

The rule *strip* fuses the introduction rules for quantification and implication and could be illustrated as shown below. The inference forms part of a proof tree that is rooted at the top. Preceding the inference is its context on the left of the bar, and following the inference is the new context on the left of the bar and the proof sought on the right. Note that the line following the inference is only valid if the rest of proof is possible – so in this sense this illustration is unlike standard descriptions of inference rules. The rest of the proof process serves to define the proof term M and the substitution ϕ'. The symbol ϕ abbreviates $(\rho ⨾ \phi') \restriction Q_{\exists}$; additional symbols are explained in the column on the right of the rule.

$$\frac{\begin{array}{c} Q, \Lambda \qquad | \\ \forall\vec{x}.\vec{D} \longrightarrow A \end{array}}{\begin{array}{c} A\rho \\ Q^{+}_{\forall}\vec{y}, \Lambda\lambda\vec{y}\lambda\vec{u}^{\vec{D}\phi} \,| \qquad M^{A\phi} \end{array}}$$

The $\vec{u}$ are fresh assumption variables that are associated with $\vec{D}$ ordinate-wise. The list $\vec{y}$ abbreviates $\vec{x}\rho$, where ρ is a renaming that avoids shadowing in Λ, which would lead to variable capture in M.

The rule above is automatically applied to whittle a goal while gradually building its proof term. The rest of the proof process attempts to define M. The precise definition of the inference rule is presented next.

```
strip :: Int → (QPrefix, Sequent) → Maybe (QPrefix, Sequent, Exp → Proof, Subst)
  -- return new prefix, additional proof goals, a function to build proof (when the new proof goals have
  -- been dischanged), and a substitution in case any bound variables need renaming.
strip cnt (q, p ⟹ f) =
  case f of
    ∀xs f →
      let
        -- now ensure that xs is disjoint from all names in use so far
      xs' =
        let
        $namesSoFar =$
          **let**
            $assumpVars = (\lambda(x, \_, \_) \rightarrow x)\ \overrightarrow{p}$
            $prefixVars = q_L ⧺ q_\exists ⧺ q_\forall$
          **in** $assumpVars ⧺ prefixVars$
        **in** $freshen\ xs\ namesSoFar$
      $renaming =$
        **let**
          $ren1 = zip\ xs\ (\overline{\cdot}^{\mathrm{Exp}}\ \overrightarrow{(\ulcorner xs' \urcorner)})$
          $loop\ [\,] = [\,]$
          $loop\ (this@(x, y) : rest) =$
            **if** $x \equiv y$   -- remove (x,Var x) substitutions
            **then** $loop\ rest$
            **else** $this : (loop\ rest)$
        **in** $loop\ ren1$
        -- forbid the variables
      $q' = foldl\ (\cdot^{+}_{\forall})\ q\ xs'$
        -- produce subgoal
      **in** $Just\ (q', (p \Longrightarrow appSubstForm\ renaming\ f), \lambda t \rightarrow \overline{(\lambda\!\!\lambda xs'\ (\overline{t}^{\mathrm{FAE}}))}^{\mathrm{Exp}}, renaming)$
    $f1 \longrightarrow f2 \rightarrow$
      **let**
        $u' =$   -- come up with fresh name for this assumption
          **let**
            $assumpVars = (\lambda(x, \_, \_) \rightarrow x)\ \overrightarrow{p}$
          **in** $head\ \$\ freshen\ [\mathtt{u}]\ assumpVars$
        $p' =$   -- check if f1's already in "p", if so then increment its availability otherwise add it.
          **case** $lookup\ f1\ ((\lambda(x, y, z) \rightarrow (y, (x, z)))\ \overrightarrow{p})$ **of**
            $Nothing \rightarrow (u', f1, 2) : p$
            $Just\ (u, n) \rightarrow replace\ (u, f1, n)\ (u, f1, n + 2)\ p$
      **in** $Just\ (q, p' \Longrightarrow f2, \lambda t \rightarrow \lambda\!\!\lambda u'\ t, [\,])$
```

6.2.2 Resolve

The rule *resolve* fuses the elimination rules for quantification and implication. In the illustration below, prior to the inference being made, a suitable assumption needs to be chosen to be resolved with the current proof goal. Potentially, there might be more than one plausible candidates for u: in such a case the candidates are tried in turn until a proof is found in depth-first-search fashion.

$$\frac{Q, \Lambda \ni u^{\forall \vec{x}.\vec{G} \longrightarrow P\vec{s}} \Big| \; P\vec{r}}{Q', \Lambda^{(\rho ⨟ \phi') \restriction Q'} \;\overset{\vec{G}^{*}\rho}{\Big|}\; u \bullet \vec{x}^{*}\rho\phi' \bullet \vec{M}^{\vec{G}^{*}\rho\phi'}}$$

The $\vec{M}$ are proofs of $\vec{G}^{*}\rho\phi'$ ordinate-wise, and $\vec{X}$ are fresh variables equinumerous to $\vec{x}$. The map $\cdot^{*}$ is $x_i \mapsto X_i \bullet Q_\forall$. Crucial to this inference step is the result of unification $(Q', \rho) :=$ unify $(Q^{+}_{\exists}\vec{X})\ (\vec{r}, \vec{s}^{*})$

The subterm $\vec{x}^{*}\rho\phi'$ is equivalent to $((\vec{X}\rho\phi') \bullet Q_\forall)$ and serves to eliminate $\forall\vec{x}$ in u. The $\vec{M}$ then discharge the assumptions $\vec{G}$ in u.

```
$resolve :: Int \to (QPrefix, Sequent) \to [(QPrefix, [Sequent], [Exp] \to Proof, Subst, String)]$
  -- Last component of function's result is the assumption variable of the formula resolved with.
$resolve\ cnt\ (q, p \Longrightarrow n\langle rs\rangle) =$
  let
    $resolvants = tryMatching\ (q, p \Longrightarrow n\langle rs\rangle)$
    $loop\ [\,] = [\,]$
    $loop\ ((f, f'@(xs, gs, c), p', (q', \rho), allElims) : rest) =$
      $(q', ss', proof, \rho, u) : (loop\ rest)$
      where
        $u =$ let
            $a's = (\lambda(x, y, \_) \to (y, x))\ \overrightarrow{p'}$
          in
            case $lookup\ f\ a's$ of
              $Just\ identifier \to identifier$
              $Nothing \to$    -- This should never be the case.
                $error$ (Could not find assumption $+\!\!+$ $(show\ f')$ $+\!\!+$ \nin $+\!\!+$ $(show\ a's)$ $+\!\!+$ \n)
        $ss' = (appSubstSequ\ \rho)\ \overrightarrow{((\lambda g \to p' \Longrightarrow g)\ \overrightarrow{gs})}$   -- new subgoals
        $proof =$   -- feed it subgoals' proofs to obtain proof of original goal.
          $\lambda x \to \overline{((u \bullet allElims) \bullet (\overline{\cdot}^{\mathrm{FAE}}\ \overrightarrow{x}))}^{\mathrm{Exp}}$
  in $loop\ resolvants$
```

Finding resolvants

The function *tryMatching* sifts the antecedents in a sequent and returns a list of resolvable candidates. This process is organised into two stages:

1. Antecedents which do not conclude with the goal predicate are filtered off;
2. The goal's arguments and those of the antecedent's conclusion are unified. If this succeeds then the outcome is retained, otherwise the resolution candidate is dropped.

In effect this function returns a list of possible paths to follow in search of the proof. These are tried in turn; when all the sub-paths leading from a path do not lead to a proof then the system chronologically backtracks and searches along the next path.

```
$tryMatching :: (QPrefix, Sequent) \to [(Formula, Reformula, [(String, Formula, Int)], (QPrefix, Subst), [FAE])]$
$tryMatching\ (q, p \Longrightarrow n\langle rs\rangle) =$
  let
    $loop\ [\,] = [\,]$
    $loop\ ((u, f, cnt) : xs) =$
      if $cnt > 0$   -- heed multiplicity of use of assumption
      then
        let $f'@(vars, ante, conc) = reformulate\ f$
        in case $conc$ of
          $(m\langle ts'\rangle) \to$
            if $(n \equiv m) \wedge (length\ rs) \equiv (length\ ts')$
            then
              let
                $p' =$   -- decrement availability of this assumption
                  $replace\ (u, f, cnt)\ (u, f, cnt - 1)\ p$
              in
                $(f, f', p', (q, (rs, ts'))) : (loop\ xs)$
            else $loop\ xs$
          $\_ \rightarrow loop\ xs$
      else $loop\ xs$
    $filter1 = loop\ p$   -- first round of checks

    $prop =$
        -- this definition shows there is lots of processing to do before calling unify
      $\lambda(f, f'@(xs, fs, c), ante, p@(q, (rs, ts))) \rightarrow$
        let
          $namesSoFar =$
            let
              $assumpVars = (\lambda(x, \_, \_) \rightarrow x)\ \overrightarrow{ante}$
              $prefixVars = q_L +\!\!+\ q_\exists +\!\!+\ q_\forall$
            in $assumpVars +\!\!+\ prefixVars$
          $xs' = freshen\ xs\ namesSoFar$
          $renaming =$
            let $ren1 = zip\ xs\ xs'$
              $loop\ [\,] = [\,]$
              $loop\ ((x, y) : rest) =$
                if $x \equiv y$   -- remove (x,Var x) substitutions
                then $loop\ rest$
                else $(x, y) : (loop\ rest)$
            in $loop\ ren1$
          $fs' = (appSubstForm\ renaming)\ \overrightarrow{fs}$
          $c'\ = appSubstForm\ renaming\ c$
            -- up to now have ensured that variable names occurring resolvant and in
            -- current goal are disjoint. The unification problem is prepared next;
            -- now we must synch “ts” by applying the renaming defined previously.
          $ts' = \overline{renaming}^{\mathrm{fun}}\ \overrightarrow{ts}$
          $namesSoFar' = namesSoFar +\!\!+\ xs'$
            -- and now for the processing prescribed in the paper.
          $zs = \ulcorner\!\ulcorner(q_\forall)\urcorner\!\urcorner$
          $xS = freshen\ ((toUpper\ \overrightarrow{\cdot})\ \overrightarrow{xs'})\ namesSoFar'$   -- fresh names for raised variables
          $qStar = foldl\ (\cdot^{+}_{\exists})\ q\ xS$
          $valueTerms = (\lambda x' \rightarrow (x' \bullet zs))\ \overrightarrow{xS}$   -- value terms of the mapping defined next
          $star = (\lambda(x, x') \rightarrow (x, \overline{x'}^{\mathrm{Exp}}))\ \overrightarrow{(zip\ xs'\ valueTerms)}$
          $tsStar = \overline{star}^{\mathrm{fun}}\ \overrightarrow{ts'}$
          $gStar = (appSubstForm\ star)\ \overrightarrow{fs'}$
          $c'' = appSubstForm\ star\ c'$
          $unifProb = zip\ ((\cdot\dagger)\ \overrightarrow{rs})\ ((\cdot\dagger)\ \overrightarrow{tsStar})$
          $p' = unify\ (qStar, unifProb)$   -- try to solve the unification problem built until now.
        in case $p'$ of
          $Nothing \rightarrow Nothing$
          $Just\ (q', \rho) \rightarrow$
            let
```

```
            $f' = reformulate\ (appSubstForm\ \rho\ (unreformulate\ (xs', gStar, c'')))$
            $allElims =$    -- this is used in building the proof term, in elimination of forall
              $\overline{(renaming \fatsemi \rho)}^{\text{fun}}\ \overrightarrow{valueTerms}$
          **in** $Just\ (f, f', ante, (q', renaming \mathbin{+\!\!+} \rho), allElims)$
    $filter2 = catMaybes\ (prop\ \overrightarrow{filter1})$
  **in** $filter2$
```

6.3 Auxilliary functions

The combinators *appSubstForm* and *appSubstSequ* are defined to apply substitutions to formulas and sequents respectively. The function *freshen* ensures that two lists of names are disjoint by replacing common names in the second list with fresh names. The precise implementation details are omitted here.

6.4 Main function

The key function in this implementation is *search* and it starts the proof process after being provided with an initial quantifier prefix (usually *empty*) and a formula. Examples of output produced are provided in Appendix A.

The theorem prover produces output in two stages:

1. During the proof search process the prover reports each inference step it makes, the goal it is tackling, and whether the inference is successful. This trace serves to explain how a proof was obtained, or why none was found.
2. If a proof has been found then the prover outputs both contexts – i.e., assumption and quantifier prefix contexts – and the proof term.

Chapter 7

Conclusion

The small bits of machinery described in each chapter were combined to form the theorem prover, of which some example runs are documented in Appendix A. The laminar organisation of the implementation emphasises the interaction between components, and also renders them individually reusable. Using Haskell as the implementation language has paid off primarily in the clarity of the implementation. This fulfilled the intention to follow Schwichtenberg's specification closely, even in its presentation.

The implementation's efficiency has not yet been guaged, nor has it been proved correct; these are suggested as future work. Transforming between expression representations is sure to hinder performance, so committing to a single representation is suggested as an improvement. The presentation suffers from the bureaucratic details describing the handling of variables and ensuring the generation of fresh ones. Although of small interest, these details are crucial to ensure correctness; a more elegant solution to this would be desirable.

This is the second implementation of the algorithm described in Schwichtenberg (2004), as the first implementation was done in Scheme and forms part of Minlog. Minlog's implementation is more stringent since it allows the user to state types for object variables; it associates arities with predicates; and it also checks whether a unification problem is a pattern unification problem, and whether a formula is a goal or a clause formula. These checks formalise the side-conditions on which the algorithm's correctness is predicated on. However, in the event of invalid input it is more likely to get unification failure rather than an unsound result, by the implementation of the unification algorithm.

This work can be extended in the following ways to match the implementation in Minlog. By implementing the checks mentioned above the proof process can reject invalid input early – this would require modifying the context's definition to associate types with object variables – or, for valid formulas, switch to using Huet's unification algorithm when it encounters non-patterns. Furthermore, the logic could be extended to support conjunction and the strong existential quantifier, as explained by Schwichtenberg (2004).

Schwichtenberg suggested modifying the algorithm to produce proofs devoid of loose variables. In Minlog, canonical type inhabitants are identified upon defining a type, and these are referred to in manual proofs – appearing in the place of flexible variables in automatically-found proofs. Other extensions include extending the configurability of the prover – regarding trace output and setting multiplicity – and interfacing it with other systems.

Acknowledgements

I thank Prof. Schwichtenberg for his guidance during my stay at LMU. I have also benefited from feedback during two seminars on this subject given last April. The talk given by Andreas Abel on the use of monads in logic implementations was instructive. My stay in Munich was made possible through a Marie Curie fellowship under the auspices of the MATHLOGAPS programme, which I gratefully acknowledge.

Appendix A

Examples

The definitions that follow encode some useful abbreviations. Following these are examples of the theorem prover's use.

A.1 Definitions

The symbol $\perp$ is defined to be the predicate reserved to denote logical falsity. The symbols $\exists^{\text{cl}}$ and *stab* denote the classical existence and stability schemes respectively.

$\perp = \texttt{bottom}\langle[\,]\rangle$

$\exists^{\text{cl}} xs\ f = (\forall xs\ (f \longrightarrow \perp)) \longrightarrow \perp$

$\mathit{stab}\ \mathit{pred}\ xs = \forall xs\ (((\mathit{pred}\ xs \longrightarrow \perp) \longrightarrow \perp) \longrightarrow \mathit{pred}\ xs)$

A.2 Example 1

The first example is a minimally-valid classical statement.

example1 =

$\mathit{search}\ \mathit{empty}\ ((\forall[\texttt{x}]\ (\texttt{Q}\langle[\texttt{x}]\rangle) \longrightarrow (\exists^{\text{cl}}[\texttt{x}]\ (\texttt{Q}\langle[\texttt{x}]\rangle))))$

The output produced by the algorithm follows:

```
===Step 0
 Goal:
:-        (all x. Q x) -> (all x. Q x -> bottom) -> bottom
Stripped implication.
===Step 1
 Goal:
u       : (all x. Q x)
:-        (all x. Q x -> bottom) -> bottom
Stripped implication.
===Step 2
 Goal:
u'      : (all x. Q x -> bottom)
u       : (all x. Q x)
:-        bottom
Resolvable with u'
===Step 3
```

```
 Goal:
u'      : (all x. Q x -> bottom)
u       : (all x. Q x)
:-        Q X
Resolvable with u
Completed step 3.
Completed step 2.
Completed step 1.
Completed step 0.
====] Proved [==================
Final sequent was:
u'      : (all x. Q x -> bottom)
u       : (all x. Q x)
:-        bottom

Final context was:
Signatures      : (none)
Flexibles       : X''
Forbiddens      : (none)

Proof   : \u\u'((u' X'') (u X''))
```

A.3 Example 2

Two slightly different formulas are tested in this example: one is not provable while the other is.

```
example2a =
  search empty (f1 ⟶ f2 ⟶ (Q⟨[]⟩))
  where
    f1 = ∀[y] ((∀[z] (R⟨[y,z]⟩)) ⟶ Q⟨[]⟩)
    f2 = ∀[y1] (R⟨[y1,y1]⟩)
example2b =
  search empty (f1 ⟶ f2 ⟶ (Q⟨[]⟩))
  where
    f1 = ∀[y] ((∀[z] (R⟨[y,z]⟩)) ⟶ Q⟨[]⟩)
    f2 = ∀[y1,y2] (R⟨[y1,y2]⟩)
```

The output from Example 2A follows:

```
===Step 0
 Goal:
:-        (all y. (all z. R y z) -> Q) -> (all y1. R y1 y1) -> Q
Stripped implication.
===Step 1
 Goal:
u       : (all y. (all z. R y z) -> Q)
:-        (all y1. R y1 y1) -> Q
Stripped implication.
===Step 2
 Goal:
u'      : (all y1. R y1 y1)
```

```
u       : (all y. (all z. R y z) -> Q)
:-       Q
Resolvable with u
===Step 3
 Goal:
u'      : (all y1. R y1 y1)
u       : (all y. (all z. R y z) -> Q)
:-       (all z. R Y z)
Stripped variable(s).
===Step 4
 Goal:
u'      : (all y1. R y1 y1)
u       : (all y. (all z. R y z) -> Q)
:-       R Y z
Failed to resolve.
Failed step 3.
Failed step 2.
Failed step 1.
Failed step 0.
Could not find a proof.
```

Example 2B leads to a positive result as shown next:

```
===Step 0
 Goal:
:-       (all y. (all z. R y z) -> Q) -> (all y1,y2. R y1 y2) -> Q
Stripped implication.
===Step 1
 Goal:
u       : (all y. (all z. R y z) -> Q)
:-       (all y1,y2. R y1 y2) -> Q
Stripped implication.
===Step 2
 Goal:
u'      : (all y1,y2. R y1 y2)
u       : (all y. (all z. R y z) -> Q)
:-       Q
Resolvable with u
===Step 3
 Goal:
u'      : (all y1,y2. R y1 y2)
u       : (all y. (all z. R y z) -> Q)
:-       (all z. R Y z)
Stripped variable(s).
===Step 4
 Goal:
u'      : (all y1,y2. R y1 y2)
u       : (all y. (all z. R y z) -> Q)
:-       R Y z
```

```
Resolvable with u'
Completed step 4.
Completed step 3.
Completed step 2.
Completed step 1.
Completed step 0.
====] Proved [==================
Final sequent was:
u'       : (all y1,y2. R y1 y2)
u        : (all y. (all z. R y z) -> Q)
:-         Q

Final context was:
Signatures      : (none)
Flexibles       : Y'
Forbiddens      : z

Proof   : \u\u'((u Y') \z((u' (\zY' z)) (\zz z)))
```

A.4 Example 3

The "drinker's problem" formula is proved next. This is not minimally-valid hence requires use of the stability axiom. Informally the formula expresses that "there is a person such that when that person drinks then everybody drinks".

```
exampleD =
  search empty (($stab\ drink$ [x]) $\longrightarrow$ ($\exists^{cl}$[x] ($drink$ [x] $\longrightarrow$ ($\forall$[y] ($drink$ [y])))))
  where
    $drink\ xs = \mathtt{Q}\langle\ulcorner\ulcorner xs\urcorner\urcorner\rangle$
```

Unlike in the previous examples the proof involves backtracking, as can be seen in steps 11 and 13.

```
===Step 0
 Goal:
:-         (all x. Q x -> bottom -> bottom -> Q x) ->
             (all x. Q x -> (all y. Q y) -> bottom) -> bottom
Stripped implication.
===Step 1
 Goal:
u        : (all x. Q x -> bottom -> bottom -> Q x)
:-         (all x. Q x -> (all y. Q y) -> bottom) -> bottom
Stripped implication.
===Step 2
 Goal:
u'       : (all x. Q x -> (all y. Q y) -> bottom)
u        : (all x. Q x -> bottom -> bottom -> Q x)
:-         bottom
Resolvable with u'
===Step 3
 Goal:
u'       : (all x. Q x -> (all y. Q y) -> bottom)
```

```
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q X -> (all y. Q y)
Stripped implication.
===Step 4
 Goal:
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        (all y. Q y)
Stripped variable(s).
===Step 5
 Goal:
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y
Resolvable with u
===Step 6
 Goal:
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y -> bottom -> bottom
Stripped implication.
===Step 7
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        bottom
Resolvable with u''' and u'
===Step 8
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y
Resolvable with u
===Step 9
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y -> bottom -> bottom
Stripped implication.
===Step 10
 Goal:
```

```
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        bottom
Resolvable with u''' and u'
===Step 11
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y
Failed to resolve.
===Step 11
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q X' [y] -> (all y. Q y)
Stripped implication.
===Step 12
 Goal:
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        (all y. Q y)
Stripped variable(s).
===Step 13
 Goal:
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y'
Failed to resolve.
Failed step 12.
Failed step 11.
Failed step 10.
Failed step 9.
Failed step 8.
===Step 8
 Goal:
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
```

```
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q X' [y] -> (all y. Q y)
Stripped implication.
===Step 9
 Goal:
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        (all y. Q y)
Stripped variable(s).
===Step 10
 Goal:
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y'
Resolvable with u
===Step 11
 Goal:
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y' -> bottom -> bottom
Stripped implication.
===Step 12
 Goal:
u'''''  : Q y' -> bottom
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        bottom
Resolvable with u''''' and u'''
===Step 13
 Goal:
u'''''  : Q y' -> bottom
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y'
Failed to resolve.
```

```
===Step 13
 Goal:
u'''''  : Q y' -> bottom
u''''   : Q X' [y]
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        Q y
Resolvable with u''''
Completed step 13.
Completed step 12.
Completed step 11.
Completed step 10.
Completed step 9.
Completed step 8.
Completed step 7.
Completed step 6.
Completed step 5.
Completed step 4.
Completed step 3.
Completed step 2.
Completed step 1.
Completed step 0.
====] Proved [==================
Final sequent was:
u'''''  : Q y' -> bottom
u''''   : Q y
u'''    : Q y -> bottom
u''     : Q X
u'      : (all x. Q x -> (all y. Q y) -> bottom)
u       : (all x. Q x -> bottom -> bottom -> Q x)
:-        bottom

Final context was:
Signatures      : (none)
Flexibles       : X
Forbiddens      : y' and y

Proof   : \u\u'((u' X) \u''\y((u (\yy y)) \u'''((u' (\yy y))
           \u''''\y'((u ((\y'\yy' y') y)) \u'''''(u''' u''''))))))
```

Appendix B

Supporting functions

This section collects general functions used in the implementation. The function *mapp* accepts two lists of values and builds a function that maps between the two coordinate-wise.

```
$mapp :: Eq\ a \Rightarrow [a] \rightarrow [b] \rightarrow a \rightarrow b$
$mapp\ [\,]\ \_ = mapp\ [\,]\ [\,]$
$mapp\ \_\ [\,] = mapp\ [\,]\ [\,]$
$mapp\ (x : xs)\ (y : ys) =$
  $\lambda z \rightarrow$ if $z \equiv x$
      then $y$
      else $mapp\ xs\ ys\ z$
```

The function *countUp* builds a list of n numbered elements.

```
$countUp :: Integer \rightarrow [Integer]$
$countUp\ 0 = [\,]$
$countUp\ 1 = [0]$
$countUp\ n = (n' : (countUp\ n'))$
  where $n' = n - 1$
```

Function *swap* is used to reorient unification problems.

```
$swap :: (a, b) \rightarrow (b, a)$
$swap\ (x, y) = (y, x)$
```

Combinator *foldR* is a more general form of *foldr*, since the step function is passed the rest of the list.

```
$foldR :: (a \rightarrow [a] \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
$foldR\ \_\ b\ [\,] = b$
$foldR\ f\ b\ (x : xs) = f\ x\ xs\ (foldR\ f\ b\ xs)$
```

Boolean function *distinct* checks that the elements of a list are pairwise distinct.

```
$distinct\ l = foldR\ (\lambda x \rightarrow \lambda xs \rightarrow \lambda next \rightarrow (x \notin xs) \wedge next)\ True\ l$
```

The following function encodes the standard definition of restriction.

```
$\_ \upharpoonright [\,] = [\,]$
$[\,] \upharpoonright \_ = [\,]$
$((v, e) : es) \upharpoonright vs =$
  $\textbf{if}\ v \in vs$
  $\textbf{then}\ ((v, e) : (es \upharpoonright vs))$   -- keep
  $\textbf{else}\ (es \upharpoonright vs)$   -- drop
```

The trampoline function *genTrampoline* is used to manage iteration of functions.

```
$genTrampoline\ p\ f\ x = p\ \$\ iterate\ f\ x$
```

Pointwise intersection of two lists is defined next.

```
$[\,] \Cap \_ = [\,]$
$\_ \Cap [\,] = [\,]$
$(x : xs) \Cap (y : ys) =$
  $\textbf{if}\ x \equiv y$
  $\textbf{then}\ (x : (intersectPT\ xs\ ys))$
  $\textbf{else}\ (intersectPT\ xs\ ys)$
```

The next definition adapts the *map* function to a list of pairs of the same type.

```
$mapPair\ f\ l = zip\ l1\ l2$
  $\textbf{where}\ l' = unzip\ l$
    $l1 = f\ \overrightarrow{(fst\ l')}$
    $l2 = f\ \overrightarrow{(snd\ l')}$
```

The following definition abbreviates mapping a function to the second component in a triple.

```
$mapTriple2\ f\ l = (\lambda(x, y, z) \rightarrow (x, f\ y, z))\ \overrightarrow{l}$
```

The *replace* function over lists has the obvious definition.

```
$replace :: Eq\ a \Rightarrow a \rightarrow a \rightarrow [\,a\,] \rightarrow [\,a\,]$
$replace\ x\ y\ [\,] = [\,]$
$replace\ x\ y\ (z : zs) =$
  $\textbf{let}\ rest = replace\ x\ y\ zs$
  $\textbf{in if}\ x \equiv z$
    $\textbf{then}\ (y : rest)$
    $\textbf{else}\ (z : rest)$
```

Bibliography

Berger, U. & Schwichtenberg, H. (1991), An inverse of the evaluation functional for typed λ–calculus, *in* R. Vemuri, ed., 'Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, Los Alamitos, pp. 203–211.

Berghofer, S. (2003), Proofs, Programs and Executable Specifications in Higher Order Logic, PhD thesis, Institut für Informatik, Technische Universität München.

Jones, S. (2003), *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press.

Miller, D. (1991), 'A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification', *Journal of Logic and Computation* **1**(4), 497–536.

Nipkow, T. (1993), 'Functional Unification of Higher-Order Patterns', *Logic in Computer Science* pp. 64–74.

Schwichtenberg, H. (2004), Proof Search in Minimal Logic, *in* 'Artificial Intelligence and Symbolic Computation', Vol. 3249/2004 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 15–25.

Wand, M. (1987), 'A Simple Algorithm and Proof for Type Inference', *Fundamenta Infomaticae* **10**, 115–122.

Index

- Q_L, 15
- Q_L^+x, 15
- Q_L^-x, 15
- $Q_\exists$, 15
- $Q_\exists^+x$, 15
- $Q_\exists^-x$, 15
- $Q_\forall$, 15
- $Q_\forall^+x$, 15
- $Q_\forall^-x$, 15
- $\Cap$, 41
- Γ, 8
- $\downarrow_t^x e$, 13
- $\fatsemi$, 7
- $[\![Exp]\!]_{Ctxt}$, 13
- $\overline{\rho}^{\text{fun}}$, 19
- $\restriction$, 41
- $\sharp_x$, 13
- $\sharp_{ns} n$, 19
- $\subseteq_\Lambda$, 19
- $\tau\{i \mapsto \tau_2\}$, 11
- τ, 8
  - $\Rightarrow$, 8
  - O, 8
- τ', 8
  - $\overset{\circ}{\Rightarrow}$, 8
  - $\overset{\circ}{n}$, 8
- $\ulcorner xs \urcorner$, 19
- $\ulcorner\!\ulcorner ns \urcorner\!\urcorner$, 19
- $\uparrow_t^x e$, 13
- $r\{x \mapsto s\}$, 6
- *Eqns*, 16
- *Exp*, 4
  - $\bullet$, 4
  - $\lambda\!\!\lambda$, 4
  - x, 4
- *FAE*, 4
  - $\bullet$, 4
  - $\lambda\!\!\lambda$, 4
  - x, 4
- *Formula*, 22
  - $P\langle\text{FAE}\rangle$, 22
  - $\forall$, 22
  - $\longrightarrow$, 22
- *Goal*, 9
- *Proof*, 23
- *QFormula*, 22
- *QPrefix*, 15
- *QSequent*, 23
- *QTerm*, 15
- *Reformula*, 23
- *Sem*, 12
  - *APPLY*, 12
  - *FREE*, 12
  - *LAM*, 12
  - *SYN*, 12
- *Sequent*, 23
  - $\Longrightarrow$, 23
- *Subst*, 6
- *Subs*, 9
- *TEqn*, 8
- *TVariable*, 8
- *Term*, 5
  - *Exp*, 5
  - *FAE*, 5
  - *Formula*, 22
  - *QTerm*, 15
  - *Sequent*, 23
  - *UProblem*, 17
- *UProblem*, 16
- *actionTable*, 9
- *add_ctx*, 13
- *appSubstForm*, 30
- *appSubstSequ*, 30
- *boundVars*, 5
- *canPrune*, 19
- *compact1*, 5
- *compact*, 5
- *countUp*, 40
- *distinct*, 40
- *doSubst*, 7
- *dropExtra*, 7

empty_ctx, 13
foUnifn, 10
foldR, 40
freeVars, 5
freshen, 30
genTrampoline, 41
isClosed, 5
isPattern, 16
leastInstance, 11
liftExpForm, 23
liftFrmtoSeq, 23
lookup_ctx, 13
mapPair, 41
mapTriple2, 41
mapp, 40
mkSubst, 6
nbe, 14
normalise, 5
occursCheck, 10
propagPrune, 19
prove, 24
reformulate, 23
remSelfMap, 7
replace, 41
resolve, 27
shouldPrune, 18
skeleton, 9
strip, 26
swap, 40
toCommonNotation, 6
toFunctorArguments, 6
tryMatching, 28
undefInit, 9
unreformulate, 23
normalisNbE, 14
search, 30
typeOf, 11
unify, 20