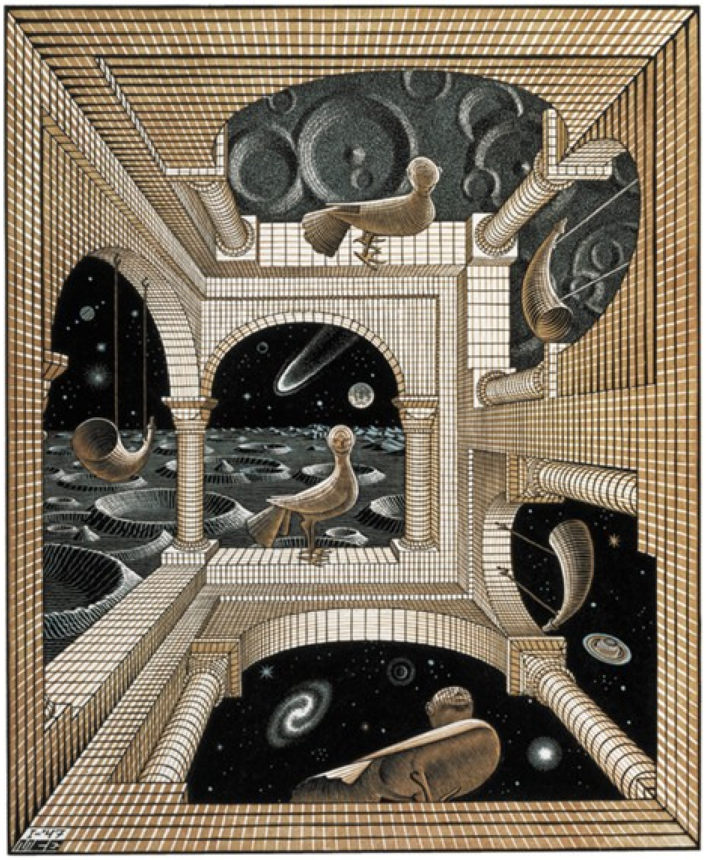
An Implementation of Escher

A Work In Perpetual Progress
Version 2015



Kee Siong Ng

Contents

Chapter 1

Introduction

Escher is a functional logic programming language first introduced in [Llo95] and [Llo99]. It was designed with the intention to provide in a simple computational mechanism the best features of functional and logic programming. The basic approach taken in the desgin of Escher is simple: start from Haskell and add logic programming facilities. (There are other approaches one can take in the design of functional logic programming languages; see, for example, [NM98] and [Han94].)

To understand Escher, we need to understand two things. The first is the form of a valid Escher program. The second is the underlying computational mechanism of the language. These are covered in §1.1, which is essentially a summary of [Llo03, Chap. 5]. In §1.2 the relationships between Escher, Haskell and Prolog are clarified. We give some example Escher programs in §1.3.

1.1 Logical Foundation

The logic underlying Escher is a polymorphically typed higher-order logic. The *terms* of the logic are the terms of the typed λ-calculus, formed in the usual way by application, abstraction, and tupling from the set of constants and a set of variables. An Escher program is a theory in the logic in which each formula is a particular kind of equation, namely, a statement.

Definition 1.1.1. A *statement* is a term of the form $h = b$, where h has the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, each free variable in h occurs exactly once in h, and b is type-weaker than h.

The term h is called the *head* and the term b is called the *body* of the statement. The statement is said to be *about* f.

For our purpose here, we say a term $s : \sigma$ is type weaker than a term $t : \tau$ if there exists a type substitution γ such that $\tau = \sigma\gamma$ and every free variable in s is a free variable in t. The type weaker condition stipulates that the body of a statement cannot contain free variables not already occurring in the head of the statement.

Definition 1.1.2. The definition of a function f is the collection of all statements about f, together with the signature for f.

Definition 1.1.3. An *Escher program* is a collection of definitions.

Example 1.1.4. Here are two Escher programs for performing list concatenations.

$$\mathit{concat_1} : \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$$

$$\mathit{concat_1}\ []\ x = x$$

$$\mathit{concat_1}\ (\#\ x\ y)\ z = (\#\ x\ (\mathit{concat_1}\ y\ z))$$

$$
\begin{aligned}
&\mathit{concat_2} : \mathit{List}\ a \times \mathit{List}\ a \times \mathit{List}\ a \to \Omega \\
&\mathit{concat_2}\ (u, v, w) = (u = [\,]\ \wedge\ v = w) \vee \\
&\qquad\qquad\qquad \exists r.\exists x.\exists y.(u = (\#\, r\, x)\ \wedge\ w = (\#\, r\, y)\ \wedge\ \mathit{concat_2}\ (x, v, y))
\end{aligned}
$$

The first is written in the functional programming style. (It is in fact a valid Haskell program.) The second is written in the relational or logic programming style. The term $\mathit{concat_2}\ (x, y, z)$ evaluates to $\top$ iff z is a concatenation of x and y. We will look at $\mathit{concat_2}$ in more details to see how logic programming is supported in Escher shortly.

Definition 1.1.5. A *redex* of a term t is a subterm of t that is α-equivalent to an instance of the head of a statement.

Recall that two terms are α-equivalent iff they differ only in the names of bound variables. A subterm s of t is a redex if we can find a statement $h = b$ and a term substitution θ mapping variables to terms such that $h\theta$ is α-equivalent to s.

A redex is outermost if it is not a proper subterm of another redex. Two outermost redexes are by definition disjoint. We are interested primarily in outermost redexes because we want the evaluation strategy to be lazy.

Given an Escher program and a term t, a redex selection rule S maps t to a subset of the set of outermost redexes in t. A standard redex selection rule is the leftmost selection rule S_L. Given a term t, the rule S_L picks out the (single) leftmost outermost redex in t. This is the selection rule implemented in the current Escher interpreter.

Definition 1.1.6. A term s is obtained from a term t by a *computation step* using the selection rule S if the following conditions are satisfied.

1. $S(t) = \{r_i\}$ is non-empty.
2. For each i, the redex r_i is α-equivalent to an instance $h_i\theta_i$ of the head of a statement $h_i = b_i$ for some term substitution θ_i.
3. s is the term obtained from t by replacing each redex r_i by $b_i\theta_i$.

Definition 1.1.7. A *computation* from a term t is a sequence $\{t_i\}_{i=1}^{n}$ of terms where $t = t_1$ and t_{i+1} is obtained from t_i by a computation step. The term t_1 is called the *goal* of the computation and t_n is called the *answer*.

As is standard in typed declarative languages, run-time type checking is not necessary in Escher. The fact that the body of every statement is type weaker than its head and that every free variable in the head of a statement occurs exactly once ensures that every computation step produces a new term that is well typed.

Central to Escher are some basic functions defined in the booleans module. These functions, together with the term rewriting mechanism described above, provide logic programming facilities in the functional programming setting. I list here some of these boolean functions.

$$\top \wedge x = x \tag{1.1}$$

$$\bot \wedge x = \bot \tag{1.2}$$

$$\exists x.\bot = \bot \tag{1.3}$$

$$\mathbf{u} \wedge \exists x.\mathbf{v} = \exists x.(\mathbf{u} \wedge \mathbf{v}) \tag{1.4}$$

$$\exists x_1.\exists x_2.\cdots.\exists x_n.(\mathbf{x}\ \wedge\ (x_1 = \mathbf{u})\ \wedge\ \mathbf{y}) = \exists x_2.\cdots.\exists x_n.(\mathbf{x}\{x_1/\mathbf{u}\} \wedge \mathbf{y}\{x_1/\mathbf{u}\}) \tag{1.5}$$

$$\forall x.(\bot \to \mathbf{u}) = \top \tag{1.6}$$

$$
\begin{aligned}
&\forall x_1.\forall x_2.\cdots.\forall x_n.(\mathbf{x}\ \wedge\ (x_1 = \mathbf{u})\ \wedge\ \mathbf{y} \to \mathbf{v}) = \\
&\qquad\qquad \forall x_2.\cdots.\forall x_n.(\mathbf{x}\{x_1/\mathbf{u}\}\ \wedge\ \mathbf{y}\{x_1/\mathbf{u}\} \to \mathbf{v}\{x_1/\mathbf{u}\})
\end{aligned}
\tag{1.7}
$$

Most of these equations are fairly straightforward. One thing is worth noting though. Variables typeset in bold above are actually syntactical variables. So an equation like (1.4) actually stands for a (possibly infinite) collection of Escher statements with **u** and **v** instantiated to all possible terms of type boolean. The use of syntactical variables usually come with side conditions. For example, for (1.4) to be applicable, the syntactical variable **u** must not contain a free occurrence of x. Similarly, x_1 must not occur free in **u** for (1.5) and (1.7) to work.

Example 1.1.8. The following is an example Escher computation using the S_L redex selection rule. The redex selected at each time step is highlighted. Note how Equation (1.5) given above is used to remove the existential quantifiers.

$$
\begin{aligned}
& \underline{\mathit{concat_2}\ ([\mathit{1}],[\mathit{2}],w)} \\
= \ & (\underline{[1] = []} \wedge [2] = w) \ \vee \ \exists r.\exists x.\exists y.([1] = (\# \, r \, x) \ \wedge \ w = (\# \, r \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y)) \\
= \ & \underline{(\bot \wedge [2] = w)} \ \vee \ \exists r.\exists x.\exists y.([1] = (\# \, r \, x) \ \wedge \ w = (\# \, r \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y)) \\
= \ & \underline{\bot \ \vee \ \exists r.\exists x.\exists y.([1] = (\# \, r \, x) \ \wedge \ w = (\# \, r \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y))} \\
= \ & \exists r.\exists x.\exists y.(\underline{[1] = (\# \, r \, x)} \ \wedge \ w = (\# \, r \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y)) \\
= \ & \underline{\exists r.\exists x.\exists y.(r = 1 \ \wedge \ x = [] \ \wedge \ w = (\# \, r \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y))} \\
= \ & \underline{\exists x.\exists y.(x = [] \ \wedge \ w = (\# \, 1 \, y) \ \wedge \ \mathit{concat_2}\ (x,[\mathit{2}],y))} \\
= \ & \exists y.(w = (\# \, 1 \, y) \ \wedge \ \underline{\mathit{concat_2}\ ([],[\mathit{2}],y)}) \\
& \ldots \\
= \ & \underline{\exists y.(w = (\# \, 1 \, y) \ \wedge \ y = [2])} \\
= \ & (w = [1,2])
\end{aligned}
$$

Example 1.1.9. Given the goal $\mathit{concat_2}\ (x, y, [\mathit{1}, \mathit{2}])$, Escher will return with the following answer

$$(x = [] \ \wedge \ y = [1,2]) \ \vee \ (x = [1] \ \wedge \ y = [2]) \ \vee \ (x = [1,2] \ \wedge \ y = []),$$

which is computed using the same mechanism described in the previous example.

1.2 Escher, Haskell and Prolog

We first explore the relationship between Escher and Haskell. At the logic level, every Haskell program is an Escher program and every Escher program is a (syntactically-correct) Haskell program which may not compile. In that sense, Escher is a superset of Haskell. The difference between Escher and Haskell comes down to the following two points.

- Haskell allows pattern matching only on data constructors. Escher extends this by allowing pattern matching on function symbols as well as data constructors. Examples of equations that Haskell cannot handle but Escher can are those in the booleans module given earlier.
- The second thing that Escher can do but Haskell can't is reduction of terms inside lambda abstractions. This mechanism allows Escher to handle sets (and similar data types) in a natural and intensional way. This is usually achieved with the use of syntactical variables.

The extra expressiveness afforded by Escher comes with a price tag, however. Some common optimisation techniques developed for efficient compilation of Haskell code (see [Pey87]) cannot be used in the implementation of Escher. In other words, efficiency is at present still a non-trivial issue for Escher.

We next explore the relationship between Escher and Prolog. Perhaps suprisingly, there is actually a significant overlap between the two languages. In fact, any Prolog program defined without using cuts can be mechanically translated into Escher via Clark's completion algorithm [Cla78]. For example, the Escher program $\mathit{concat_2}$ given earlier is just the completion of the following Prolog definition.

$\mathit{concat_2}\ ([], L, L).$

$\mathit{concat_2}\ ([X|L\mathit{1}], L\mathit{2}, [X|L\mathit{3}]) \longleftarrow \mathit{concat_2}\ (L\mathit{1}, L\mathit{2}, L\mathit{3}).$

Procedurally, there is a difference between Escher and Prolog in that Prolog computes alternative answers one at a time via backtrackking whereas Escher returns all alternative answers in a disjunction (a set). This point is illustrated in Example 1.1.9 above.

1.3 Example Programs

I end this short introduction with some example Escher programs. The aim here is to showcase the different styles of declarative programming supported by Escher. An Escher interpreter is available for download as a separate program from `http://rsise.anu.edu.au/~kee`.

Example 1.3.1. Here is how quick sort can be written in Escher. This is just a vanilla Haskell program that doesn't make use of special logic programming facilities in Escher.

$$\begin{aligned}&\mathit{qsort} : \mathit{List}\ a \to \mathit{List}\ a\\&\mathit{qsort}\ [] = []\\&\mathit{qsort}\ (\#\, x\, y) = \mathit{concat}_{\mathit{1}}\ (\mathit{qsort}\ (\mathit{filter}\ (\leq x)\ y))\ (\#\, x\ (\mathit{qsort}\ (\mathit{filter}\ (> x)\ y)))\end{aligned}$$

$$\begin{aligned}&\mathit{filter} : (a \to \mathit{\Omega}) \to \mathit{List}\ a \to \mathit{List}\ a\\&\mathit{filter}\ p\ [] = []\\&\mathit{filter}\ p\ (\#\, x\, y) = \mathit{if}\ \ (p\, x)\ \mathit{then}\ (\#\ x\ (\mathit{filter}\ p\ y)\ \mathit{else}\ (\mathit{filter}\ p\ y)\end{aligned}$$

Example 1.3.2. The following is an example of an Escher program for computing permutations of lists. The function *permute* returns true iff the two input arguments are permutations of each other. The function *delete* is a subsidiary function of *permute* that returns true iff the third argument is the result of removing the first argument from the second argument.

$$\begin{aligned}&\mathit{permute} : (\mathit{List}\ a) \times (\mathit{List}\ a) \to \mathit{\Omega}\\&\mathit{permute}\ ([], x) = (x = [])\\&\mathit{permute}\ ((\#\ x\ y), w) = \exists u.\exists v.\exists z.(w = (\#\ u\ v)\ \wedge\ \mathit{delete}\ (u, (\#\ x\ y), z)\ \wedge\ \mathit{permute}\ (z, v))\end{aligned}$$

$$\begin{aligned}&\mathit{delete} : a \times (\mathit{List}\ a) \times (\mathit{List}\ a) \to \mathit{\Omega}\\&\mathit{delete}(x, [], y) = \bot\\&\mathit{delete}(x, (\#\ y\ z), w) = (x = y\ \wedge\ w = z)\ \vee\ \exists v.(w = (\#\ y\ v)\ \wedge\ \mathit{delete}\ (x, z, v))\end{aligned}$$

Given $\mathit{permute}\ ([\mathit{1}, \mathit{2}, \mathit{3}], [\mathit{2}, \mathit{1}, \mathit{3}])$, Escher will return $\top$. Given $\mathit{permute}\ ([\mathit{1}, \mathit{2}, \mathit{2}], x)$, Escher will return the answer

$$x = [1, 2, 3]\ \vee\ x = [1, 3, 2]\ \vee\ x = [2, 1, 3]\ \vee\ x = [2, 3, 1]\ \vee\ x = [3, 1, 2]\ \vee\ x = [3, 2, 1].$$

Example 1.3.3. Here are some standard functions defined on sets.

$$\begin{aligned}&\mathit{union} : (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega})\\&\mathit{union}\ s\ t = \lambda x.((s\ x)\ \vee\ (t\ x))\end{aligned}$$

$$\begin{aligned}&\mathit{intersect} : (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega})\\&\mathit{intersect}\ s\ t = \lambda x.((s\ x)\ \wedge\ (t\ x))\end{aligned}$$

$$\begin{aligned}&\mathit{minus} : (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega})\\&\mathit{minus}\ s\ t = \lambda x.((s\ x)\ \wedge\ \neg(t\ x))\end{aligned}$$

$$\begin{aligned}&\mathit{subset} : (a \to \mathit{\Omega}) \to (a \to \mathit{\Omega}) \to \mathit{\Omega}\\&\mathit{subset}\ s\ t = \forall x.((s\ x) \to (t\ x)))\end{aligned}$$

Similar functions for multisets can be just as easily defined.

Chapter 7 contains more programming examples.

1.4 Installing Escher

The source code for Escher is available for download from `http://rsise.anu.edu.au/~kee/`. The file README.1ST contains installation instructions.

1.5 Using Escher

The following shows a simple session with Escher.

```
> escher -i
prompt> import booleans.es ;
  Reading booleans.es...done
prompt> import numbers.es ;
prompt> : (add 37.4 4.6) ;
  Query: ((add 37.4) 4.6)
  Answer: 42 ;
prompt> import lists.es ;
prompt> type ListInt = (List Int) ;
prompt> myg : ListInt -> Bool ;
prompt> (myg []) = False ;
prompt> (myg (# x y)) = True ;
prompt> set1 : ListInt -> Bool ;
prompt> set1 = { [], [1,2,3] } ;
prompt> set2 : ListInt -> Bool ;
prompt> set2 = { [] } ;
prompt> : \exists y.(&& (set1 y) (myg y)) ;
  Query: (sigma \y.((&& (set1 y)) (myg y)))
  Answer: True ;
prompt> : \exists y.(&& (set2 y) (myg y)) ;
  Query: (sigma \y.((&& (set2 y)) (myg y)))
  Answer: False ;
prompt> import lists.es ;
  Reading lists.es...done
prompt> : (permute ([1,2],x)) ;
  Query: (permute (((# 1) ((# 2) [])),x))
  Answer: ((|| ((== x) ((# 1) ((# 2) [])))) ((== x) ((# 2) ((# 1) [])))) ;
prompt> quit ;
Quiting Escher...
```

1.6 Escher Prelude

Escher supports six system-defined types: `Bool`, `Int`, `Float`, `Char`, `String`, `ListString`. The last of these, `ListString`, is actually a synonym for `(List Char)`.

There are a number of system modules: Booleans, Numbers, Lists, and Sets. These can be found in Chapter 6.

1.7 Escher's Syntax

We give the grammar for valid input to the interpreter in this section. Regular expressions for tokens like `FILENAME`, `IDENTIFIER1`, `IDENTIFER2` etc are given at the end of the section.

The Escher interpreter takes `program_statements` one at a time. A program statement is either an import statement, an Escher statement, a query or a quit instruction.

```
    input : program_statements ;

    program_statements : /* empty */ | program_statements program_statement ;
```

```
program_statement : import | statement | type_info | query | quit ;

quit : "quit" ';'
import : "import" FILENAME ';'
statement : term '=' term ;
```

We now look at the grammar for terms. A `term` is a term possibly with syntactic variables in it. The grammar for `term` is defined inductively as follows. Each syntactic variable is a term. Each variable is a term. Each constant, which can be either a function or a data constructor, is a term. If `t1` and `t2` are terms having appropriate types, then `(t1 t2)` is a term. If `x` is a variable and `t` is a term, then `\x.t` is a term. If `t1`, `t2`, …, `tn` are terms, then `(t1,t2,...,tn)` is a term. Syntactic variables can come with side conditions. These are stated using `sv_condition`. There are four kinds of condition we can state. We can specify that a syntactic variable must be a variable or a constant. We can also require that the instantiation of a syntactic variable be equal or not equal to the instantiation of an earlier syntactic variable.

```
term : SYNTACTIC_VARIABLE | SYNTACTIC_VARIABLE sv_condition
     | VARIABLE
     | FUNCTION | DATA_CONSTRUCTOR | DATA_CONSTRUCTOR_INT
     | DATA_CONSTRUCTOR_FLOAT | DATA_CONSTRUCTOR_STRING
     | IDENTIFIER1 | IDENTIFIER2
     | '(' term term ')'
     | '\' VARIABLE '.' term
     | '(' terms_product ')'
     | term_sugar
     ;
terms : term | terms term ;

terms_product : /* empty */ | terms_product ',' term ;

sv_condition : '/' VAR '/' | '/' CONST '/'
             | '/' EQUAL ',' SYNTACTIC_VARIABLE '/'
             | '/' NOTEQUAL ',' SYNTACTIC_VARIABLE '/'
             ;
```

Terms as defined can be cumbersome to work with. To ease the writing of the spec file, syntactic sugars are provided for sets, lists, and the quantifiers. This is how they works.

- A set like `{t1, t2}` will be turned into the term

  ```
  \x.(ite (== x t) True (ite (== x t2) True False))
  ```

  before Escher can process it. Here `ite` is the familiar *if-then-else* function.
- A list like `[ t1, t2 ]` will be turned into the term `(# t1 (# t2 []))`.
- In accordance with the mathematics (see [Llo03, pg. 43]), a formula like `\exists x.t` will be turned into the term `(sigma \x.t)` and a formula like `\forall x.t` will be turned into the term `(pi \x.t)`.

Another syntactic sugar we provide is the ability to enclose terms obtained from multiple applications within a single pair of brackets. So, a term like `(((f x) y) z)` can be more simply written as `(f x y z)`.

```
term_sugar : '(' term term terms ')'
           | '{' terms_product '}'
           | '[' terms_product ']'
           | '\' "exists" VARIABLE '.' term
           | '\' "forall" VARIABLE '.' term
           ;
```

To facilitate type checking, the signature of every constant used in the program must be explicitly stated. The grammar for such type declarations is given below. Type synonyms can be used to simplify these type declarations.

```
 type_info : functionsymbol ’:’ type ’;’
           | dataconstructors ’:’ type ’;’
           | "type" IDENTIFIER2 ’=’ type ’;’
           ;
functionsymbol : IDENTIFIER1 | FUNCTION ;

dataconstructors : dataconstructor | dataconstructors ’,’ dataconstructor ;
dataconstructor : IDENTIFIER2 | DATA_CONSTRUCTOR ;
```

We now look at the grammar for types. A parameter (type variable) is a type. These are alphanumerics that start with a lower case letter. Each of the basic nullary type constructors like Bool, Number and String is a type. If `T` is a *n*-ary type constructor and `t1`, `t2`, …, `tn` are types, then `(T t1 t2 ... tn)` is a type. If `t1` and `t2` are types, then `t1 -> t2` is a type. If `t1`, `t2`, …, `tn` are types, then `(t1 * t2 * ... * tn)` is a type.

```
 type : IDENTIFIER1
      | "Bool" | "Number" | "String" | IDENTIFIER2
      | ’(’ IDENTIFIER2 types ’)’
      | ’(’ products ’)’
      | type "->" type
      | ’(’ type ’)’
      ;
 products : products ’*’ type | type ’*’ type ;

 types : type | types type ;
```

Here are the regular expressions for the tokens used in the grammar. IDENTIFIER1 are alphanumerics that start with a lower case letter. IDENTIFIER2 are alphanumerics that start with an upper case letter. System-defined data constructors and functions are declared here. A file that can be imported into the spec file must end with ".e". Variables and syntactic variables are also governed by fixed rules here. Care should be taken with variables. A lot of programming errors are associated with the use of variable names that does not actually conform to the grammar.

```
IDENTIFIER1 = [a-z][a-zA-Z0-9\_\’]*
IDENTIFIER2 = [A-Z][a-zA-Z0-9\_\’]*
DATA_CONSTRUCTOR = (True | False | # | [])
DATA_CONSTRUCTOR_FLOAT = -?[0-9]+\.[0-9]+
DATA_CONSTRUCTOR_INT = -?[0-9]+
DATA_CONSTRUCTOR_STRING = \"[a-zA-Z0-9\-\_\+\: ]*\"
FUNCTION = (== | /= | <= | < | >= | > | && | ||)
FILENAME = [a-zA-Z\/0-9\_\.]+\.es
VARIABLE = [m-z][0-9]*
SYNTACTIC_VARIABLE = [a-zA-Z][0-9]*\_SV
```

Chapter 2

Types and Terms

2.1 Types

Comment 2.1.1. Types are defined inductively in the logic, thus lending itself nicely to the use of composite pattern [GHJV95, p.163] for its implementation.

We differentiate between atomic and composite types. Atomic types are obtained from type constructors with arity 0. Examples of these include *int*, *float*, *nat*, *char*, *string*, etc. (Note that *string* is a nullary type constructor in this case. Strings in general can also be constructed from *List char*.) They are the base types, and occupy the leaf nodes of a composite type structure. Everything else are composite types. Examples of composite types include types obtained from type constructors of non-zero arity like *List* α, *Btree* α, *Graph* α β, etc; function types like *set* α (this is equivalent to $\alpha \to \Omega$) and *multiset* α ($\alpha \to nat$); and product types obtained from the tuple-forming operator.

The following is an outline of the data types module. We first give the abstract classes, followed by the actual data types.

8 ⟨types.h 8⟩≡

```
#ifndef _DATATYPE_H_
#define _DATATYPE_H_

#include <set>
#include <vector>
#include <string>
#include <assert.h>
#include <iostream>
using namespace std;
#define dcast dynamic_cast
#define unint unsigned int

extern const string underscore, alpha, Parameter, Tuple, Arrow,
                    gBool, gInt, gFloat, gChar, gString;

⟨type::function declarations 12f⟩
⟨type::type 9b⟩
⟨type::composite types 10c⟩
⟨type::parameters 11d⟩
⟨type::tuples 13c⟩
⟨type::algebraic types 16b⟩
⟨type::abstractions 14b⟩
⟨type::synonyms 13b⟩
```

```
  #endif
```

9a ⟨types.cc 9a⟩≡

```
  #include "types.h"
  #include <stdlib.h>


  ⟨type::functions 10b⟩
  ⟨type::composite types::implementation 10d⟩
  ⟨type::parameters::implementation 12a⟩
  ⟨type::tuples::implementation 13d⟩
  ⟨type::algebraic types::implementation 17a⟩
  ⟨type::abstractions::implementation 14c⟩
```

Comment 2.1.2. The top-level `type` structure contains as members those variables and functions that are common to all types. Every type obviously has a name.

The functions `setAlpha` and `addAlpha` are used to configure subtypes; they are defined only for composite types like tuples and list. (See Comment 2.1.4 for details.)

9b ⟨type::type 9b⟩≡

```
  class type {
  public:
          int count;
          type() { count = 0; }
          type(string n) : tag(n) { count = 0; }
          virtual ~type() {}
          virtual void setAlpha(type * x, unsigned int y) {}
          virtual void addAlpha(type * x) {}
          virtual type * getAlpha(unsigned int x) { return NULL; }
          virtual int alphaCount() { return 0; }
          virtual bool isComposite() { return false; }
          virtual bool isTuple() { return false; }
          virtual bool isAbstract() { return false; }
          virtual bool isParameter() { return false; }
          virtual bool isSynonym() { return false; }
          virtual bool isUdefined() { return false; }
          virtual string getName() { return tag; }
          virtual string & getTag() { return tag; }
          virtual type * clone() { count++; return this; }
          virtual void deccount() { count--; }
          virtual void getParameters(set<string> & ret) {}
          virtual void renameParameters() {}
          virtual void renameParameter(string name) {}
  protected:
          string tag;
  };
```

Uses `getParameters` 11b, `renameParameter` 11c, and `renameParameters` 11c.

Comment 2.1.3. We use reference counting for the memory management of the base types. The variable `count` keeps track of the number of references to a type. Deallocation of a type structure is done using the function `delete-type` defined as follows.

10a ⟨type::type 9b⟩+≡

```
void delete_type(type ∗ x);
```

Defines:
`delete_type`, used in chunks 10d, 13b, 18–20, 23b, 25a, 27d, 41b, 141e, 149a, 153d, and 157a.

10b ⟨type::functions 10b⟩≡

```
void delete_type(type ∗ x) {
        // if (x->isComposite() || x->isParameter()) assert(x->count == 0);
        if (x→count ≡ 0) delete x; else x→deccount();
}
```

Defines:
`delete_type`, used in chunks 10d, 13b, 18–20, 23b, 25a, 27d, 41b, 141e, 149a, 153d, and 157a.

Comment 2.1.4. The following is the class declaration for composite types. The member `alpha` stores the sub-types in the composite structure. It serves different purposes for different kinds of composite types.

10c ⟨type::composite types 10c⟩≡

```
class type_composite : public type {
protected:
        vector<type ∗> alpha;
public:
        virtual ∼type_composite();
        bool isComposite() { return true; }
        virtual void deccount();
        virtual void setAlpha(type ∗ x, unsigned int y);
        virtual void addAlpha(type ∗ x) { alpha.push_back(x); }
        virtual type ∗ getAlpha(unsigned int x);
        virtual int alphaCount() { return alpha.size(); }
        virtual string getName();
        virtual type ∗ clone() { assert(false); }
        virtual void getParameters(set<string> & x);
        virtual void renameParameters();
        virtual void renameParameter(string name);
};
```

Uses `getParameters` 11b, `renameParameter` 11c, and `renameParameters` 11c.

10d ⟨type::composite types::implementation 10d⟩≡

```
type_composite::∼type_composite() {
        for (unsigned int i=0; i≠alpha.size(); i++) delete_type(alpha[i]);
}
```

Uses `delete_type` 10a 10b.

10e ⟨type::composite types::implementation 10d⟩+≡

```
void type_composite::deccount() {
        count−−;
        for (unsigned int i=0; i≠alpha.size(); i++) alpha[i]→deccount();
}
```

11a ⟨type::composite types::implementation 10d⟩+≡

```
void type_composite::setAlpha(type ∗ x, unsigned int y)
       { assert(y < alpha.size()); alpha[y] = x; }

type ∗ type_composite::getAlpha(unsigned int x)
       { assert(x < alpha.size()); return alpha[x]; }

string type_composite::getName() { assert(false); return ""; }
```

Comment 2.1.5. The following functions are used during unification and type-checking. The first one collects in a set all the parameters in a type. This is used in the unification algorithm. The second and third functions are used to rename parameters during instantiation and type checking.

11b ⟨type::composite types::implementation 10d⟩+≡

```
void type_composite::getParameters(set<string> & ret) {
         for (unsigned int i=0; i≠alpha.size(); i++)
                  alpha[i]→getParameters(ret);
}
```

Defines:
`getParameters`, used in chunks 9–12 and 19a.

11c ⟨type::composite types::implementation 10d⟩+≡

```
void type_composite::renameParameters() {
         set<string> ps;
         getParameters(ps);
         set<string>::iterator p = ps.begin();
         while (p ≠ ps.end()) { renameParameter(∗p); inc_counter(); p++; }
}
void type_composite::renameParameter(string name) {
         for (unsigned int i=0; i≠alpha.size(); i++)
                  alpha[i]→renameParameter(name);
}
```

Defines:
`renameParameter`, used in chunks 9–12.
`renameParameters`, used in chunks 9–12 and 23a.
Uses `getParameters` 11b and `inc_counter` 13a.

Comment 2.1.6. Parameters are type variables.

11d ⟨type::parameters 11d⟩≡

```
class type_parameter : public type {
public: type_parameter();
         type_parameter(string x) { tag = Parameter; vname = x; }
         type ∗ clone() { return new type_parameter(vname); }
         bool isParameter() { return true; }
         string getName() { return tag + underscore + vname; }
         void getParameters(set<string> & ret);
         void renameParameters();
         void renameParameter(string name);
private:
         string vname;
};

extern string newParameterName();
```

Uses `getParameters` 11b, `renameParameter` 11c, and `renameParameters` 11c.

Comment 2.1.7. When we create a new type parameter, a distinct name of the form `alpha_i` where `i` is a number will be assigned to the parameter.

12a ⟨type::parameters::implementation 12a⟩≡

```
  #include "global.h"
  static int parameterCount = 0;
  type_parameter::type_parameter() {
          tag = Parameter; vname = newParameterName(); }
```

Comment 2.1.8. New parameter names are created using this next function. The variable `parameterCount` is used here as the index for new parameter names. This function can be replaced with `newVar` in terms.nw.

12b ⟨type::parameters::implementation 12a⟩+≡

```
  string newParameterName() {
          string vname = alpha + numtostr(parameterCount++);
          return vname;
  }
```

12c ⟨type::parameters::implementation 12a⟩+≡

```
  void type_parameter::getParameters(set<string> & ret) {
          string temp = tag + underscore + vname;
          ret.insert(temp);
  }
```

Uses `getParameters` 11b and `insert` 30e.

12d ⟨type::parameters::implementation 12a⟩+≡

```
  void type_parameter::renameParameters()
        { string temp = tag+underscore+vname; renameParameter(temp); inc_counter(); }
```

Uses `inc_counter` 13a, `renameParameter` 11c, and `renameParameters` 11c.

Comment 2.1.9. If a parameter has been indexed, we will first remove its index and then attach a new one. The function `rfind` returns `npos` if an underscore cannot be found in `vname`. (Search proceeds from the end of `vname`.)

12e ⟨type::parameters::implementation 12a⟩+≡

```
  void type_parameter::renameParameter(string name) {
          string tname = tag + underscore + vname;
          if (tname ≠ name) return;
          char temp[10]; sprintf(temp, "_%d", get_counter_value());

          unint i = vname.rfind(underscore);
          if (i ≥ 0 ∧ i < vname.size()) vname.erase(i, vname.size()-i);

          string temp2(temp); vname = vname + temp2;
  }
```

Uses `get_counter_value` 13a and `renameParameter` 11c.

Comment 2.1.10. Some times, parameters need to be renamed to avoid name capture. We use a global counter for this purpose.

12f ⟨type::function declarations 12f⟩≡

```
  void inc_counter();
  int get_counter_value();
```

Uses `get_counter_value` 13a and `inc_counter` 13a.

13a
```
⟨type::functions 10b⟩+≡
  static int counter = 0;
  void inc_counter() { counter++; }
  int get_counter_value() { return counter; }
```
Defines:
`get_counter_value`, used in chunk 12.
`inc_counter`, used in chunks 11 and 12.

Comment 2.1.11. Users can define type synonyms of the form $t_1 = t_2$, where t_1 is an identifier and t_2 the actual type. These are handled using the following class. The identifier t_1 is stored in `tname`; the actual type t_2 is stored in `actual`.

13b
```
⟨type::synonyms 13b⟩≡
  class type_synonym : public type {
  public:
          type_synonym(string name, type * ac)
                { tag = name; tname = name; actual = ac; }
          ~type_synonym() { delete_type(actual); }
          type * clone() {
                  // assert(actual); count++; actual->count++; return this; }
                  assert(actual);return new type_synonym(tname,actual→clone());}
          void deccount() { assert(false); }
          bool isSynonym() { return true; }
          type * getActual() { return actual; }
          string getName() { return actual→getName(); }
  private:
          type * actual;
          string tname;
  };
```
Uses `delete_type` 10a 10b.

Comment 2.1.12. We support the following base types: boolean, integer, float point number and string. Natural number is not supported because we can always use integer in its place.

Comment 2.1.13. The following is used to create product types.

13c
```
⟨type::tuples 13c⟩≡
  class type_tuple : public type_composite {
  public:
          type_tuple() { tag = Tuple; }
          type * clone();
          bool isTuple() { return true; }
          string getName();
  };
```

13d
```
⟨type::tuples::implementation 13d⟩≡
  type * type_tuple::clone() {
          type_tuple * ret = new type_tuple;
          for (int i=0; i≠alphaCount(); i++)
                  ret→addAlpha(alpha[i]→clone());
          return ret;
  }
```

14a ⟨type::tuples::implementation 13d⟩+≡

```
string type_tuple::getName() {
        string ret = "( ";
        for (unsigned int i=0; i≠alpha.size()-1; i++)
                ret = ret + alpha[i]→getName() + " * ";
        ret = ret + alpha[alpha.size()-1]→getName() + ")";
        return ret;
}
```

Comment 2.1.14. This is used for the construction of function types. It is worth mentioning that sets and multisets have function types.

Function types of particular interest here are those for transformations. The variable `rank` is used to record the rank of transformations. This value can be calculated using `compRank`. The functions `getSource` and `getTarget` returns the source and target of a transformation. The function `getArg` returns the n-th argument.

14b ⟨type::abstractions 14b⟩≡

```
class type_abs : public type_composite {
public:
        int rank;
        type_abs() { tag = Arrow; rank = -5; }
        type_abs(type ∗ source, type ∗ target) {
                tag = Arrow; rank = -5;
                addAlpha(source); addAlpha(target);
        }
        bool isAbstract() { return true; }
        type ∗ clone();
        type ∗ getArg(int n);
        type ∗ getSource();
        type ∗ getTarget();
        string getName();
        int compRank();
};
```

Defines:
  `getArg`, never used.
  `getSource`, never used.
  `getTarget`, never used.
Uses `compRank` 16a.

14c ⟨type::abstractions::implementation 14c⟩≡

```
type ∗ type_abs::clone() {
        type_abs ∗ ret = new type_abs(alpha[0]→clone(), alpha[1]→clone());
        ret→rank = rank;
        return ret;
}
```

15a ⟨type::abstractions::implementation 14c⟩+≡

```
string type_abs::getName() {
        string ret;
        if (alpha[0]→isComposite())
                ret = "(" + alpha[0]→getName() + ") -> ";
        else ret = alpha[0]→getName() + " -> ";
        if (alpha[1]→isComposite())
                ret = ret + "(" + alpha[1]→getName() + ")";
        else ret = ret + alpha[1]→getName();
        return ret;
}
```

15b ⟨type::abstractions::implementation 14c⟩+≡

```
type ∗ type_abs::getArg(int n) {
        assert(n < rank);
        type ∗ p = this;
        int temp = 0;
        while (temp ≠ n) { p = p→getAlpha(1); temp++; }
        return p→getAlpha(0);
}
```

Defines:
`getArg`, never used.

15c ⟨type::abstractions::implementation 14c⟩+≡

```
type ∗ type_abs::getSource() {
        assert(rank ≠ -5);
        type ∗ p = this;
        for (int i=0; i≠rank; i++) p = p→getAlpha(1);
        assert(p→getAlpha(0)); return p→getAlpha(0);
}
```

Defines:
`getSource`, never used.

15d ⟨type::abstractions::implementation 14c⟩+≡

```
type ∗ type_abs::getTarget() {
        assert(rank ≠ -5);
        type ∗ p = this;
        for (int i=0; i≠rank; i++) p = p→getAlpha(1);
        return p→getAlpha(1);
}
```

Defines:
`getTarget`, never used.

Comment 2.1.15. This function computes the rank of a transformation. We inspect the spine of the type and count the number of predicate types appearing in it.

16a ⟨type::abstractions::implementation 14c⟩+≡

```
int type_abs::compRank() {
        if (alpha[1]→isAbstract() ∧ alpha[0]→isAbstract() ∧
            alpha[0]→getAlpha(1)→getTag() ≡ gBool) {
                type_abs ∗ t = dcast<type_abs ∗>(alpha[1]);
                return 1 + t→compRank();
        }
        return 0;
}
```

Defines:
`compRank`, used in chunk 14b.

Comment 2.1.16. Algebraic types are supported using the following classes. The class `type_udefined` supports nullary type constructors; the class `type_alg` supports non-nullary type constructors. Perhaps it makes sense to combine the two in one type.

16b ⟨type::algebraic types 16b⟩≡

```
class type_udefined : public type {
        const vector<string> values;
public:
        type_udefined(string & tname, const vector<string> &vals)
                : type(tname), values(vals) {}
        type_udefined(string & tname) : type(tname) {}
        bool isUdefined() { return true; }
        // type * clone() { count++; return this; }
        const vector<string> & getValues() { return values; }
};
```

16c ⟨type::algebraic types 16b⟩+≡

```
class type_alg : public type_composite {
public:
        type_alg(string tid) { tag = tid; }
        type_alg(string tid, vector<type ∗> x) {
                tag = tid;
                for (unsigned int i=0; i≠x.size(); i++)
                        addAlpha(x[i]→clone());
        }
        type_alg(string tid, type_tuple ∗ x) {
                tag = tid;
                for (int i=0; i≠x→alphaCount(); i++)
                        addAlpha(x→getAlpha(i)→clone());
        }
        type ∗ clone() { return new type_alg(tag, alpha); }
        string getName();
};
```

17a ⟨type::algebraic types::implementation 17a⟩≡

```
string type_alg::getName() {
        string ret = "(" + tag;
        for (unint i=0; i≠alpha.size()-1; i++)
                ret = ret + " " + alpha[i]→getName();
        ret = ret + " " + alpha[alpha.size()-1]→getName() + ")";
        return ret;
}
```

2.1.1 Unification

Comment 2.1.17. We now discuss type unification. The type unification algorithm given here is adapted from the one given in [Pey87, Chap.5].

17b ⟨unification.h 17b⟩≡

```
#ifndef _UNIFICATION_H_
#define _UNIFICATION_H_

#include "terms.h"
#include "types.h"
#include <vector>
#include <utility>
struct term_type { term ∗ first; type ∗ second; };
extern bool unify(vector<pair<string,type ∗> > &eqns,type ∗tvn,type ∗t);
extern type ∗ apply_subst(vector<pair<string, type ∗> > & eqns, type ∗ x);
extern type ∗ wellTyped(term ∗ t);
extern pair<type ∗, vector<term_type> > mywellTyped(term ∗ t);
extern type ∗ get_type_from_syn(type ∗ in);

#endif
```

Defines:
  `term_type`, used in chunks 22b, 23b, 28b, 105, 138, and 143a.
Uses `apply_subst` 18b, `get_type_from_syn` 19c, `mywellTyped` 28b, `unify` 20, and `wellTyped` 28a.

17c ⟨unification.cc 17c⟩≡

```
#include <iostream>
#include <utility>
#include <vector>
#include <string>
#include "types.h"
#include "unification.h"
using namespace std;

bool unify_verbose = false; // set this to see the unification process
⟨unification body 18a⟩
⟨type checking 28a⟩
```

Comment 2.1.18. The function `getBinding` returns the binding for parameter x in a type substitution θ.

18a ⟨unification body 18a⟩≡

```
type ∗ getBinding(vector<pair<string, type ∗> > & eqns, type ∗ x) {
        assert(x→isParameter());
        string vname = x→getName();
        for (unsigned int i=0; i≠eqns.size(); i++)
                if (eqns[i].first ≡ vname) return eqns[i].second;
        return x;
}
```

Defines:
`getBinding`, used in chunks 18b and 20.

Comment 2.1.19. Given a type substitution θ and a type t with parameters, `apply_subst` computes $t\theta$.

18b ⟨unification body 18a⟩+≡

```
type ∗ apply_subst(vector<pair<string, type ∗> > & eqns, type ∗ t) {
        if (t→isParameter())
                return getBinding(eqns, t)→clone();
        type ∗ ret = t→clone();
        for (int i=0; i≠ret→alphaCount(); i++) {
                type ∗ temp = apply_subst(eqns, ret→getAlpha(i));
                delete_type(ret→getAlpha(i));
                ret→setAlpha(temp, i);
        }
        return ret;
}
```

Defines:
`apply_subst`, used in chunks 17b, 19b, 20, and 25a.
Uses `delete_type` 10a 10b and `getBinding` 18a.

Comment 2.1.20. This function extends a substitution θ with an additional equation $x = t$. If t is x, then the extension succeeds trivially. Otherwise, unless x appears in t, the extension succeeds.

18c ⟨unification body 18a⟩+≡

```
bool extend(vector<pair<string, type ∗> > & eqns, type ∗ x, type ∗ t) {
        assert(x→isParameter());
        ⟨delete eqns of the form x = x 18d⟩
        ⟨if x appears in t, return false 19a⟩
        ⟨apply (x,t) to each eqn in eqns, extend eqns and return true 19b⟩
}
```

Defines:
`extend`, used in chunk 20.

18d ⟨delete eqns of the form x = x 18d⟩≡

```
if (t→isParameter())
        if (x→getName() ≡ t→getName()) return true;
```

19a ⟨if x appears in t, return false 19a⟩≡

```
// case of t not a parameter
set<string> parameters;
t→getParameters(parameters);
// set<string>::iterator p = parameters.begin();
// cout << "parameters : ";
// while (p != parameters.end()) { cout << *p << " "; p++; }
if (parameters.find(x→getName()) ≠ parameters.end())
        return false;
```

Uses `getParameters` 11b.

19b ⟨apply (x,t) to each eqn in eqns, extend eqns and return true 19b⟩≡

```
for (unsigned int i=0; i≠eqns.size(); i++) {
        type * temp = eqns[i].second;
        eqns[i].second = apply_subst(eqns, temp);
        delete_type(temp);
}
pair<string, type *> eqn(x→getName(), t→clone());
eqns.push_back(eqn);
return true;
```

Uses `apply_subst` 18b and `delete_type` 10a 10b.

Comment 2.1.21. This function extracts the actual type of a synonym. We may need to go through several redirections to get to the actual type.

19c ⟨unification body 18a⟩+≡

```
type * get_type_from_syn(type * in) {
        type * ret = in;
        while (ret→isSynonym())
                ret = dcast<type_synonym *>(ret)→getActual();
        return ret;
}
```

Defines:
`get_type_from_syn`, used in chunks 17b, 20, and 25a.

Comment 2.1.22. This function returns whether two types `tvn` and `t` are unifiable. If one of the two, say `tvn`, is a parameter, we will try extending `eqns` with the equation (`tvn` = `t`). Otherwise, we compare the tags and try to recursively unify the subtypes if the tags match.

20 ⟨unification body 18a⟩+≡

```
bool unify(vector<pair<string,type ∗> > &eqns, type ∗ tvn, type ∗ t) {
        ⟨unify::verbose 1 21b⟩
        if (tvn→isSynonym()) tvn = get_type_from_syn(tvn);
        if (t→isSynonym()) t = get_type_from_syn(t);
        ⟨unify::verbose 2 21c⟩

        bool ret = false;
        if (tvn→isParameter()) {
                type ∗ phitvn = getBinding(eqns, tvn)→clone();

                type ∗ phit = apply_subst(eqns, t);
                // if phitvn == tvn
                if (phitvn→isParameter()) {
                        if (tvn→getName() ≡ phitvn→getName()) {
                                ret = extend(eqns, tvn, phit);
                                delete_type(phit); delete_type(phitvn);
                                if (unify_verbose) cerr ≪ ret ≪ endl;
                                return ret;
                        }
                } else {
                        ret = unify(eqns, phitvn, phit);
                        delete_type(phit); delete_type(phitvn);
                        if (unify_verbose) cerr ≪ ret ≪ endl;
                        return ret;
                }
        }
        // switch place
        if (tvn→isParameter() ≡ false ∧ t→isParameter())
                return unify(eqns, t, tvn);

        ⟨unify::case of both non-parameters 21a⟩
        return true;
}
```

Defines:
`unify`, used in chunks 17b, 21a, 25a, and 26a.

Uses `apply_subst` 18b, `delete_type` 10a 10b, `extend` 18c, `getBinding` 18a, and `get_type_from_syn` 19c.

```
21a   ⟨unify::case of both non-parameters 21a⟩≡
        if (tvn→isParameter() ≡ false ∧ t→isParameter() ≡ false) {
              if (tvn→getTag() ≠ t→getTag()) return false;
              if (tvn→getTag() ≡ Tuple ∧ t→getTag() ≡ Tuple)
                    if (tvn→alphaCount() ≠ t→alphaCount()) {
                          if (unify_verbose) cerr ≪ false ≪ endl;
                          return false;
                    }
              // unify each component
              if (tvn→alphaCount() ≠ t→alphaCount()) {
                    cerr ≪ "Error in unification. Argument counts don't match.\n";
                    cerr ≪ "tvn = " ≪ tvn→getName() ≪ endl;
                    cerr ≪ "  t = " ≪ t→getName() ≪ endl;
                    assert(false);
              }
              for (int i=0; i≠tvn→alphaCount(); i++) {
                    bool r = unify(eqns,tvn→getAlpha(i),t→getAlpha(i));
                    if (r ≡ false) return false;
              }
        }
      Uses unify 20.
```

Comment 2.1.23. We print out some information to help debugging.

```
21b   ⟨unify::verbose 1 21b⟩≡
        if (unify_verbose)
              cerr ≪ "Unifying " ≪ tvn→getName() ≪ " and " ≪ t→getName() ≪endl;
```

```
21c   ⟨unify::verbose 2 21c⟩≡
        if (unify_verbose) cerr ≪ "After transformation:\n";
        ⟨unify::verbose 1 21b⟩
```

2.1.2 Type Checking

Comment 2.1.24. The type-checking procedure implements the following algorithm. For more details on type checking and type inference, see, for example, [Mit96, Chap. 11].

$$\begin{aligned}
WT(C) &= \alpha \quad \text{where } \alpha \text{ is the declared signature of } C\\
WT(x) &= \begin{cases} \alpha & \text{if } WT(x) = \alpha \text{ has been established before;}\\ a & \text{otherwise; here, } a \text{ is a fresh parameter.} \end{cases}\\
WT((t_1, \ldots, t_n)) &= WT(t_1) \times \cdots \times WT(t_n)\\
WT(\lambda x.t) &= \begin{cases} \alpha \to \beta & \text{if } WT(t) = \beta \text{ and } x \text{ is free with relative type } \alpha \text{ in } t.\\ a \to \beta & \text{where } a \text{ is a parameter otherwise.} \end{cases}\\
WT((s\ t)) &= \beta\theta \quad \text{if } WT(s) = \alpha \to \beta,\ WT(t) = \gamma, \text{ and } \alpha \text{ and } \gamma \text{ are unifiable using } \theta.
\end{aligned}$$

The input term is not well-typed if any one of the WT calls on its subterms fails.

22a ⟨type checking actual 22a⟩≡

```
type * wellTyped2(term * t, vector<var_name> bvars, int scope) {
        type * ret = NULL;
        ⟨wellTyped2::case of t a constant 23a⟩
        ⟨wellTyped2::case of t a variable 24a⟩
        ⟨wellTyped2::case of t an application 25a⟩
        ⟨wellTyped2::case of t an abstraction 26b⟩
        ⟨wellTyped2::case of t a modal term 26c⟩
        ⟨wellTyped2::case of t a tuple 27a⟩
        return ret;
}
```

Defines:
`wellTyped2`, used in chunks 25–28.

Comment 2.1.25. We first look at some data structures. The vector `term_types` is used to store the inferred type for each subterm of the input term. The structure `var_name` is used to handle variables; see Comment 2.1.28 for more details.

22b ⟨type checking variables 22b⟩≡

```
vector<term_type> term_types;
struct var_name { int vname; string pname; };
```

Uses `term_type` 17b.

Comment 2.1.26. If the input term t is a constant, we find its signature α from the global constants repository (the function `get_signature` will halt with an error if t is unknown), rename all the parameters in α to obtain α' and then return α'. We need to rename parameters because some of the parameters in α may have been introduced (and constrained) up to this point in the type checking process. To illustrate, consider the following type declarations.

$$top : a \to \mathit{\Omega}$$
$$ind : a \to \mathit{\Omega}$$

The term $(top\ ind)$ is clearly well-typed. But the type checking procedure will fail if we do not first rename, say, the first parameter a because the unification procedure will fail when attempting to equate a and $a \to \mathit{\Omega}$.

23a ⟨wellTyped2::case of t a constant 23a⟩≡

```
if (t→isF() ∨ t→isD()) {
        if (t→isint) ret = new type(gInt);
        else if (t→isfloat) ret = new type(gFloat);
        else if (t→isChar()) ret = new type(gChar);
        else if (t→isString()) ret = new type(gString);
        else {
                ret = get_signature(t→cname);
                if (ret) { ret = ret→clone(); ret→renameParameters(); }
                else return NULL;
        }
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `get_signature` 153c, `isChar` 33a, `isD` 30a, `isF` 30a, `isString` 33a, and `renameParameters` 11c.

Comment 2.1.27. Each subterm is stored in `term_types` the moment its type is inferred. These entries may be updated later on when parameters get instantiated further. See Comment 2.1.29.

23b ⟨wellTyped2::save n return 23b⟩≡

```
term_type res; res.first = t; res.second = ret;
term_types.push_back(res);
if (t→ptype) delete_type(t→ptype);
t→ptype = ret→clone();
return ret;
```

Uses `delete_type` 10a 10b and `term_type` 17b.

Comment 2.1.28. To determine the type of a variable x, we need to know two things:

1. Is it a bound or a free variable?
2. Has it occurred before?

If x is a bound variable that has occurred previously, we just recycle the previously computed type. Else if x is a bound variable that has not occurred previously, we use the parameter name that has been assigned earlier to create a new parameter. (See Comment 2.1.32.) Otherwise, if x is free, we check (in `term_types`) to see whether a type for x has been inferred earlier. If so, we return the inferred type. Otherwise, we create a new parameter with a new parameter name.

24a ⟨wellTyped2::case of t a variable 24a⟩≡

```
if (t→isVar()) {
        if (t→cname ≡ iWildcard) {
                ret = new type_parameter();
                ⟨wellTyped2::save n return 23b⟩
        }
        unint start = 0;
        for (int i=(int)bvars.size()-1; i≠-1; i--)
                if (t→cname ≡ bvars[i].vname) {
                        start = scope;
                        ⟨variable case::lookup previous occurrence 24b⟩
                        ret = new type_parameter(bvars[i].pname);
                        ⟨wellTyped2::save n return 23b⟩
                }
        ⟨variable case::lookup previous occurrence 24b⟩
        ret = new type_parameter();
        ⟨wellTyped2::save n return 23b⟩
}
if (t→tag ≡ SV) {
        for (unint j=0; j≠term_types.size(); j++)
                if (term_types[j].first→tag ≡ SV) {
                        if (t→cname ≡ term_types[j].first→cname) {
                                ret = term_types[j].second→clone();
                                ⟨wellTyped2::save n return 23b⟩
                        }
                }
        ret = new type_parameter();
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `iWildcard` 145 and `isVar` 30a.

24b ⟨variable case::lookup previous occurrence 24b⟩≡

```
for (unint j=start; j≠term_types.size(); j++)
        if (term_types[j].first→isVar())
                if (t→cname ≡ term_types[j].first→cname) {
                        ret = term_types[j].second→clone();
                        ⟨wellTyped2::save n return 23b⟩
                }
```

Uses `isVar` 30a.

Comment 2.1.29. If the input term is an application of the form $(s\,t)$, we first infer the types of s and t separately. Assuming the type of s has the form $\alpha \rightarrow \beta$, we then attempt to unify α with γ, the type of t. If there exists a θ that unifies the two, we can then return $\beta\theta$ as the type for $(s\,t)$. We also update entries in `term_types` with θ to reflect new knowledge. The variable `vlength` keeps track of the part of `term_types` we can safely change.

25a ⟨wellTyped2::case of t an application 25a⟩≡

```
if (t→isApp()) {
        unsigned int vlength = term_types.size();
        type * t1 = wellTyped2(t→lc(), bvars, scope);
        if (t1→isSynonym()) t1 = get_type_from_syn(t1);
        ⟨wellTyped2::application::t1 should have right form 25b⟩
        type * t2 = wellTyped2(t→rc(), bvars, scope);
        if (¬t2) { printErrorMsg(t→rc()); return NULL; }

        vector<pair<string, type *> > slns;
        bool result = unify(slns, t1→getAlpha(0), t2);
        if (¬result) { ⟨wellTyped2::application::error reporting2 26a⟩ }
        ret = apply_subst(slns, t1→getAlpha(1));

        for (unint i=vlength; i≠term_types.size(); i++) {
                type * temp = term_types[i].second;
                term_types[i].second = apply_subst(slns, temp);
                delete_type(temp);
        }
        for (unint j=0; j≠slns.size(); j++) delete_type(slns[j].second);
        slns.clear();
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `apply_subst` 18b, `delete_type` 10a 10b, `get_type_from_syn` 19c, `isApp` 30a, `lc` 30e, `printErrorMsg` 27b, `rc` 30e, `unify` 20, and `wellTyped2` 22a.

Comment 2.1.30. The type `t1` should be a function type. If this is not the case but `t1` is a parameter, we can rescue the situation by making `t1` a type of the form $a \to b$, where both a and b are parameters. (This is equivalent to saying that s has type c, and that $c = a \to b$.) If `t1` is not a parameter and not a function type, we have a typing error.

25b ⟨wellTyped2::application::t1 should have right form 25b⟩≡

```
if (¬t1) { printErrorMsg(t→lc()); return NULL; }

if (¬t1→isAbstract() ∧ t1→isParameter()) {
        type * temp = t1;
        t1 = new type_abs(temp, new type_parameter());
        term_types[term_types.size()-1].second = t1;
}
if (¬t1→isAbstract()) {
        int osel = getSelector();
        setSelector(STDERR); ioprint("*** Error: ");
        t→lc()→print(); ioprint(" : "); ioprintln(t1→getName());
        ioprintln(" does not have function type.");
        setSelector(osel);
        return NULL;
}
```

Uses `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, `lc` 30e, `printErrorMsg` 27b, and `setSelector` 164 165.

Comment 2.1.31. Given $s : \alpha \to \beta$ and $t : \gamma$, the term $(s\,t)$ is not well typed if we cannot unify α and γ.

26a ⟨wellTyped2::application::error reporting2 26a⟩≡

```
int osel = getSelector();
setSelector(STDERR); t→print(); ioprintln(" is not well typed.");
ioprint(t1→getAlpha(0)→getName()); ioprint(" and ");
ioprint(t2→getName()); ioprintln(" are not unifiable\n");
slns.clear();
unify_verbose = true;
unify(slns, t1→getAlpha(0), t2);
setSelector(osel); unify_verbose = false;
return NULL;
```

Uses `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, `setSelector` 164 165, and `unify` 20.

Comment 2.1.32. Given a lambda term $\lambda x.t$, the variable x is given a new parameter name (stored in `bvars`), and every occurrence of x in t will use the same parameter name afterwards.

The type checking procedure is simple. We first check the type of t. Then we find the relative type of x in t (recorded in `term_types`). If t does not contain x, then we just use the initially assigned parameter name to create a new parameter. If x has type α and t has type β, we return $\alpha \to \beta$.

26b ⟨wellTyped2::case of t an abstraction 26b⟩≡

```
if (t→isAbs()) {
        unint vlength = term_types.size();

        var_name tmp; tmp.vname = t→fields[0]→cname;
        tmp.pname = newParameterName();
        bvars.push_back(tmp);

        type ∗ t2 = wellTyped2(t→fields[1], bvars, vlength);
        if (¬t2) { printErrorMsg(t); return NULL; }

        type ∗ vt = NULL;
        for (unint i=vlength; i≠term_types.size(); i++)
                if (term_types[i].first→isVar(t→fields[0]→cname))
                        { vt = term_types[i].second→clone(); break; }
        if (vt ≡ NULL) { vt = new type_parameter(tmp.pname); }

        ret = new type_abs(vt, t2→clone());
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `isAbs` 30a, `isVar` 30a, `printErrorMsg` 27b, and `wellTyped2` 22a.

Comment 2.1.33. We now look at modal terms. Given $\Box_i t$, if we can infer t has type α, then we can infer $\Box_i t$ has type α.

26c ⟨wellTyped2::case of t a modal term 26c⟩≡

```
if (t→isModal()) {
        type ∗ ret = wellTyped2(t→fields[0], bvars, scope);
        if (¬ret) { printErrorMsg(t); return NULL; }
        ret = ret→clone();
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `isModal` 30a, `printErrorMsg` 27b, and `wellTyped2` 22a.

Comment 2.1.34. The case for tuples is easy. We just infer the types of each component and then put them together.

27a ⟨wellTyped2::case of t a tuple 27a⟩≡

```
if (t→isProd()) {
        ret = new type_tuple;
        for (unsigned int i=0; i≠t→fieldsize; i++) {
                type ∗ ti = wellTyped2(t→fields[i], bvars, scope);
                if (¬ti) { printErrorMsg(t); return NULL; }
                ret→addAlpha(ti→clone());
        }
        ⟨wellTyped2::save n return 23b⟩
}
```

Uses `isProd` 30a, `printErrorMsg` 27b, and `wellTyped2` 22a.

27b ⟨type checking subsidiary functions 27b⟩≡

```
void printErrorMsg(term ∗ t) {
        int osel = getSelector();
        setSelector(STDERR); t→print();
        ioprintln(" is not well typed."); setSelector(osel);
}
```

Defines:
  `printErrorMsg`, used in chunks 25–28.
Uses `getSelector` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

Comment 2.1.35. This is a function written for debugging purposes. It prints out the contents of `term_types`.

27c ⟨type checking subsidiary functions 27b⟩+≡

```
void print_term_types() {
        int osel = getSelector(); setSelector(STDOUT);
        ioprintln(" *** ");
        for (unint i=0; i≠term_types.size(); i++) {
                term_types[i].first→print();
                ioprint(" : "); ioprintln(term_types[i].second→getName());
        }
        setSelector(osel);
}
```

Uses `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

Comment 2.1.36. We need to free up the memory occupied by the intermediate types inferred for the subterms.

27d ⟨type checking subsidiary functions 27b⟩+≡

```
void cleanup_term_types() {
        // print_term_types();
        for (unint i=0; i≠term_types.size(); i++)
                delete_type(term_types[i].second);
        term_types.clear();
}
```

Defines:
  `cleanup_term_types`, used in chunk 28a.
Uses `delete_type` 10a 10b.

Comment 2.1.37. The function `wellTyped` is a wrapper around the actual type-checking procedure `wellTyped2`.

28a ⟨type checking 28a⟩≡

```
#include <string>
#include <vector>
#include "global.h"
#include "terms.h"

⟨type checking variables 22b⟩
⟨type checking subsidiary functions 27b⟩
⟨type checking actual 22a⟩

type ∗ wellTyped(term ∗ t) {
        vector<var_name> bvars;
        type ∗ ret = wellTyped2(t, bvars, 0);
        if (¬ret) { printErrorMsg(t); return NULL; }
        ret = ret→clone();
        cleanup_term_types();
        return ret;
}
```

Defines:
`wellTyped`, used in chunks 17b and 149a.
Uses `cleanup_term_types` 27d, `printErrorMsg` 27b, and `wellTyped2` 22a.

Comment 2.1.38. The following is a version of `wellTyped` that returns both the type of the term being checked and the type of each subterm computed. The latter is needed for checking `typeof` side conditions on statements.

28b ⟨type checking 28a⟩+≡

```
pair<type ∗, vector<term_type> > mywellTyped(term ∗ t) {
        pair<type ∗, vector<term_type> > res;
        vector<var_name> bvars;
        type ∗ ret = wellTyped2(t, bvars, 0);
        if (¬ret) { printErrorMsg(t); res.first = NULL; return res; }
        ret = ret→clone();
        res.first = ret; res.second = term_types;
        term_types.clear();
        return res;
}
```

Defines:
`mywellTyped`, used in chunks 17b and 105.
Uses `printErrorMsg` 27b, `term_type` 17b, and `wellTyped2` 22a.

2.2 Terms

2.2.1 Term Representation

Comment 2.2.1. We use a standard approach to represent terms. A term is a graph of nodes, where each node is a term-schema as defined. One possible optimization is to distinguish between boxed and unboxed fields [Pey87, pg. 190]. For a discussion on term representations, see [Pey87, Chap. 10].

Comment 2.2.2. A term schema can be any one of the following: a syntactical variable (SV), a variable (V), a function symbol (F), a data constructor (D), an application (APP), an abstraction (ABS), a product (PROD) or a modal term (MOD). This information is recorded in `tag`.

29a ⟨term::type defs 29a⟩≡

```
enum kind { SV, V, F, D, APP, ABS, PROD, MODAL };
```

29b ⟨term parts 29b⟩≡

```
kind tag;
```

Comment 2.2.3. Syntactic variables, variables, functions and data constructors have names. For efficiency considerations, we use integers to represent names. (See Comment 5.0.41 for the mappings.) Modal terms have indices.

29c ⟨term parts 29b⟩+≡

```
int cname;
char modality;
type ∗ ptype;
```

29d ⟨term init 29d⟩≡

```
cname = -5;
modality = -5;
ptype = NULL;
```

29e ⟨heap term init 29e⟩≡

```
ret→cname = -5;
ret→modality = -5;
ret→ptype = NULL;
```

29f ⟨term clone parts 29f⟩≡

```
ret→cname = cname;
if (tag ≡ MODAL) ret→modality = modality;
// if (ptype) ret->ptype = ptype->clone();
```

29g ⟨term replace parts 29g⟩≡

```
cname = t→cname;
if (t→tag ≡ MODAL) modality = t→modality;
// if (t->ptype) ptype = t->ptype->clone();
```

Comment 2.2.4. Terms with names are called atomic terms. Terms that does not have names are called composite terms.

30a ⟨term::function declarations 30a⟩≡
```
bool isF() { return (tag ≡ F); }
bool isF(int code) { return (tag ≡ F ∧ cname ≡ code); }
bool isApp() { return (tag ≡ APP); }
bool isD() { return (tag ≡ D); }
bool isD(int code) { return (tag ≡ D ∧ cname ≡ code); }
bool isVar() { return (tag ≡ V); }
bool isVar(int v) { return (tag ≡ V ∧ cname ≡ v); }
bool isAbs() { return (tag ≡ ABS); }
bool isProd() { return (tag ≡ PROD); }
bool isModal() { return (tag ≡ MODAL); }
```
Defines:

`isAbs`, used in chunks 26b, 62c, 69a, and 105.
`isApp`, used in chunks 25a, 32, 33a, 36, 55b, 56a, 58, 60a, 73, 75b, 78b, 80, 89–91, 105, 159c, 160, and 163a.
`isD`, used in chunks 23a, 33a, 36, 56a, 58, 62d, 64b, 65, 67, 68, 75c, 78c, 82d, 105, 159c, and 160.
`isF`, used in chunks 23a, 32f, 35, 55b, 56a, 58, 60a, 62b, 68, 69b, 73, 75, 78, 80, 83e, 85b, 87, 105, and 163a.
`isModal`, used in chunks 26c, 55c, 80, and 105.
`isProd`, used in chunks 27a, 63a, and 105.
`isVar`, used in chunks 24, 26b, 62, 69b, 72a, 77b, and 105.

Comment 2.2.5. The parameters `tag` and `kind` does not have default values. They are initialized in the constructor code with pass-in values.

Comment 2.2.6. Application, abstraction and product terms have subterms. These are captured in the vector `fields`.

30b ⟨term vector parts 30b⟩≡
```
// vector<term *> fields;
term ∗ fields[10];
unint fieldsize;
```

30c ⟨term init 29d⟩+≡
```
fieldsize = 0;
```

30d ⟨heap term init 29e⟩+≡
```
ret→fieldsize = 0;
```

30e ⟨term::function declarations 30a⟩+≡
```

term ∗ lc() { /∗assert(tag == APP);∗/ return fields[0]; }
term ∗ rc() { /∗assert(tag == APP);∗/ return fields[1]; }

void insert(term ∗ t) {
   fields[fieldsize] = t; fieldsize++;
   if (fieldsize > 10) assert(false);
}
```
Defines:

`insert`, used in chunks 12c, 33c, 34a, 41a, 49a, 55a, 58–60, 74, 80, 107a, 156c, and 158a.
`lc`, used in chunks 25, 32–36, 55b, 56a, 58–62, 65, 67–69, 71–80, 85, 87–89, 91b, 93b, 102, 105, 107a, 109b, 159c, 160, and 163a.
`rc`, used in chunks 25a, 33a, 35, 36, 55, 56, 58–62, 65, 67–69, 71–80, 85, 87, 88a, 91b, 93b, 102, 105, 107a, 109b, 159c, 160, and 163a.

Comment 2.2.7. Certain basic data constructors like numbers can best be dealt with in their original machine representations. (Otherwise, a lot of conversions from and to strings are needed.) The variable `num` replaces the `cname` field for numbers.

Cloning of `isfloat`, `isint`, `numi` and `numf` is done in the `clone()` procedure. We do not have to worry about them here.

31a ⟨term bool parts 31a⟩≡

```
bool isfloat, isint;
```

31b ⟨term parts 29b⟩+≡

```
long long int numi;
double numf;
```

31c ⟨term init 29d⟩+≡

```
isfloat = false; isint = false;
```

31d ⟨heap term init 29e⟩+≡

```
ret→isfloat = false; ret→isint = false;
```

31e ⟨term replace parts 29g⟩+≡

```
if (t→tag ≡ D) { isfloat = t→isfloat; isint = t→isint;
                 numi = t→numi; numf = t→numf; }
```

Comment 2.2.8. Sometimes we want to prevent a certain subterm from being modified. This is done by setting a `freeze` flag.

31f ⟨term bool parts 31a⟩+≡

```
bool freeze;
```

Defines:
`freeze`, used in chunks 31, 48a, 52, 71a, and 72a.

31g ⟨term init 29d⟩+≡

```
freeze = false;
```

Uses `freeze` 31f.

31h ⟨heap term init 29e⟩+≡

```
ret→freeze = false;
```

Uses `freeze` 31f.

31i ⟨term replace parts 29g⟩+≡

```
freeze = t→freeze;
```

Uses `freeze` 31f.

Comment 2.2.9. A term of the form $(t_1\ (t_2 \cdots (t_{n-1}\ t_n) \cdots))$ can be visualized to take on the shape of a spine. (Draw it!) The (leftmost) term t_1 is called the tip of the spine. At different places throughout a computation, we need to access the leftmost term in a nested application node, and the following two functions provide this service. The input `x` to the second function will get assigned the value $n - 1$. We currently perform a (linear) traversal down the spine. It is possible to make this go faster if necessary.

We cache the results in `spinetip` and `spinelength`.

32a ⟨term parts 29b⟩+≡

```
term * spinetip;
int spinelength;
int spine_time;
```

32b ⟨term init 29d⟩+≡

```
spinetip = NULL; spinelength = -1; spine_time = -5;
```

32c ⟨heap term init 29e⟩+≡

```
ret→spinetip = NULL; ret→spinelength = -1; ret→spine_time = -5;
```

Comment 2.2.10. All these values become obsolete on replacing.

32d ⟨term replace parts 29g⟩+≡

```
spinetip = NULL; spinelength = -1; spine_time = -5;
```

32e ⟨term::function definitions 32e⟩≡

```
term * term::spineTip() {
        if (spinetip ∧ spinelength > -1 ∧ spine_time ≡ltime) return spinetip;
        spine_time = ltime;
        if (tag ≠ APP) { spinetip = this; spinelength = 1; return spinetip; }
        spinelength = 2; spinetip = fields[0];
        while (spinetip→isApp())
                { spinetip = spinetip→fields[0]; spinelength++; }
        return spinetip;
}
term * term::spineTip(int & numarg) {
        if (tag ≠ APP) { numarg = 0; return this; }
        numarg = 1; term * p = fields[0];
        while (p→isApp()) { p = p→fields[0]; numarg++; }
        return p;
}
```

Defines:
`spineTip`, used in chunks 64, 82d, 83e, 87, 90, 111, and 163a.
Uses `isApp` 30a.

Comment 2.2.11. The following function checks whether the current term has the general form $((f\ t_1)\ t_2)$, where f is given as input. If `spinetip` has already been computed, we can do things slightly faster.

32f ⟨term::function definitions 32e⟩+≡

```
bool term::isFunc2Args() {
        if (spinetip ∧ spinelength ≡ 3 ∧ spinetip→isF()) return true;
        return (isApp() ∧ lc()→isApp() ∧ lc()→lc()→isF());
}
bool term::isFunc2Args(int f) {
        if (spinetip ∧ spinelength ≡ 3 ∧ spinetip→isF(f)) return true;
        return (isApp() ∧ lc()→isApp() ∧ lc()→lc()→isF(f));
}
```

Defines:
`isFunc2Args`, used in chunks 35, 58, 69b, 71b, 72a, 76–79, 90, 93b, and 111.
Uses `isApp` 30a, `isF` 30a, and `lc` 30e.

Comment 2.2.12. This function checks whether a term is a string.

33a ⟨term::function definitions 32e⟩+≡

```
bool term::isAString() {
        return (isApp() ∧ lc()→isApp() ∧ lc()→lc()→isD(iHash)
                ∧ lc()→rc()→isChar());
}
bool term::isChar() {
        if (isfloat ∨ isint) return false;
        return (tag ≡ D ∧ cname ≥ 2000 ∧ cname < 3000);
}
bool term::isString() {
        if (isfloat ∨ isint) return false;
        return (tag ≡ D ∧ strings.find(cname) ≠ strings.end());
}
```

Defines:
`isAString`, used in chunks 36a, 62d, 81a, and 111.
`isChar`, used in chunks 23a, 36a, and 111.
`isString`, used in chunks 23a and 111.
Uses `iHash` 145, `isApp` 30a, `isD` 30a, `lc` 30e, and `rc` 30e.

Comment 2.2.13. Constants that are rigid have the same meaning in each possible world. A term is rigid if every constant in it is rigid.

33b ⟨term::function definitions 32e⟩+≡

```
bool term::isRigid() {
        if (tag ≡ V ∨ tag ≡ D) return true;
        if (tag ≡ F) return is_rigid_constant(cname);
        if (tag ≡ ABS) return fields[1]→isRigid();
        if (tag ≡ MODAL) return fields[0]→isRigid();
        assert(tag ≡ PROD ∨ tag ≡ APP);
        for (unint i=0; i≠fieldsize; i++)
                if (¬fields[i]→isRigid()) return false;
        return true;
}
```

Comment 2.2.14. The following function creates a new term having the form $((f\ t_1)\ t_2)$ where f (given) is a function symbol of arity two. The arguments t_1 and t_2 needs to be initialized by the calling function.

33c ⟨terms.cc::local functions 33c⟩≡

```
term * newT2Args(kind k, int f) {
        term * ret = new_term(APP);
        ret→insert(new_term(APP)); ret→lc()→insert(new_term(k, f));
        return ret;
}
```

Defines:
`newT2Args`, used in chunks 59, 63a, 64a, 74, 89b, and 111.
Uses `insert` 30e, `lc` 30e, and `new_term` 40a.

Comment 2.2.15. The following function initializes the two arguments of a term created using `newT2Args`.

34a ⟨term::function declarations 30a⟩+≡

```
void initT2Args(term ∗ t1, term ∗ t2) {
    lc()→insert(t1); insert(t2);
}
```

Defines:
`initT2Args`, used in chunks 59, 63a, 64a, 74, and 89b.
Uses `insert` 30e and `lc` 30e.

Comment 2.2.16. The following function checks whether two terms are equal to each other. This is currently only used in debugging code.

34b ⟨term::function definitions 32e⟩+≡

```
bool term::equal(term ∗ t) {
        if (tag ≠ t→tag) return false;
        if (cname ≠ t→cname) return false;
        if (modality ≠ t→modality) return false;
        ⟨term schema::equal::numbers 34c⟩
        // unint size1 = fieldsize;
        // unint size2 = t->fieldsize;
        if (fieldsize ≠ t→fieldsize) return false;
        for (unint i=0; i≠fieldsize; i++)
                if (fields[i]→equal(t→fields[i]) ≡ false)
                        return false;
        return true;
}
```

Defines:
`equal`, used in chunks 56, 88a, 92c, 101a, and 111.

Comment 2.2.17. We treat numbers in a slightly peculiar way. We will equate an integer and a floating-point number (even though the types do not agree) if they are the same number. We do this because the internal arithmetic of Escher can add, subtract, multiply and divide integers and floating-point numbers to produce another floating-point number. See Comment 3.1.11.

34c ⟨term schema::equal::numbers 34c⟩≡

```
if (isint ∧ t→isint ∧ numi ≠ t→numi) return false;
if (isint ∧ t→isfloat ∧ (double)numi ≠ t→numf) return false;
if (isfloat ∧ t→isint ∧ numf ≠ (double)t→numi) return false;
if (isfloat ∧ t→isfloat ∧ numf ≠ t→numf) return false;
```

Comment 2.2.18. This is used for marking and printing redexes.

34d ⟨term bool parts 31a⟩+≡

```
bool redex;
```

34e ⟨term init 29d⟩+≡

```
redex = false;
```

34f ⟨heap term init 29e⟩+≡

```
ret→redex = false;
```

Comment 2.2.19. The variable `redex` does not really play a part during cloning and reusing.

Comment 2.2.20. A term is printed in the way it is represented. The redex (if one exists) is marked out using square brackets. Shared nodes are also marked with their reference count.

```
35  ⟨term::function definitions 32e⟩+≡
      extern const string pve;
      void term::print() {
              if (getSelector() ≡ SILENT) return;
              ⟨term schema::print strings 36a⟩
              ⟨term schema::print lists 36b⟩
              if (redex) ioprint(" [[[ ");
              ⟨term schema::print if-then-else 36c⟩
              // if (refcount > 1) ioprint("_s_");
              if (cname ≥ 5000) { ioprint(pve); ioprint(cname - 5000); }
              else if (cname > 0) ioprint(getString(cname));
              else if (isfloat) ioprint(numf);
              else if (isint) ioprint(numi);
              else if (isFunc2Args()) {
                      ioprint("("); lc()→lc()→print(); ioprint(" ");
                      lc()→rc()→print(); ioprint(" "); rc()→print(); ioprint(")");
              } else if (tag ≡ APP ∧ (lc()→isF(iSigma) ∨ lc()→isF(iPi))) {
                      if (lc()→isF(iSigma)) ioprint("\\exists ");
                      else ioprint("\\forall ");
                      rc()→fields[0]→print(); ioprint(".");
                      rc()→fields[1]→print();
              } else if (tag ≡ APP) {
                      ioprint("("); fields[0]→print(); ioprint(" ");
                      fields[1]→print(); ioprint(")");
              } else if (tag ≡ ABS) {
                      ioprint("\\"); fields[0]→print();
                      ioprint("."); fields[1]→print();
              } else if (tag ≡ PROD) {
                      int size = fieldsize;
                      if (size ≡ 0) { ioprint("()"); return; }
                      ioprint("(");
                      for (int i=0; i≠size-1; i++)
                           { fields[i]→print(); ioprint(","); }
                      fields[size-1]→print(); ioprint(")");
              } else if (tag ≡ MODAL) {
                      ioprint("["); ioprint(modality); ioprint("] ");
                      fields[0]→print();
              } else { ⟨print error handling 36d⟩ }
              if (redex) ioprint(" ]]] ");
      }
```

Uses `getSelector` 164 165, `getString` 147, `iPi` 145, `iSigma` 145, `ioprint` 164 165, `isF` 30a, `isFunc2Args` 32f, `lc` 30e, and `rc` 30e.

Comment 2.2.21. (Composite) strings are represented as lists of characters. Printing them as lists is not good for the eyes. What we do here is to collect the characters together and print a string as a string.

36a ⟨term schema::print strings 36a⟩≡

```
if (isAString()) {
        string temp = ""; temp += getString(lc()→rc()→cname)[1];
        term ∗ arg2 = rc();
        while (¬arg2→isD(iEmptyList)) {
                assert(arg2→lc()→rc()→isChar());
                temp += getString(arg2→lc()→rc()→cname)[1];
                arg2 = arg2→rc();
        }
        ioprint("\""); ioprint(temp); ioprint("\""); return;
}
```

Uses `getString` 147, `iEmptyList` 145, `ioprint` 164 165, `isAString` 33a, `isChar` 33a, `isD` 30a, `lc` 30e, and `rc` 30e.

Comment 2.2.22. We print a list in the syntactic sugar form.

36b ⟨term schema::print lists 36b⟩≡

```
if (isApp() ∧ lc()→isApp() ∧ lc()→lc()→isD(iHash)) {
        ioprint("["); lc()→rc()→print();
        term ∗ arg2 = rc();
        while (arg2→isD(iEmptyList) ≡ false) {
                ioprint(", ");
                if (arg2→isApp() ∧ arg2→lc()→isApp() ∧
                    arg2→lc()→lc()→isD(iHash))
                        { arg2→lc()→rc()→print(); arg2 = arg2→rc(); }
                else { arg2→print(); break; }
        }
        ioprint("]");
        return;
}
```

Uses `iEmptyList` 145, `iHash` 145, `ioprint` 164 165, `isApp` 30a, `isD` 30a, `lc` 30e, and `rc` 30e.

Comment 2.2.23. We print if-then-else statements in a more human-readable form here.

36c ⟨term schema::print if-then-else 36c⟩≡

```
if (isApp() ∧ lc()→cname ≡ iIte) {
        ioprint("if "); rc()→fields[0]→print();
        ioprint(" then "); rc()→fields[1]→print();
        /∗ ioprint("\n\t"); ∗/ ioprint(" else "); rc()→fields[2]→print();
        return;
}
```

Uses `iIte` 145, `ioprint` 164 165, `isApp` 30a, `lc` 30e, and `rc` 30e.

36d ⟨print error handling 36d⟩≡

```
cerr ≪ "Printing untagged term.\ttag = " ≪ tag ≪ endl;
assert(false);
```

Comment 2.2.24. In vertical printing, we print the current term vertically (with some indentation). Miscellaneous information about the individual subterms are also printed. This is a convenient way to look at sharing and other information associated with each node.

37a ⟨term::function definitions 32e⟩+≡

```
void term::printVertical(unint level) {
        if (getSelector() ≡ SILENT) return;
        ⟨print white spaces 37b⟩
        if (cname ≥ 5000) { ioprint(pve); ioprint(cname-5000); }
        else if (cname > 0)
                { ioprint(getString(cname)); ⟨print extra information 37c⟩ }
        else if (isfloat) { ioprint(numf); ⟨print extra information 37c⟩ }
        else if (isint) { ioprint(numi); ⟨print extra information 37c⟩ }
        else if (tag ≡ APP) {
                ioprint("("); ⟨print extra information 37c⟩
                fields[0]→printVertical(level+1);
                fields[1]→printVertical(level+1);
                ⟨print white spaces 37b⟩ ioprint(")\n");
        } else if (tag ≡ ABS) {
                ioprint("\\"); fields[0]→print(); ioprint(".");
                ⟨print extra information 37c⟩
                fields[1]→printVertical(level+1);
        } else if (tag ≡ PROD) {
                int size = fieldsize;
                if (size ≡ 0)
                      { ioprint("()"); ⟨print extra information 37c⟩ return; }
                ioprint("("); ⟨print extra information 37c⟩
                for (int i=0; i≠size-1; i++) {
                        fields[i]→printVertical(level+1); ioprint(",\n"); }
                fields[size-1]→printVertical(level+1);
                ⟨print white spaces 37b⟩ ioprint(")\n");
        } else if (tag ≡ MODAL) {
                assert(false);
        } else { ⟨print error handling 36d⟩ }
}
```

Defines:
`printVertical`, used in chunk 111.
Uses `getSelector` 164 165, `getString` 147, and `ioprint` 164 165.

37b ⟨print white spaces 37b⟩≡

```
for (unint i=0; i≠level; i++) ioprint(" ");
```

Uses `ioprint` 164 165.

37c ⟨print extra information 37c⟩≡

```
ioprint("\t\t");
if (refcount > 1) { ioprint("shared"); ioprint(refcount); }
ioprintln();
```

Uses `ioprint` 164 165, `ioprintln` 164 165, and `shared` 43e.

2.2.1.1 Constraints for Syntactic Variables

Comment 2.2.25. We have a (limited) syntax for specifying constraints on what sort of terms a syntactical variable can range over. (See the grammar for Escher.) Four types of constraints are supported at present. The constraint CVAR forces a syntactical variable to range over variables only; CCONST forces a syntactical variable to range over constants only. The constraint CEQUAL dictates that the value of one syntactical variable must be equal to the value of one other; The constraint CNOTEQUAL dictates that the value of one syntactical variable must not be equal to the value of one other. For details on how these constraints are implemented, see Comment 3.1.100.

38a ⟨term::definitions 38a⟩≡

```
#define CVAR 1
#define CCONST 2
#define CEQUAL 3
#define CNOTEQUAL 4
```

38b ⟨term::supporting types 38b⟩≡

```
struct condition { int tag; int svname; };
```

38c ⟨term parts 29b⟩+≡

```
condition ∗ cond; // only applies to SV
```

38d ⟨term init 29d⟩+≡

```
cond = NULL;
```

38e ⟨heap term init 29e⟩+≡

```
ret→cond = NULL;
```

38f ⟨term clone parts 29f⟩+≡

```
if (cond) { assert(tag ≡ SV);
            ret→cond = new condition;
            ret→cond→svname = cond→svname; ret→cond→tag = cond→tag; }
```

38g ⟨term replace parts 29g⟩+≡

```
if (cond) delete cond;
cond = t→cond;
```

2.2.2 Memory Management

Comment 2.2.26. We look at some memory management issues in this section. A naive scheme relying on `new` and `delete` is in use at the moment. It is not clear to the author whether a separate heap-allocating scheme would make the system go a whole lot faster.

Comment 2.2.27. We put wrappers around `new` and `delete` to collect some statistics. The procedure `mem_report` shows the total number of terms allocated and subsequently freed. This is used to check whether there is a memory leak.

39a ⟨term::memory management 39a⟩≡

```
extern void makeHeap();
extern term ∗ new_term(kind k);
extern term ∗ new_term(kind k, int code);
extern term ∗ new_term_int(int x);
extern term ∗ new_term_int(long long int x);
extern term ∗ new_term_float(float x);
extern void mem_report();
```

Uses `mem_report` 40b, `new_term` 40a, `new_term_float` 40a, and `new_term_int` 40a.

39b ⟨terms.cc::local functions 33c⟩+≡

```
#ifdef DEBUG_MEM
static long int allocated = 0;
static long int freed = 0;
#endif

#define HEAPSIZE 100000
term heap[HEAPSIZE];
term ∗ avail;

void makeHeap() {
     cout ≪ "Sizeof(term) = " ≪ sizeof(term) ≪ endl;
     cout ≪ "Sizeof(char) = " ≪ sizeof(char) ≪ endl;
     cout ≪ "Sizeof(short) = " ≪ sizeof(short) ≪ endl;
     cout ≪ "Sizeof(int) = " ≪ sizeof(int) ≪ endl;
     cout ≪ "Sizeof(bool) = " ≪ sizeof(bool) ≪ endl;
    avail = heap;
    for (int i=0; i≠HEAPSIZE-1; i++) {
        heap[i].next = &(heap[i+1]);
    }
    heap[HEAPSIZE-1].next = NULL;
}

term ∗ myalloc() {
    if (avail ≡ NULL) assert(false);
    term ∗ ret = avail; avail = avail→next;
    ⟨heap term init 29e⟩
    return ret;
}
inline void mydealloc(term ∗ p) { p→next = avail; avail = p; }
```

40a ⟨terms.cc::local functions 33c⟩+≡

```
term ∗ new_term(kind k) {
     term ∗ ret = myalloc(); ret→tag = k;
     return ret;
}

term ∗ new_term(kind k, int code) {
     term ∗ ret = myalloc();
     ret→tag = k;
     ret→cname = code;
     return ret;
}

term ∗ new_term_int(int x) {
    term ∗ ret = myalloc();
    ret→tag = D;
    ret→isint = true; ret→numi = x; return ret;
}

term ∗ new_term_int(long long int x) {
    term ∗ ret = myalloc();
    ret→tag = D;
    ret→isint = true; ret→numi = x; return ret;
}

term ∗ new_term_float(float x) {
     term ∗ ret = myalloc();
     ret→tag = D;
     ret→isfloat = true; ret→numf = x; return ret;
}
```

Defines:
- `new_term`, used in chunks 33c, 39a, 41a, 58, 59, 62–64, 67, 74, 76–78, 80, 93b, and 107a.
- `new_term_float`, used in chunks 39a, 41a, 65, 66, 68, 161, and 162.
- `new_term_int`, used in chunks 39a, 41a, 65, and 66.

40b ⟨terms.cc::local functions 33c⟩+≡

```
inline void delete_term(term ∗ x) { mydealloc(x); }

void mem_report() {
#ifdef DEBUG_MEM
        cout ≪ "\n\nReport from Memory Manager:\n";
        cout ≪ "\tAllocated " ≪ allocated ≪ endl;
        cout ≪ "\tFreed " ≪ freed ≪ endl;
        cout ≪ "\tUnaccounted " ≪ allocated - freed ≪ endl ≪ endl;
#endif
} // >>
```

Defines:
- `delete_term`, used in chunk 41b.
- `mem_report`, used in chunk 39a.

Comment 2.2.28. Cloning of a term with shared nodes will result in an identical term without shared nodes.

41a ⟨term::function definitions 32e⟩+≡

```
term ∗ term::clone() {
        term ∗ ret;
        if (isfloat) ret = new_term_float(numf);
        else if (isint) ret = new_term_int(numi);
        else if (tag ≥ SV ∧ tag ≤ D) ret = new_term(tag, cname);
        else ret = new_term(tag);
        ⟨term clone parts 29f⟩

        int size = fieldsize;
        for (int i=0; i≠size; i++) ret→insert(fields[i]→clone());
        return ret;
}
```

Uses `insert` 30e, `new_term` 40a, `new_term_float` 40a, and `new_term_int` 40a.

Comment 2.2.29. We explicitly free memory instead of relying on destructors. The function freememory must take node sharing into account. A term is in use while its reference count is still non-zero.

41b ⟨term::function definitions 32e⟩+≡

```
÷∗
void term::freememory() {
        refcount−−;
        ⟨freememory error checking 42a⟩
        if (refcount ≠ 0) return;
        if (ptype) { delete_type(ptype); }
        if (cond) delete cond;
        int size = fieldsize;
        for (int i=0; i≠size; i++) if (fields[i]) fields[i]→freememory();
        fieldsize = 0;
        delete_term(this);
}
∗÷
void term::freememory() {
        refcount−−;
        ⟨freememory error checking 42a⟩
        if (refcount ≠ 0) return;

        term ∗ p = this;
        delete_term(this);

        if (p→ptype) delete_type(p→ptype);
        if (p→cond) delete p→cond;

        int size = p→fieldsize;
        for (int i=0; i≠size; i++)
            if (p→fields[i]) p→fields[i]→freememory();
        p→fieldsize = 0;
}
```

Uses `delete_term` 40b and `delete_type` 10a 10b.

42a
```
⟨freememory error checking 42a⟩≡
  // if (refcount < 0) { setSelector(STDERR); print(); ioprintln();
  //                 ioprint("refcount = "); ioprintln(refcount); }
  assert(refcount ≥ 0);
```
Uses `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

Comment 2.2.30. This function overwrites the root of the current term with the input term *t*. We need to do this if the current node is shared (see §2.2.3) or when the current term is the root term (with no parent). The procedure is simple. The information on the root of *t* is copied, and all the child nodes of *t* are reused. We first `reuse` the child nodes of `t` because we could be replacing the current term with its children, in which case `t` can get deleted before we can reuse its child nodes if we are not careful.

42b
```
⟨term::function definitions 32e⟩+≡
  void term::replace(term ∗ t) {
        tag = t→tag;
        ⟨term replace parts 29g⟩
        int tsize = t→fieldsize;
        for (int i=0; i≠tsize; i++) t→fields[i]→reuse();

        int size = fieldsize;
        for (int i=0; i≠size; i++) if (fields[i]) fields[i]→freememory();
        // fields.resize(tsize);
        // copy(t->fields.begin(), t->fields.end(), fields.begin());
        fieldsize = t→fieldsize;
        for (int i=0; i≠tsize; i++)
          fields[i] = t→fields[i];
  }
```
Defines:
`replace`, used in chunks 52a, 54b, 56–59, 62a, 78d, 88b, 93b, 107a, and 111.
Uses `reuse` 43d.

2.2.3 Sharing of Nodes

Comment 2.2.31. We use reference counting to implement sharing of nodes.

43a ⟨term parts 29b⟩+≡
```
int refcount;
```

43b ⟨term init 29d⟩+≡
```
refcount = 1;
```

43c ⟨heap term init 29e⟩+≡
```
ret→refcount = 1;
```

Comment 2.2.32. A cloned object of a shared term would have `refcount` 1. Also, after replacing, the term retains its original `refcount` value.

43d ⟨term::function declarations 30a⟩+≡
```
term * reuse() { refcount++; return this; }
```
Defines:
`reuse`, used in chunks 42b, 53a, 63a, 64a, 69a, 71a, 74–76, 78–80, 87, and 95–97.

43e ⟨term::function declarations 30a⟩+≡
```
bool shared() { return (refcount > 1); }
```
Defines:
`shared`, used in chunks 37c and 53a.

Comment 2.2.33. A few notes on sharing. One of the biggest advantages of sharing is that common subexpressions need only be evaluated once. Sharing of nodes can, however, interfere with a few basic operations in Escher.

Firstly, I believe the operation of checking for free-variable capture, a test we need to do frequently during pattern matching (see §3.1.3) and term substitution (see §2.2.6), cannot be done efficiently if a variable that occurs both free and bound in a term is shared.

Second, sharing of nodes is not always safe. Some statements in the booleans module, especially the ones that support logic programming (see for example Comment 3.1.14), can potentially change shared nodes in destructive ways. The extensive use of such sharing-unfriendly statements in Escher is the primary reason I gave up on sharing.

In the absence of sharing, the computational saving that can be obtained from common subexpression evaluation can be achieved using (intelligent) caching.

Having said all that, sharing does have at least one important use in our interpreter; see Comment 3.1.82.

Comment 2.2.34. The following function, which is no longer in use, provides a way to unshare shared nodes using the side effect of the cloning operation (see Comment 2.2.28). Time complexity: the entire term needs to be traversed; nodes that are not traversed by this function will be traversed by `clone`.

43f ⟨term::function declarations 30a⟩+≡
```
void unshare(term * parent, unint id);
```

Comment 2.2.35. We should `assert(parent)` because a shared node, by definition, have at least two parents.

44a
```
⟨term::function definitions 32e⟩+≡
  void term::unshare(term ∗ parent, unint id) {
        if (refcount > 1) {
              assert(parent); term ∗ temp = clone();
              parent→fields[id]→freememory(); parent→fields[id] = temp;
              return; }
        int size = fieldsize;
        for (int i=0; i≠size; i++) fields[i]→unshare(this, i);
  }
```

2.2.4 Free and Bound Variables

Comment 2.2.36. One must be careful when dealing with free and bound variables. This is something that is not difficult to get right, but incredibly easy to get wrong! So please pay some attention.

Definition 2.2.37. An occurrence of a variable x in a term is *bound* if it occurs within a subterm of the form $\lambda x.t$.

Definition 2.2.38. An occurrence of a variable in a term is *free* if it is not a bound occurrence.

Fact 2.2.39. *A variable is free in a term iff it has a free occurrence.*

Comment 2.2.40. The following function returns all the free variables inside a term. It is assumed that we have called `labelVariables` on the term to initialize all the labels and binding labels.

Comment 2.2.41. Computed free variables are cached in the array `frvars`. The flag `freevars_computed` tells us whether `frvars` has been initialized. An array instead of a set is used to store the free variables. This means free variables with multiple occurrences will be recorded multiple times. We need to record multiple occurrences; see Comment 4.1.11. Further, using an array is faster than using a set.

44b
```
⟨term bool parts 31a⟩+≡
  bool freevars_computed;
```

44c
```
⟨term parts 29b⟩+≡
  int time_computed;
```

44d
```
⟨term vector parts 30b⟩+≡
  int frvars[20];
  int frvarsize;
```

44e
```
⟨term init 29d⟩+≡
  frvarsize = 0;
  freevars_computed = false;
  time_computed = -5;
```

44f
```
⟨heap term init 29e⟩+≡
  ret→frvarsize = 0;
  ret→freevars_computed = false;
  ret→time_computed = -5;
```

Comment 2.2.42. These values become obsolete on replacing.

45a ⟨term replace parts 29g⟩+≡

```
frvarsize = 0;
freevars_computed = false;
time_computed = -5;
```

45b ⟨term::function definitions 32e⟩+≡

```
void term::getFreeVars() {
        if (freevars_computed ∧ time_computed ≡ ltime) return;

        frvarsize = 0;
        freevars_computed = true; time_computed = ltime;

        if (tag ≡ D ∨ tag ≡ F) return;
        if (tag ≡ V) { frvars[0] = cname; frvarsize = 1; return; }
        if (tag ≡ ABS) {
                fields[1]→getFreeVars();

                for (int i=0; i≠fields[1]→frvarsize; i++) {
                    if (fields[1]→frvars[i] ≡ fields[0]→cname) continue;
                    frvars[frvarsize] = fields[1]→frvars[i];
                    frvarsize++;
                }
                assert(frvarsize ≤ 20);
                return;
        }
        for (unint i=0; i≠fieldsize; i++) {
                fields[i]→getFreeVars();
                for (int j=0; j≠fields[i]→frvarsize; j++) {
                    if (j > 0 ∧ fields[i]→frvars[j] ≡ fields[i]→frvars[j-1])
                        continue;
                    frvars[frvarsize] = fields[i]→frvars[j];
                    frvarsize++;
                }
                assert(frvarsize ≤ 20);
        }
        return;
}
```

Defines:
`getFreeVars`, used in chunks 47b, 48b, 105, and 111.

Comment 2.2.43. For terms that stay unchanged throughout the whole computation (e.g. program statements), freeness checking of variables can be done (slightly) more efficiently by flagging each bound variable in the term directly up front. This is achieved using the following function `labelStaticBoundVars()`.

Comment 2.2.44. We first look at the `free` parameter. To ensure safe use, the `free` parameter is only valid if the `validfree` parameter is true. (The function `labelStaticBoundVars` is responsible for setting this latter parameter. Its value will get set to `false` during cloning and replacing.)

45c ⟨term bool parts 31a⟩+≡

```
bool free;
bool validfree;
```

46a
```
⟨term init 29d⟩+≡
  validfree = false;
```

46b
```
⟨heap term init 29e⟩+≡
  ret→validfree = false;
```

Comment 2.2.45. If the whole term t on which `labelBoundVars` is called is to be cloned, then the existing value of the `free` parameter would remain correct. However, if only a subterm t_1 of t is to be cloned, then some variables that are bound in t can become free in t_1. Variables that are free in t would remain free in t_1 though. However, if t (respectively t_1) is then subsequently substituted into another term (using the mechanism of syntactical variables), then free variables in t (respectively t_1) can become bound. For all these reasons, we will not attempt to recycle values of `free` parameters during cloning and replacing.

46c
```
⟨term clone parts 29f⟩+≡
  ret→validfree = false;
```

Comment 2.2.46. Ditto for replacing. Free variables can become bound after replacing while bound variables remain bound after replacing. The trouble here is that we do not really want to traverse the input graph to label the variables. At present, we only use the `free` parameters inside the head of a program statement during pattern matching. We will just mark in the `replace` code that proper handling of the `free` parameter is not yet implemented.

46d
```
⟨term replace parts 29g⟩+≡
  validfree = false;
```

46e
```
⟨term::function declarations 30a⟩+≡
  bool isFree() { assert(tag ≡ V ∧ validfree); return free; }
```

Defines:
`isFree`, used in chunks 53h, 96, 97, and 101c.

Comment 2.2.47. A straightforward tree traversal is used to label the bound variables. Bound variables inside a lambda term are marked before the free variables. Hence the way labelling is done inside the `(tag == V)` case.

46f
```
⟨term::function definitions 32e⟩+≡
  void term::unmarkValidfree() {
        validfree = false;
        for (unint i=0; i≠fieldsize; i++) fields[i]→unmarkValidfree();
  }
  void term::labelStaticBoundVars() {
        if (tag ≡ F ∨ tag ≡ D ∨ tag ≡ SV) return;
        if (tag ≡ V) { if (¬validfree) { validfree = true; free = true; }
                    return; }
        if (tag ≡ ABS) {
              fields[0]→validfree = true; fields[0]→free = false;
              fields[1]→labelBound(fields[0]→cname);
              fields[1]→labelStaticBoundVars();
              return;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++) fields[i]→labelStaticBoundVars();
  }
```

Defines:
`labelStaticBoundVars`, used in chunks 52a, 54b, 89b, 111, and 141d.
Uses `labelBound` 47a.

47a ⟨term::function definitions 32e⟩+≡

```
void term::labelBound(int x) {
        if (tag ≡ F ∨ tag ≡ D ∨ tag ≡ SV) return;
        if (tag ≡ V) { if (cname ≡ x) { validfree = true; free = false; }
                    return; }
        if (tag ≡ ABS) { fields[1]→labelBound(x); return; }
        int size = fieldsize;
        for (int i=0; i≠size; i++) fields[i]→labelBound(x);
}
```

Defines:
`labelBound`, used in chunks 46f and 111.

Comment 2.2.48. The functions `isFree` and `isFreeInside` discussed above allows one to check whether a subterm s of a term t occurs free inside t. Some times we want to check whether a variable x has a free occurrence inside another term. The following functions allow us to do that. Some occurrences of the input variable could be bound. We return upon seeing the first free occurrence the input variable.

There are two versions of this function. The first, `occursFree`, uses `getFreeVars` to compute all the free variables in a term and then check whether `var` is inside this set. If `occursFree` is called repeatedly by the same term, this caching of computed free variables is beneficial. The second, `occursFreeNaive`, performs a simple traversal of the term to check whether `var` occurs free.

47b ⟨term::function definitions 32e⟩+≡

```
bool term::occursFree(int var) {
        getFreeVars();
        for (int i=0; i≠frvarsize; i++) if (frvars[i] ≡ var) return true;
        return false;
}
```

Defines:
`occursFree`, used in chunks 69b, 72a, 73, 77b, 108b, and 111.
Uses `getFreeVars` 45b.

47c ⟨term::function definitions 32e⟩+≡

```
bool term::occursFreeNaive(int var) {
        vector<int> boundv; return occursFreeNaive(var, boundv);
}
```

Defines:
`occursFreeNaive`, used in chunks 48a, 72a, and 111.

48a ⟨term::function definitions 32e⟩+≡

```
bool term::occursFreeNaive(int var, vector<int> boundv) {
        if (tag ≡ F ∨ tag ≡ D) return false;
        if (tag ≡ V) {
                if (freeze) return false;
                if (cname ≡ var ∧ inVector(cname,boundv)≡ false) return true;
                return false; }
        if (tag ≡ ABS) {
                boundv.push_ back(fields[0]→cname);
                return fields[1]→occursFreeNaive(var, boundv);
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                if (fields[i]→occursFreeNaive(var, boundv)) return true;
        return false;
}
```

Uses `freeze` 31f, `inVector` 159a, and `occursFreeNaive` 47c.

Comment 2.2.49. This function checks whether any free variable inside the calling term is captured by at least one of the bounded variables. The index of the captured variable is recorded in `captd`. We store pointers to binding abstraction terms instead of strings for two reasons. First, we sometimes need to change the name of a binding variable when a free variable is captured. This happens, for example, during term substitution. Having a pointer to the abstraction term allows us to jump straight to the offending term. Second, in terms of memory usage, storing pointers to terms is cheaper. If we want to use a set instead of a vector to store the binding variables (maybe for efficiency reasons), it is easy to put a wrapper around `term_schema *` and define a pointer p to be less than q iff `p->fields[0]->name < q->fields[0]->name`.

48b ⟨term::function definitions 32e⟩+≡

```
bool term::captured(vector<term *> & bvars, int & captd) {
        if (bvars.empty()) return false;
        getFreeVars();

        int bsize = bvars.size();
        for (int i=0; i≠frvarsize; i++)
                for (int j=0; j≠bsize; j++)
                        if (frvars[i] ≡ bvars[j]→fields[0]→cname) {
                                captd = j; return true; }
        return false;
}
```

Defines:
  `captured`, used in chunks 54a, 102a, 103d, and 111.
Uses `getFreeVars` 45b.

Comment 2.2.50. For small terms, the use of vector for both `frvars` and `bvars` is probably okay. For larger terms, the use of set (red-black trees) could be much better.

Comment 2.2.51. The following function collects in a multiset all the bound variables in a term.

49a ⟨term::function definitions 32e⟩+≡

```
void term::collectLambdaVars(multiset<int> & ret) {
        if (tag ≥ SV ∧ tag ≤ D) return;
        if (tag ≡ ABS) {
                ret.insert(fields[0]→cname);
                fields[1]→collectLambdaVars(ret); return;
        }
        for (unint i=0; i≠fieldsize; i++)
                fields[i]→collectLambdaVars(ret);
}
```

Uses `insert` 30e.

2.2.5 Variable Renaming

Comment 2.2.52. Different forms of variable renaming are required in performing computations. We discuss these operations in this section.

Comment 2.2.53. This function renames all occurrences of a variable `var1` inside the current term to `var2`. Note that both free and bound occurrences are renamed. This is okay since the function is only called (sensibly) as a subroutine by the other variable-renaming functions in this section.

49b ⟨term::function definitions 32e⟩+≡

```
void term::rename(int var1, int var2) {
        if (tag ≡ SV ∨ tag ≡ F ∨ tag ≡ D) return;
        if (tag ≡ V) { if (cname ≡ var1) cname = var2; return; }
        int size = fieldsize;
        for (int i=0; i≠size; i++) fields[i]→rename(var1, var2);
}
```

Defines:
  `rename`, used in chunks 50 and 111.

Comment 2.2.54. This function renames one particular lambda variable in a term. This is used in term substitutions in the case when a free variable capture occurs.

50 ⟨term::function definitions 32e⟩+≡

```
void term::renameLambdaVar(int var1, int var2) {
        freevars_computed = false;
        if (tag ≡ SV ∨ tag ≡ V ∨ tag ≡ F ∨ tag ≡ D) return;
        if (tag ≡ ABS) {
                if (fields[0]→cname ≡ var1) {
                        fields[0]→cname = var2;
                        fields[1]→rename(var1, var2);
                }
                // if lambda variables are distinct, this is not needed
                fields[1]→renameLambdaVar(var1, var2);
                return;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                fields[i]→renameLambdaVar(var1, var2);
}
```

Defines:
`renameLambdaVar`, used in chunks 54a and 111.

Uses `rename` 49b.

2.2.6 Term Substitution

Definition 2.2.55. [Llo03, pg. 55] A *t*erm substitution is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$ where each x_i is a variable, each t_i is a term distinct from x_i, and $x_1, \ldots, x_n$ are distinct.

Comment 2.2.56. Each pair x_i/t_i is represented as a structure as follows.

51 ⟨term::type defs 29a⟩+≡

```
struct substitution {
        int first;
        term * second;
        substitution() { second = NULL; }
        substitution(int v, term * t) { first = v; second = t; }
};
```

Defines:
`substitution`, used in chunks 52, 54b, 57a, 69, 71a, 75a, 78d, 79b, 87, 88a, 99–101, 107–111, and 140.

Definition 2.2.57. [Llo03, pg. 56] Let t be a term and $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ a term substitution. The instance $t\theta$ of t by θ is the well-formed expression defined as follows.

1. If t is a variable x_i for some $i \in \{1, \ldots, n\}$, then $x_i\theta = t_i$.
   If t is a variable y distinct from all the x_i, then $y\theta = y$.
2. If t is a constant C, then $C\theta = C$.
3. If t is an abstraction $\lambda x_i.s$, for some $i \in \{1, \ldots, n\}$, then

   $$(\lambda x_i.s)\theta = \lambda x_i.(s\{x_1/t_1, \ldots, x_{i-1}/t_{i-1}, x_{i+1}/t_{i+1}, \ldots, x_n/t_n\}).$$

   If t is an abstraction $\lambda y.s$, where y is distinct from all the x_i,

   (a) if for some $i \in \{1, \ldots, n\}$, y is free in t_i and x_i is free in s, then

   $$(\lambda y.s)\theta = \lambda z.(s\{y/z\}\theta)$$

   where z is a new variable.

   (b) else $(\lambda y.s)\theta = \lambda y.(s\theta)$;
4. If t is an application $(u\ v)$, then $(u\ v)\theta = (u\theta\ v\theta)$.
5. If t is a tuple $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)\theta = (t_1\theta, \ldots, t_n\theta)$.

Comment 2.2.58. Term substitutions are performed by the function `subst`. There are two versions of it, one deals with singleton sets, the other with non-singleton sets. In both cases, real work is done by the function `subst2`.

Comment 2.2.59. A single traversal of the tree achieves the desired parallel-instantiation-of-variables effect.

Comment 2.2.60. Given t and θ, the function `subst` will handle the special case where t is a variable (and thus free) or a syntactical variable. All other cases are handled by `subst2`. Before calling `subst2`, we call `labelStaticBoundVars` to label the variables. The `free` values computed are safe for use here because they are read only once by `subst2` and changes introduced by `subst2` are all localized on the spots where free variables live in the term.

Comment 2.2.61. Pointers to terms in `subs` are all pointers to subterms in an existing structure that will be deleted after the term substitution. For that reason, these pointers can be safely reused once, but not more than that.

For the special case where the current term is a variable or a syntactical variable, we need to make the term replacement *i*n place using `replace`.

```
52a ⟨term::function definitions 32e⟩+≡
  void term::subst(vector<substitution> & subs) {
        if (tag ≡ V ∨ tag ≡ SV) {
              if (freeze) return;
              int size = subs.size();
              for (int i=0; i≠size; i++)
                    if (cname ≡ subs[i].first) {
                          this→replace(subs[i].second); return; }
              return;
        }
        labelStaticBoundVars();
        vector<term ∗> bindingAbss;
        subst2(subs, bindingAbss, NULL);
        unmarkValidfree();
  }
```

Defines:
`subst`, used in chunks 54b, 57a, 69, 71a, 76c, 78d, 79b, 87–89, 92c, 111, and 140.
Uses `freeze` 31f, `labelStaticBoundVars` 46f, `replace` 42b, `subst2` 52b, and `substitution` 51.

Comment 2.2.62. All the complications in Definition 2.2.57 are in the abstraction case. Operationally, checking all those conditions every time we encounter an abstraction is expensive. We can perform these checks only when strictly necessary by delaying them until before we apply a substitution, that is, until we see a free variable in t that matches one of the variables in θ.

```
52b ⟨term::function definitions 32e⟩+≡
  void term::subst2(vector<substitution> & subs, vector<term ∗> bindingAbss,
                    term ∗∗ pointer) {
        if (tag ≡ SV) { if (freeze) return; ⟨subst2::case of SV 52c⟩ }
        if (tag ≡ V) { if (freeze) return; ⟨subst2::case of V 53h⟩ }
        if (tag ≡ F ∨ tag ≡ D) return;
        if (tag ≡ ABS) {
              if (fields[0]→tag ≡ SV)
                    fields[0]→subst2(subs, bindingAbss, &fields[0]);
              bindingAbss.push_back(this);
              fields[1]→subst2(subs, bindingAbss, &fields[1]);
              return;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++)
              fields[i]→subst2(subs, bindingAbss, &fields[i]);
  }
```

Defines:
`subst2`, used in chunks 52a, 54b, and 111.
Uses `freeze` 31f and `substitution` 51.

Comment 2.2.63. Term substitution is not formally defined for syntactical variables. It should behave like a free variable (see Comment 2.2.65), except that we do not have to worry about free variable capture.

```
52c ⟨subst2::case of SV 52c⟩≡
  int size = subs.size();
  for (int i=0; i≠size; i++)
        if (cname ≡ subs[i].first) { ⟨subst2::replace by ti 53a⟩ return; }
  return;
```

Comment 2.2.64. See the first part of Comment 2.2.61 for why we do what we do here. The parent pointer must exist because the case where it does not exist is handled by `subst`.

53a

```
⟨subst2::replace by ti 53a⟩≡
  assert(pointer);
  this→freememory();
  if (¬subs[i].second→noredex ∧ subs[i].second→shared())
        ∗pointer = subs[i].second→clone();
  else ∗pointer = subs[i].second→reuse();
```

Uses `reuse` 43d and `shared` 43e.

53b

```
⟨term bool parts 31a⟩+≡
  bool noredex;
```

53c

```
⟨term init 29d⟩+≡
  noredex = false;
```

53d

```
⟨heap term init 29e⟩+≡
  ret→noredex = false;
```

53e

```
⟨term clone parts 29f⟩+≡
  ret→noredex = noredex;
```

53f

```
⟨term replace parts 29g⟩+≡
  noredex = t→noredex;
```

53g

```
⟨term::function declarations 30a⟩+≡
  void setNoRedex() {
        noredex = true;
        for (unint i=0; i≠fieldsize; i++)
              fields[i]→setNoRedex();
  }
```

Comment 2.2.65. We now look at the `tag == V` case. If the current term is a bound variable in t, then the first part of Definition 2.2.57 (3) applies and nothing changes. If the current term is a free variable in t and does not occur in θ, the second part of Definition 2.2.57 (1) applies and again nothing happens. If the current term is a free variable in t that matches an x_i in θ, then the first part of Definition 2.2.57 (1) applies and we substitute the current term with t_i. Before we do that, however, we check whether any free variable in t_i is captured by any λ abstraction that encloses the current term. If yes, part (a) of Definition 2.2.57 (3) applies and we must rename the offending λ variable before replacing the current term with t_i. Otherwise, part (b) of Definition 2.2.57 (3) applies and we can just go ahead and replace the current term with t_i.

53h

```
⟨subst2::case of V 53h⟩≡
  if (isFree() ≡ false) return;
  int size = subs.size();
  for (int i=0; i≠size; i++) {
        if (cname ≠ subs[i].first) continue;
        ⟨subst2::free variable captured 54a⟩
        ⟨subst2::replace by ti 53a⟩
        return;
  }
  return;
```

Uses `isFree` 46e.

54a ⟨subst2::free variable captured 54a⟩≡

```
int k;
while (subs[i].second→captured(bindingAbss,k))
    bindingAbss[k]→renameLambdaVar(bindingAbss[k]→fields[0]→cname,newPVar());
```

Uses `captured` 48b, `newPVar` 148c, and `renameLambdaVar` 50.

Comment 2.2.66. The use of `captured` (hence the use of cached computed free variables) here warrants some caution. If `subs[i].second` does not remain unchanged throughout the term substitution process, errors can creep in. We now argue that `subs[i].second` stays unchanged throughout.

Term substitution is only used in two places in Escher. The first place is in the construction of body instances after successful pattern matching on the head of a statement. (See Comment 3.1.71.) The use of `subst` has no problem here because all the terms in θ are in the matched redex, whereas we only do surgery on the (cloned) body of a statement.

The other place term substitutions take place is in some of the internal simplification routines described in §3.1.1. Such uses only ever involve a single pair $\{x/t\}$. In all routines except beta reduction, t will remain unchanged because of the requirement that x does not occur free in t. In beta reduction (see Comment 3.1.14), it is easy to see that t remains unchanged since substitution is a once off operation. That is, even if x occurs free in t, it will never be substituted. (Otherwise, we will have an infinite recursion.)

Comment 2.2.67. The following is the version of `subst` that handles singleton term substitutions. We make a vector out of the single pair and use `subst2` to do the job.

54b ⟨term::function definitions 32e⟩+≡

```
void term::subst(substitution & sub) {
        if (tag ≡ V ∨ tag ≡ SV) {
                if (cname ≡ sub.first) this→replace(sub.second); return; }
        labelStaticBoundVars();
        vector<term *> bindingAbs;
        vector<substitution> subs; subs.push_back(sub);
        subst2(subs, bindingAbs, NULL);
        unmarkValidfree();
}
```

Uses `labelStaticBoundVars` 46f, `replace` 42b, `subst` 52a, `subst2` 52b, and `substitution` 51.

Comment 2.2.68. A correct implementation of substitution should get the following right. Given the statement

```
(func z) = \x.\y.\x.(&& z (|| x y)).
```

and the query

```
: (func (f x y)),
```

Escher should produce the following

```
\pve0.\pve1.\pve0.(&& (f x y) (|| pve0 pve1)).
```

Notice that *t*wo free variables got captured along the way.

2.2.7 Theorem Prover Helper Functions

Comment 2.2.69. We now look at some functions that check whether a given term satisfy some properties. These functions are needed by the theorem prover.

Comment 2.2.70. A free variable is a variable with a `cname` that is larger or equal to 100000.

55a ⟨term::function definitions 32e⟩+≡

```
bool isUVar(term ∗ t) { return (t→tag ≡ V ∧ t→cname ≥ 100000); }
bool isUVar(int cn) { return (cn ≥ 100000); }

bool term::containsFreeVariable() {
        if (tag ≡ V) return isUVar(cname);
        for (unint i=0; i≠fieldsize; i++)
                if (fields[i]→containsFreeVariable()) return true;
        return false;
}

void term::collectFreeVariables(set<int> & fvars) {
        if (tag ≡ V ∧ isUVar(cname)) fvars.insert(cname);
        for (unint i=0; i≠fieldsize; i++)
                fields[i]→collectFreeVariables(fvars);
        return;
}
```

Defines:
`collectFreeVariables`, used in chunk 111.
`containsFreeVariable`, used in chunk 111.
`isUVar`, used in chunks 108b and 111.
Uses `insert` 30e.

Comment 2.2.71. This next function checks whether the current term has the form $\neg\phi$ for some ϕ.

55b ⟨term::function definitions 32e⟩+≡

```
bool term::isNegation() { return (isApp() ∧ lc()→isF(iNot)); }
```

Defines:
`isNegation`, used in chunks 55c, 56b, and 111.
Uses `iNot` 145, `isApp` 30a, `isF` 30a, and `lc` 30e.

Comment 2.2.72. This function checks whether the current term has the form $\neg\Box_i\neg\phi$ ($\equiv \Diamond\phi$) for some ϕ.

55c ⟨term::function definitions 32e⟩+≡

```
bool term::isDiamond() {
        return (isNegation() ∧ rc()→isModal() ∧
                rc()→fields[0]→isNegation());
}
```

Defines:
`isDiamond`, used in chunk 111.
Uses `isModal` 30a, `isNegation` 55b, and `rc` 30e.

Comment 2.2.73. The next function checks whether the input term $t2$ is the negation (at the syntactic level only) of the calling term $t1$. We have to worry about symmetries here.

56a ⟨term::function definitions 32e⟩+≡

```
bool term::isNegationOf(term ∗ t2) {
        term ∗ t1 = this;
        if ((t1→isD(iTrue) ∧ t2→isD(iFalse)) ∨
            (t1→isD(iFalse) ∧ t2→isD(iTrue)))
                return true;
        if (t2→isApp() ∧ t2→lc()→isF(iNot) ∧ t1→equal(t2→rc()))
                return true;
        if (t1→isApp() ∧ t1→lc()→isF(iNot) ∧ t2→equal(t1→rc()))
                return true;
        return false;
}
```

Defines:
`isNegationOf`, used in chunk 111.
Uses `equal` 34b, `iFalse` 145, `iNot` 145, `iTrue` 145, `isApp` 30a, `isD` 30a, `isF` 30a, `lc` 30e, and `rc` 30e.

Comment 2.2.74. This function strips off double negations from a term.

56b ⟨term::function definitions 32e⟩+≡

```
void term::stripNegations() {
        if (isNegation() ∧ rc()→isNegation()) {
                term ∗ term = rc()→rc();
                rc()→fields[1] = NULL;
                this→replace(term);
        }
}
```

Defines:
`stripNegations`, used in chunk 111.
Uses `isNegation` 55b, `rc` 30e, and `replace` 42b.

Comment 2.2.75. Sometimes, we need to replace a subterm s in t by another term r. The following function performs this operation. Note that free-variable capture can occur as a result; no attempt is made to check for this condition.

56c ⟨term::function definitions 32e⟩+≡

```
bool term::termReplace(term ∗ s, term ∗ r, term ∗ parent, int id) {
        if (equal(s)) {
                if (parent ≡ NULL) this→replace(r→clone());
                else { parent→fields[id]→freememory();
                        parent→fields[id] = r→clone(); }
                return true;
        }
        bool ret = false;
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                ret = (ret ∨ fields[i]→termReplace(s, r, this, i));
        return ret;
}
```

Defines:
`termReplace`, used in chunk 111.
Uses `equal` 34b and `replace` 42b.

Comment 2.2.76. The function `matchReplace` takes a term s having the form $\Box_{i_1}\cdots\Box_{i_j}x$, a term r having the form $\Box_{j_1}\cdots\Box_{j_k}x$, and replaces every subterm t of the calling term such that $t = s\theta$ for some θ with the term $r\theta$. We find θ using the `redex_match` function used for pattern matching. This is perhaps a lazy way of doing things....

57a ⟨term::function definitions 32e⟩+≡

```
extern bool redex_match(term ∗ head, term ∗ body, vector<substitution> & theta);

bool term::matchReplace(term ∗ s, term ∗ r, term ∗ parent, int id) {
        vector<substitution> theta;
        if (redex_match(s, this, theta)) {
                term ∗ r2 = r→clone();
                r2→subst(theta);
                if (parent ≡ NULL) this→replace(r2);
                else {  parent→fields[id]→freememory();
                        parent→fields[id] = r2; }
                return true;
        }
        bool ret = false;
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                ret = (ret ∨ fields[i]→matchReplace(s, r, this, i));
        return ret;
}
```

Defines:
`matchReplace`, used in chunk 111.
Uses `redex_match` 99a 99b 100a, `replace` 42b, `subst` 52a, and `substitution` 51.

Comment 2.2.77. We normalise a term to contain only some minimal set of system-defined constants. This is done in two steps. In the first step, we perform the basic transformations. This process generates many double negations. We remove these in the second step.

57b ⟨term::function definitions 32e⟩+≡

```
term ∗ term::normalise()
        { return (this→normalise1())→normalise2(); }
```

Defines:
`normalise`, used in chunks 58, 107a, and 111.
Uses `normalise1` 58 and `normalise2` 60a.

Comment 2.2.78. This next function transforms the calling term into normal form. A term is in normal form if it contains only the following system-defined function symbols: *False*, *not*, $\|\|$, $\Diamond(\equiv not\Box not)$ and $\exists$.

```
58  ⟨term::function definitions 32e⟩+≡
      term ∗ term::normalise1() {
          for (unint i=0; i≠fieldsize; i++)
              fields[i] = fields[i]→normalise();
          term ∗ ret;
          if (isD(iTrue)) {
              // ret = new_term(APP); ret->lc = new_term(F, iNot);
              // ret->rc = new_term(D, iFalse);

              ret = new_term(APP); ret→insert(new_term(F, iNot));
              ret→insert(new_term(D, iFalse));
              this→replace(ret); return this;
          }
          if (isFunc2Args(iImplies)) {
              lc()→lc()→cname = iOr;
              ret = new_term(APP);
              // ret->lc = new_term(F, iNot);
              // ret->rc = lc->rc;
              ret→insert(new_term(F, iNot));
              ret→insert(lc()→rc());
              lc()→fields[1] = ret;
              return this;
          }
          if (isFunc2Args(iAnd)) { ⟨normalise1::and 59a⟩ }
          if (isFunc2Args(iIff)) { ⟨normalise1::iff 59b⟩ }
          if (isApp() ∧ lc()→isF(iPi)) {
              lc()→cname = iSigma;
              ret = new_term(APP);
              ret→insert(new_term(F, iNot));
              ret→insert(rc()→fields[1]);
              rc()→fields[1] = ret;
              ret = new_term(APP); ret→insert(new_term(F, iNot));
              ret→insert(this);
              return ret;
          }
          return this;
      }
```

Defines:
`normalise1`, used in chunks 57b, 59b, and 111.

Uses `iAnd` 145, `iFalse` 145, `iIff` 145, `iImplies` 145, `iNot` 145, `iOr` 145, `iPi` 145, `iSigma` 145, `iTrue` 145, `insert` 30e, `isApp` 30a, `isD` 30a, `isF` 30a, `isFunc2Args` 32f, `lc` 30e, `new_term` 40a, `normalise` 57b, `rc` 30e, and `replace` 42b.

Comment 2.2.79. We turn a formula of the form `(&& p q)` into another formula `(|| (not p) (not q))`.

59a ⟨normalise1::and 59a⟩≡

```
ret = new_term(APP);
// ret->lc = new_term(F, iNot);
ret→insert(new_term(F, iNot));
term ∗ arg2 = newT2Args(F, iOr);
term ∗ arg21 = new_term(APP);
// arg21->lc = new_term(F, iNot); arg21->rc = lc->rc->clone();
arg21→insert(new_term(F, iNot)); arg21→insert(lc()→rc()→clone());
term ∗ arg22 = new_term(APP);
// arg22->lc = new_term(F, iNot); arg22->rc = rc->clone();
arg22→insert(new_term(F, iNot)); arg22→insert(rc()→clone());
arg2→initT2Args(arg21, arg22);
// ret->rc = arg2;
ret→insert(arg2);
this→replace(ret); return this;
```

Uses `iNot` 145, `iOr` 145, `initT2Args` 34a, `insert` 30e, `lc` 30e, `newT2Args` 33c, `new_term` 40a, `rc` 30e, and `replace` 42b.

Comment 2.2.80. We change a formula of the form `(iff p q)` into a formula of the form `(&& (|| (not p) q) (|| (not q) p))` and then normalise again to turn the conjunction into a disjunction.

59b ⟨normalise1::iff 59b⟩≡

```
ret = newT2Args(F, iAnd);
term ∗ arg1 = newT2Args(F, iOr);
term ∗ arg1a = new_term(APP);
// arg1a->lc = new_term(F, iNot); arg1a->rc = lc->rc->clone();
arg1a→insert(new_term(F, iNot)); arg1a→insert(lc()→rc()→clone());
arg1→initT2Args(arg1a, rc()→clone());
term ∗ arg2 = newT2Args(F, iOr);
term ∗ arg2a = new_term(APP);
// arg2a->lc = new_term(F, iNot); arg2a->rc = rc->clone();
arg2a→insert(new_term(F, iNot)); arg2a→insert(rc()→clone());
arg2→initT2Args(arg2a, lc()→rc()→clone());
ret→initT2Args(arg1, arg2);
ret = ret→normalise1();
this→replace(ret); return this;
```

Uses `iAnd` 145, `iNot` 145, `iOr` 145, `initT2Args` 34a, `insert` 30e, `lc` 30e, `newT2Args` 33c, `new_term` 40a, `normalise1` 58, `rc` 30e, and `replace` 42b.

60a ⟨term::function definitions 32e⟩+≡

```
term ∗ term::normalise2() {
        if (isApp() ∧ rc()→isApp() ∧ lc()→isF(iNot) ∧
            rc()→lc()→isF(iNot)) {
                term ∗ ret = rc()→rc();
                rc()→fields[1] = NULL;
                freememory();
                ret = ret→normalise2();
                return ret;
        }
        for (unint i=0; i≠fieldsize; i++)
                fields[i] = fields[i]→normalise2();
        return this;
}
```

Defines:
`normalise2`, used in chunks 57b and 111.
Uses `iNot` 145, `isApp` 30a, `isF` 30a, `lc` 30e, and `rc` 30e.

Comment 2.2.81. The next function allows us to collect together all the function symbols in a term.

60b ⟨term::function definitions 32e⟩+≡

```
void term::collectFunctionNames(set<int> & x) {
        if (tag ≡ F) { x.insert(cname); return; }
        for (unint i=0; i≠fieldsize; i++)
                fields[i]→collectFunctionNames(x);
}
```

Uses `insert` 30e.

Chapter 3

Equational Reasoning

3.1 Term Rewriting

3.1.1 Internal Rewrite Routines

Comment 3.1.1. To capture precisely and completely statement schemas in the booleans module, some of which have complicated side conditions on syntactical variables, we implement them as algorithms. These algorithms form the internal rewrite module of Escher, and they are called before any other program statements.

Comment 3.1.2. This next function implements the following equality statements:

$$= : \ a \rightarrow a \rightarrow \Omega$$

$$(\mathbf{C}\, x_1 \ldots x_n = \mathbf{C}\, y_1 \ldots y_n) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$

% where **C** is a data constructor of arity n.

$$(\mathbf{C}\, x_1 \ldots x_n = \mathbf{D}\, y_1 \ldots y_m) = \bot$$

% where **C** and **D** are data constructors of arity n and m respectively, and $\mathbf{C} \neq \mathbf{D}$.

$$(() = ()) = \top$$

$$((x_1, \ldots, x_n) = (y_1, \ldots, y_n)) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$

% where $n = 2, 3, \ldots$.

61

```
⟨term::function definitions 32e⟩+≡
  bool term::simplifyEquality(term * parent, unint id) {
        bool changed = false;
        term * ret = this;
        term * t1 = lc()→rc(), * t2 = rc();
        ⟨simplifyEquality::local variables 63d⟩

        ⟨simplifyEquality::identical variables and function symbols 62b⟩
        ⟨simplifyEquality::irrelevant cases 62c⟩
        ⟨simplifyEquality::case of strings 62d⟩
        ⟨simplifyEquality::case of products 63a⟩
        ⟨simplifyEquality::case of applications 64a⟩

  simplifyEquality_cleanup:
        if (changed) { ⟨simplify update pointers 62a⟩ }
        return changed;
  }
```

Defines:

`simplifyEquality`, used in chunks 63c, 90, and 111.
Uses `lc` 30e and `rc` 30e.

Comment 3.1.3. The pointer `ret` points to the current term under consideration. If `changed` is true by the end of the operation, then an equality embedded in the current term would have been simplified. Otherwise, it stays the same as before the function is called. Assuming the term has been changed, we have two cases to consider. If the current term is the root term (`parent == NULL`), then we overwrite the current term with `ret`. Otherwise, we simply redirect the pointer `parent->fields[id]` to `ret`. Note that this code chunk is used in the other simplification routines as well.

62a ⟨simplify update pointers 62a⟩≡

```
assert(ret);
if (parent ≡ NULL) {
        this→replace(ret); ret→freememory();
} else { parent→fields[id]→freememory(); parent→fields[id] = ret; }
```

Uses `replace` 42b.

62b ⟨simplifyEquality::identical variables and function symbols 62b⟩≡

```
if ((t1→isVar() ∧ t2→isVar()) ∨ (t1→isF() ∧ t2→isF()))
        if (t1→cname ≡ t2→cname) {
                changed = true; ret = new_term(D, iTrue);
                goto simplifyEquality_cleanup;
        }
```

Uses `iTrue` 145, `isF` 30a, `isVar` 30a, and `new_term` 40a.

Comment 3.1.4. This simplification does not apply when one of the terms is a variable. We also do not handle equality of abstractions. That is done using statements in the booleans module.

62c ⟨simplifyEquality::irrelevant cases 62c⟩≡

```
if (t1→isVar() ∨ t2→isVar()) return false;
if (t1→isAbs()) return false;
```

Uses `isAbs` 30a and `isVar` 30a.

Comment 3.1.5. We have a special case for strings. Strings are represented internally as lists of characters. Using the default rule to check the equality of two lists involves making many smaller steps. The procedure here reduces comparison of strings to a single-step operation. Surprisingly, this is actually not a great deal faster than the default multi-step procedure.

62d ⟨simplifyEquality::case of strings 62d⟩≡

```
if (t1→isAString() ∧ t2→isAString()) {
        changed = true;
        term ∗ p1 = t1, ∗ p2 = t2;
        while (true) {
                if (p1→isD(iEmptyList) ∧ p2→isD(iEmptyList)) break;
                if (p1→tag ≠ p2→tag ∨
                    p1→lc()→rc()→cname≠p2→lc()→rc()→cname)
                      { ret=new_term(D,iFalse); goto simplifyEquality_cleanup;}
                p1 = p1→rc();
                p2 = p2→rc();
        }
        ret = new_term(D, iTrue);
        goto simplifyEquality_cleanup;
}
```

Uses `iEmptyList` 145, `iFalse` 145, `iTrue` 145, `isAString` 33a, `isD` 30a, `lc` 30e, `new_term` 40a, and `rc` 30e.

Comment 3.1.6. We need to check that both `t1` and `t2` are products before proceeding because one of them can be a (nullary) function symbol that stands for another product. However, once we have done that, we only have to check the dimension of `t1` because the type checker would have made sure that `t2` has the same dimension. Given $(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, we create a term of the form $((\cdots((x_1 \wedge y_1) \wedge (x_2 \wedge y_2))\cdots) \wedge (x_n \wedge y_n))$.

63a ⟨simplifyEquality::case of products 63a⟩≡
```
if (t1→isProd() ∧ t2→isProd()) {
        changed = true; unint t1_args = t1→fieldsize;

        ⟨simplifyEquality::case of products::empty tuples 63b⟩
        ⟨simplifyEquality::case of products::error handling 63c⟩

        term ∗ eq1 = newT2Args(F, iEqual);
        eq1→initT2Args(t1→fields[0]→reuse(), t2→fields[0]→reuse());
        term ∗ eq2 = newT2Args(F, iEqual);
        eq2→initT2Args(t1→fields[1]→reuse(), t2→fields[1]→reuse());

        ret = newT2Args(F, iAnd); ret→initT2Args(eq1, eq2);
        for (unint i=0; i≠t1_args-2; i++) {
                term ∗ eqi = newT2Args(F, iEqual);
                eqi→initT2Args(t1→fields[i+2]→reuse(),
                                t2→fields[i+2]→reuse());
                term ∗ temp = newT2Args(F, iAnd);
                temp→initT2Args(ret, eqi);
                ret = temp;
        }
        goto simplifyEquality_cleanup;
}
```
Uses `iAnd` 145, `iEqual` 145, `initT2Args` 34a, `isProd` 30a, `newT2Args` 33c, and `reuse` 43d.

Comment 3.1.7. The boolean module as it stands in [Llo03] does not handle the expression $() = ()$. We will cater for that case here, which should of course evaluate to $\top$.

63b ⟨simplifyEquality::case of products::empty tuples 63b⟩≡
```
if (t1_args ≡ 0) { ret = new_term(D, iTrue); goto simplifyEquality_cleanup; }
```
Uses `iTrue` 145 and `new_term` 40a.

Comment 3.1.8. Besides the empty tuple, we handle all finite-length tuples of dimension at least two. It does not make a great deal of sense to have a tuple of dimension one.

63c ⟨simplifyEquality::case of products::error handling 63c⟩≡
```
if (t1_args ≠ t2→fieldsize ∨ t1_args < 2) {
        setSelector(STDERR); ioprint("Error in simplifyEquality:products\n");
        t1→print(); ioprintln(); t2→print(); ioprintln();
}
assert(t1_args ≡ t2→fieldsize ∧ t1_args ≥ 2);
```
Uses `ioprint` 164 165, `ioprintln` 164 165, `setSelector` 164 165, and `simplifyEquality` 61.

63d ⟨simplifyEquality::local variables 63d⟩≡
```
int t1_arity = 0, t2_arity = 0;
```

64a ⟨simplifyEquality::case of applications 64a⟩≡

```
⟨simplifyEquality::check whether we have data constructors 64b⟩
changed = true;

if (t1_arity ≡ 0 ∧ t2_arity ≡ 0) {
        if (t1→spineTip()→isfloat ∧ t2→spineTip()→isfloat) {
                if (t1→spineTip()→numf ≡ t2→spineTip()→numf)
                        ret = new_term(D, iTrue);
                else ret = new_term(D, iFalse);
                goto simplifyEquality_cleanup;
        } else if (t1→spineTip()→isint ∧ t2→spineTip()→isint) {
                if (t1→spineTip()→numi ≡ t2→spineTip()→numi)
                        ret = new_term(D, iTrue);
                else ret = new_term(D, iFalse);
                goto simplifyEquality_cleanup;
        }
        if (t1→spineTip()→cname ≡ t2→spineTip()→cname)
                ret= new_term(D,iTrue);
        else ret = new_term(D, iFalse);
        goto simplifyEquality_cleanup;
}
if (t1_arity ≠ t2_arity ∨ t1→spineTip()→cname ≠ t2→spineTip()→cname)
        { ret = new_term(D, iFalse); goto simplifyEquality_cleanup; }

ret = newT2Args(F, iEqual);
ret→initT2Args(t1→fields[1]→reuse(), t2→fields[1]→reuse());
t1_arity−−;
while (t1_arity ≠ 0) {
        t1 = t1→fields[0]; t2 = t2→fields[0];
        term ∗ temp = newT2Args(F, iEqual);
        temp→initT2Args(t1→fields[1]→reuse(), t2→fields[1]→reuse());
        term ∗ temp2 = newT2Args(F, iAnd); temp2→initT2Args(temp, ret);
        ret = temp2;
        t1_arity−−;
}
```

Uses `iAnd` 145, `iEqual` 145, `iFalse` 145, `iTrue` 145, `initT2Args` 34a, `newT2Args` 33c, `new_term` 40a, `reuse` 43d, and `spineTip` 32e.

Comment 3.1.9. We need to check whether the leftmost symbol of both `t1` and `t2` is a data constructor. If we go pass this point, `t1` and `t2` have the right form for comparison.

64b ⟨simplifyEquality::check whether we have data constructors 64b⟩≡

```
if (¬t1→spineTip(t1_arity)→isD()) return false;
if (¬t2→spineTip(t2_arity)→isD()) return false;
```

Uses `isD` 30a and `spineTip` 32e.

Comment 3.1.10. This function implements the different arithmetic operations. We currently support the following functions on numbers. More can be added if necessary.

```
65  ⟨term::function definitions 32e⟩+≡
    bool term::simplifyArithmetic(term ∗ parent, unint id) {
        if (¬(rc()→isD() ∧ lc()→rc()→isD())) return false;

        int op = fields[0]→lc()→cname;
        if (¬(op ≥ iAdd ∧ op ≤ iAtan2)) return false;

        term ∗ t1 = lc()→rc(), ∗ t2 = rc();
        if (t1→isD(iInfinity) ∨ t2→isD(iInfinity)) return false;

        term ∗ ret = NULL;
        ⟨simplifyArithmetic::add, subtract, multiply and divide 66⟩
        else if (op ≡ iMax) {
            if (t1→isfloat ∧ t2→isfloat) {
                if (t1→numf ≥ t2→numf) ret =new_term_float(t1→numf);
                else ret = new_term_float(t2→numf) ;
            } else if (t1→isint ∧ t2→isint) {
                if (t1→numi ≥ t2→numi) ret = new_term_int(t1→numi);
                else ret = new_term_int(t2→numi) ;
            } else return false;
        } else if (op ≡ iMin) {
            if (t1→isfloat ∧ t2→isfloat) {
                if (t1→numf ≤ t2→numf) ret =new_term_float(t1→numf);
                else ret = new_term_float(t2→numf) ;
            } else if (t1→isint ∧ t2→isint) {
                if (t1→numi ≤ t2→numi) ret = new_term_int(t1→numi);
                else ret = new_term_int(t2→numi) ;
            } else return false;
        } else if (op ≡ iMod) {
            assert(t1→isint ∧ t2-isint);
            ret = new_term_int(int(t1→numi % t2→numi));
        } else if (op ≡ iAtan2) {
            assert(t1→isfloat ∧ t2→isfloat);
            ret = new_term_float(atan2(t1→numf, t2→numf));
        }
        ⟨simplify update pointers 62a⟩
        return true;
    }
```

Defines:
`simplifyArithmetic`, used in chunks 90 and 111.

Uses `iAdd` 145, `iAtan2` 145, `iInfinity` 145, `iMax` 145, `iMin` 145, `iMod` 145, `isD` 30a, `lc` 30e, `new_term_float` 40a, `new_term_int` 40a, and `rc` 30e.

Comment 3.1.11. We overload the basic addition, subtraction, multiplication and division operations to act on numbers, be they integers or floating-point numbers. The definitions are fairly standard, when one of the arguments is a floating-point number, the result is a floating-point number. When both arguments are integers, the result is an integer, except when we are dividing two integers, in which case the result can be a floating-point number.

```
66  ⟨simplifyArithmetic::add, subtract, multiply and divide 66⟩≡
      if (op ≡ iAdd) {
              if (t1→isfloat ∧ t2→isfloat)
                      ret = new_term_float(t1→numf + t2→numf);
              else if (t1→isfloat ∧ t2→isint)
                      ret = new_term_float(t1→numf + t2→numi);
              else if (t1→isint ∧ t2→isfloat)
                      ret = new_term_float(t1→numi + t2→numf);
              else if (t1→isint ∧ t2→isint)
                      ret = new_term_int(t1→numi + t2→numi);
              else return false;
      } else if (op ≡ iSub) {
              if (t1→isfloat ∧ t2→isfloat)
                      ret = new_term_float(t1→numf - t2→numf);
              else if (t1→isfloat ∧ t2→isint)
                      ret = new_term_float(t1→numf - t2→numi);
              else if (t1→isint ∧ t2→isfloat)
                      ret = new_term_float(t1→numi - t2→numf);
              else if (t1→isint ∧ t2→isint)
                      ret = new_term_int(t1→numi - t2→numi);
              else return false;
      } else if (op ≡ iMul) {
              if (t1→isfloat ∧ t2→isfloat)
                      ret = new_term_float(t1→numf ∗ t2→numf);
              else if (t1→isfloat ∧ t2→isint)
                      ret = new_term_float(t1→numf ∗ t2→numi);
              else if (t1→isint ∧ t2→isfloat)
                      ret = new_term_float(t1→numi ∗ t2→numf);
              else if (t1→isint ∧ t2→isint)
                      ret = new_term_int(t1→numi ∗ t2→numi);
              else return false;
      } else if (op ≡ iDiv) {
              if (t1→isfloat ∧ t2→isfloat)
                      ret = new_term_float(t1→numf ÷ t2→numf);
              else if (t1→isfloat ∧ t2→isint)
                      ret = new_term_float(t1→numf ÷ t2→numi);
              else if (t1→isint ∧ t2→isfloat)
                      ret = new_term_float(t1→numi ÷ t2→numf);
              else if (t1→isint ∧ t2→isint) {
                      double res = (double)t1→numi ÷ (double)t2→numi;
                      if (res ≡ floor(res)) ret = new_term_int(t1→numi ÷ t2→numi);
                      else ret = new_term_float(res);
              } else return false;
      }
```

Uses `iAdd` 145, `iDiv` 145, `iMul` 145, `iSub` 145, `new_term_float` 40a, and `new_term_int` 40a.

Comment 3.1.12. This function implements the different inequalities. It has the same overall structure as `simplifyArithmetic`.

```
67  ⟨term::function definitions 32e⟩+≡
      bool term::simplifyInequalities(term ∗ parent, unint id) {
            if (¬(rc()→isD() ∧ lc()→rc()→isD())) return false;
            int rel = lc()→lc()→cname;

            if (¬(rel ≥ iLT ∧ rel ≤iGTE)) return false;

            term ∗ t1 = lc()→rc() ;
            term ∗ t2 = rc() ;

            if (t1→isD(iInfinity) ∨ t2→isD(iInfinity)) return false;

            term ∗ ret = NULL;
            if (rel ≡ iLT) {
                  if (t1→isint ∧ t2→isint) {
                        if (t1→numi < t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isfloat) {
                        if (t1→numf < t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isint ∧ t2→isfloat) {
                        if (t1→numi < t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isint) {
                        if (t1→numf < t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else return false;
            } else if (rel ≡ iLTE) {
                  if (t1→isint ∧ t2→isint) {
                        if (t1→numi ≤ t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isfloat) {
                        if (t1→numf ≤ t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isint ∧ t2→isfloat) {
                        if (t1→numi ≤ t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isint) {
                        if (t1→numf ≤ t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else return false;
            } else if (rel ≡ iGT) {
                  if (t1→isint ∧ t2→isint) {
                        if (t1→numi > t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isfloat) {
                        if (t1→numf > t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isint ∧ t2→isfloat) {
                        if (t1→numi > t2→numf) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
                  } else if (t1→isfloat ∧ t2→isint) {
                        if (t1→numf > t2→numi) ret = new_term(D,iTrue);
                        else ret = new_term(D,iFalse);
```

```
            } else return false;
        } else if (rel ≡ iGTE) {
            if (t1→isint ∧ t2→isint) {
                if (t1→numi ≥ t2→numi) ret = new_term(D,iTrue);
                else ret = new_term(D,iFalse);
            } else if (t1→isfloat ∧ t2→isfloat) {
                if (t1→numf ≥ t2→numf) ret = new_term(D,iTrue);
                else ret = new_term(D,iFalse);
            } else if (t1→isint ∧ t2→isfloat) {
                if (t1→numi ≥ t2→numf) ret = new_term(D,iTrue);
                else ret = new_term(D,iFalse);
            } else if (t1→isfloat ∧ t2→isint) {
                if (t1→numf ≥ t2→numi) ret = new_term(D,iTrue);
                else ret = new_term(D,iFalse);
            } else return false;
        }
        ⟨simplify update pointers 62a⟩
        return true;
    }
```

Defines:
  `simplifyInequalities`, used in chunks 90 and 111.

Uses `iFalse` 145, `iGT` 145, `iGTE` 145, `iInfinity` 145, `iLT` 145, `iLTE` 145, `iTrue` 145, `isD` 30a, `lc` 30e, `new_term` 40a, and `rc` 30e.

Comment 3.1.13. We use the C math library to support common math operations like sin, cos, etc.

68 ⟨term::function definitions 32e⟩+≡

```
    bool term::simplifyMath(term ∗ parent, unint id) {
        if (¬(lc()→isF() ∧ rc()→isD())) return false;
        int op = lc()→cname;
        if (¬(op ≥ iSin ∧ op ≤ iExp)) return false;

        term ∗ ret = NULL;
        if (op ≡ iSin) {
            assert(rc()→isfloat);
            ret = new_term_float(sin(rc()→numf));
        } else if (op ≡ iCos) {
            assert(rc()→isfloat);
            ret = new_term_float(cos(rc()→numf));
        } else if (op ≡ iSqrt) {
            assert(rc()→isfloat);
            ret = new_term_float(sqrt(rc()→numf));
        } else if (op ≡ iExp) {
            assert(rc()→isfloat);
            ret = new_term_float(exp(rc()→numf));
        }
        ⟨simplify update pointers 62a⟩
        return true;
    }
```

Defines:
  `simplifyMath`, used in chunks 90 and 111.

Uses `iCos` 145, `iExp` 145, `iSin` 145, `iSqrt` 145, `isD` 30a, `isF` 30a, `lc` 30e, `new_term_float` 40a, and `rc` 30e.

Comment 3.1.14. The β-reduction rule $(\lambda x.\mathbf{u}\ t) = \mathbf{u}\{x/t\}$ in the booleans module is not really a valid program statement. (The leftmost symbol on the LHS of the equation is not a function symbol.) It should therefore be thought of as a part of the internal simplification routine of Escher. This rule is also the first among a few we will encounter where sharing of nodes in the current term is not safe because of the appearance of term substitutions on the RHS of the equation. (See Comments 3.1.16, 3.1.32 and 3.1.43 for the other such rules. The existence and (heavy) use of such rules in Escher is one important reason I gave up on sharing of nodes. See Comment 2.2.33 for a more detailed discussion on the advantages and disadvantages of sharing.) In a typical program statement $h = b$ without term substitutions in the body, rewriting a subterm that is α-equivalent (see §3.1.3 for the exact details) to h in the current term with b involves only the creation and destruction of terms and redirection of pointers to terms. No actual modification to an atomic term embedded inside the current term actually takes place, which means sharing is always safe. This scenario is no longer true when term substitutions appear in the body of statements.

69a ⟨term::function definitions 32e⟩+≡

```
bool term::betaReduction(term ∗ parent, unint id) {
        if (lc()→isAbs() ≡ false) return false;

        substitution bind(lc()→fields[0]→cname, rc());
        lc()→fields[1]→subst(bind);
        term ∗ ret = lc()→fields[1]→reuse();
        ⟨simplify update pointers 62a⟩
        return true;
}
```

Defines:
`betaReduction`, used in chunks 90 and 111.
Uses `isAbs` 30a, `lc` 30e, `rc` 30e, `reuse` 43d, `subst` 52a, and `substitution` 51.

Comment 3.1.15. This implements the rule

if $(x = s)$ *then* w *else* z = *if* $(x = s)$ *then* $w\{x/s\}$ *else* z
- - where x is a variable with a free occurrence in w.

This rule is relatively new and is first needed in Bayesian tracking applications.

69b ⟨term::function definitions 32e⟩+≡

```
bool term::simplifyIte(term ∗ parent, unint id) {
        if (¬lc()→isF(iIte)) return false;
        term ∗ cond = rc()→fields[0];
        if (¬cond→isFunc2Args(iEqual)) return false;
        if (¬cond→lc()→rc()→isVar()) return false;
        int vname = cond→lc()→rc()→cname;
        if (¬rc()→fields[1]→occursFree(vname)) return false;
        substitution bind(vname, cond→rc());
        rc()→fields[1]→subst(bind);
        return true;
}
```

Defines:
`simplifyIte`, used in chunk 111.
Uses `iEqual` 145, `iIte` 145, `isF` 30a, `isFunc2Args` 32f, `isVar` 30a, `lc` 30e, `occursFree` 47b, `rc` 30e, `subst` 52a, and `substitution` 51.

Comment 3.1.16. This function implements the following conjunction rule:

$$\mathbf{u} \wedge (\mathbf{x} = \mathbf{t}) \wedge \mathbf{v} = \mathbf{u}\{\mathbf{x}/\mathbf{t}\} \wedge (\mathbf{x} = \mathbf{t}) \wedge \mathbf{v}\{\mathbf{x}/\mathbf{t}\}. \tag{3.1}$$

Here, $\mathbf{t}$ is not a variable and $\mathbf{x}$ is a variable free in $\mathbf{u}$ or $\mathbf{v}$ or both, but not free in $\mathbf{t}$. The LHS of the equation is supposed to capture every term that has a subterm $(\mathbf{x} = \mathbf{t})$ embedded conjunctively (see Definition 3.1.18) inside it. All the variables in the rule are syntactical variables because a subterm that pattern matches with the LHS of the equation can occur inside a term that binds the variable x, in which case the standard term substitution routine will not give us what we want.

The condition that $\mathbf{t}$ is not a variable is important. If $\mathbf{t}$ is a free variable and we interpret $(\mathbf{x} = \mathbf{t})$ to stand for $(\mathbf{x} = \mathbf{t})$ or $(\mathbf{t} = \mathbf{x})$, I think the correct interpretation, then loops can result from repeated application of the rule. In [Llo99], the statement

$$(t = x) = (x = t) \text{ where } x \text{ is a variable, and } t \text{ is not a variable}$$

is used to capture the symmetry between $\mathbf{x}$ and $\mathbf{t}$ in the rule. In the current implementation, we do away with the swapping rule and implement the symmetry directly to gain better efficiency. The condition that $\mathbf{t}$ is not a variable does not appear in [Llo03]; this suggests that the rule as it appears in the book is either loopy or incomplete, depending on how one interprets the rule.

There is another *s*mall problem with the rule. Note that I have been calling it a rule, not a statement. Why? In any instantiation of the rule, the variable $\mathbf{x}$ must occur free in at least two places, which means the instantiation cannot be a statement because of the no repeated free variables condition. *This error appears in every description of Escher before 22 Sep 2005, the day it was discovered.* The use of this rule, among other things, affects the run-time type checking is unnecessary result (See Proposition 5.1.3 in [Llo03]). This is not as bad as it sounds; we only have to type-check every time we use the conjunction rule, *n*ot after every computation step.

Problem 3.1.17. What is the cost, in terms of expressiveness, of omitting this rule?

Definition 3.1.18. A term t is embedded conjunctively in t and, if t is embedded conjunctively in r (or s), then t is embedded conjunctively in $r \wedge s$.

Comment 3.1.19. We could implement the rule completely using the following set of statements.

$$((\mathbf{x} = \mathbf{t}) \wedge \mathbf{u}) = ((\mathbf{x} = \mathbf{t}) \wedge \mathbf{u}\{\mathbf{x}/\mathbf{t}\})$$
$$(\mathbf{u} \wedge (\mathbf{x} = \mathbf{t})) = ((\mathbf{x} = \mathbf{t}) \wedge \mathbf{u})$$
$$\text{where } \mathbf{u} \text{ does not have the form } (y = s) \text{ for some terms } y \text{ and } s.$$
$$(((\mathbf{x} = \mathbf{t}) \wedge \mathbf{u}) \wedge \mathbf{v}) = ((\mathbf{x} = \mathbf{t}) \wedge (\mathbf{u} \wedge \mathbf{v}))$$
$$(\mathbf{u} \wedge ((\mathbf{x} = \mathbf{t}) \wedge \mathbf{v})) = ((\mathbf{x} = \mathbf{t}) \wedge (\mathbf{u} \wedge \mathbf{v}))$$
$$\text{where } \mathbf{u} \text{ does not have the form } (y = s) \text{ for some terms } y \text{ and } s.$$

The last three statements can bring out conjunctively embedded equations to the front of the term, which can then be simplified using the first statement. A loop can occur if the side conditions in the second and fourth statements are not imposed.

Comment 3.1.20. Notice that we do not need the `parent` pointer for this particular rewriting.

Comment 3.1.21. In the following, we first check that the current term has the right form, then we find (using `findEq` (see Comment 3.1.22)) a variable-instantiating equation inside the current term. By a variable-instantiating equation I mean a (sub)term having the form $(x = t)$ embedded conjunctively inside the current term which satisfies all the side conditions of Equation 3.1. (If there are more than one variable-instantiating equation, the leftmost is selected. Subsequent calls to `findEq` on the current term (rewritten using Equation 3.1) will find the remaining variable-instantiating equations in the left-to-right order.) If no such equation exists, `findEq` returns a null pointer. We rename the x in $(x = t)$ temporarily so that it does not get substituted with t by `subst`. Since we will not call `freememory` on the current term, we need to `reuse` the term `p->fields[1]` when creating `bind` to make sure the term substitution works as expected.

71a ⟨term::function definitions 32e⟩+≡

```
bool term::simplifyConjunction() {
        term ∗ p = findEq(this); if (p ≡ NULL) return false;

        term ∗ varp = p→lc()→rc();
        varp→freeze = true;

        substitution bind(varp→cname, p→fields[1]→reuse());
        subst(bind); p→fields[1]→refcount−−;
        varp→freeze = false;

        return true;
}
```

Defines:
`simplifyConjunction`, used in chunks 90 and 111.
Uses `findEq` 71b, `freeze` 31f, `lc` 30e, `rc` 30e, `reuse` 43d, `subst` 52a, and `substitution` 51.

Comment 3.1.22. The function `findEq` seeks a variable-instantiating equation inside the `root` term with the help of `isEq`. The function `findEq` assumes that the calling term is a conjunction of the form $t_1 \wedge t_2$. (See Definition 3.1.18.) If t_1 is a variable-instantiating equation, we return. Otherwise, we recurse on t_1 if it has the right (conjunctive) form. Then we do the same on t_2. This gives us the left-to-right selection order.

71b ⟨term::function definitions 32e⟩+≡

```
term ∗ term::findEq(term ∗ root) {
        term ∗ p = NULL;
        term ∗ t1 = lc()→rc();
        if (t1→isEq(root)) return t1;
        if (t1→isFunc2Args(iAnd)) { p = t1→findEq(root); if (p) return p; }

        term ∗ t2 = rc();
        if (t2→isEq(root)) return t2;
        if (t2→isFunc2Args(iAnd)) { p = t2→findEq(root); if (p) return p; }
        return NULL;
}
```

Defines:
`findEq`, used in chunks 71a and 111.
Uses `iAnd` 145, `isEq` 72a 77b, `isFunc2Args` 32f, `lc` 30e, and `rc` 30e.

Problem 3.1.23. Getting `findEq` to run fast is an interesting search problem. The first question is whether left-to-right is the right search order? We can implement top-to-bottom search by `isEq`ing t_1 and t_2 first, followed by recursion into each of them. Would that be better? Another question is can we improve search time by representing conjunctive terms differently, for example in a flat representation? Or if we do stick with the tree representation, can we augment nodes (in the spirit of binary search algorithms) to make the search go faster?

Comment 3.1.24. This function checks whether the current term is a variable-instantiating term, that is, whether it has the form $(t_1 = t_2)$, where t_1 is a variable, t_2 is a (non-variable) term such that t_1 does not occur free in it, and t_1 occurs free *e*lsewhere in the term `root`. Note the symmetry between t_1 and t_2. We *m*ust check both because any one of them can turn out to be the variable that satisfies all the conditions.

```
72a ⟨term::function definitions 32e⟩+≡
      bool term::isEq(term ∗ root) {
            if (isFunc2Args(iEqual) ≡ false) return false;
            term ∗ t1 = lc()→rc(), ∗ t2 = rc();
            if (t1→isVar() ∧ t2→isVar() ≡ false) {
                  if (t2→occursFree(t1→cname) ≡ false) {
                        t1→freeze = true;
                        if (root→occursFreeNaive(t1→cname)) {
                              t1→freeze = false; return true; }
                        t1→freeze = false;
                  }}
            if (t2→isVar() ∧ t1→isVar() ≡ false) {
                  if (t1→occursFree(t2→cname) ≡ false) {
                        t2→freeze = true;
                        bool ret = root→occursFreeNaive(t2→cname);
                        t2→freeze = false;
                        if (ret) { ⟨isEq::switch t1 and t2 72b⟩ }
                        return ret;
                  }}
            return false;
      }
```

Defines:
`isEq`, used in chunks 71b, 76–78, and 111.

Uses `freeze` 31f, `iEqual` 145, `isFunc2Args` 32f, `isVar` 30a, `lc` 30e, `occursFree` 47b, `occursFreeNaive` 47c, and `rc` 30e.

Comment 3.1.25. We use `occursFreeNaive` for `root` here because the temporary variable renaming we do affects the correctness of the caching of computed free variables. (Note: We no longer do variable renaming, do we still really need to use `occursFreeNaive`?

Comment 3.1.26. We need to swap t_1 and t_2 because procedures that call `findEq` expect the variable that satisfies all the conditions to be on the LHS of the equation.

```
72b ⟨isEq::switch t1 and t2 72b⟩≡
      term ∗ temp = t1; lc()→fields[1] = t2; fields[1] = temp;
```

Uses `lc` 30e.

Comment 3.1.27. Example execution of `simplifyConjunction`.

```
Query: ((&& y) ((&& ((== x) T1)) ((&& ((== T2) y)) x)))
Time = 1 Answer: ((&& y) ((&& ((== x) T1)) ((&& ((== T2) y)) T1)))
Time = 2 Answer: ((&& T2) ((&& ((== x) t1)) ((&& ((== y) T2)) T1)))
```

There are two variable-instantiating equations in the query. It is easy to get this wrong if one is not careful.

Comment 3.1.28. We next look at the implementation of the rules

$$\mathbf{u} \wedge (\exists x_1. \cdots \exists x_n.\mathbf{v}) = \exists x_1. \cdots \exists x_n.(\mathbf{u} \wedge \mathbf{v}) \tag{3.2}$$
$$(\exists x_1. \cdots \exists x_n.\mathbf{v}) \wedge \mathbf{u} = \exists x_1. \cdots \exists x_n.(\mathbf{v} \wedge \mathbf{u}) \tag{3.3}$$

Note that the convention on syntactic variables dictate that none of the variables x_i can appear free in **u**. The two rules can be captured by repeated applications of the following two special cases of the rules

$$\mathbf{u} \wedge (\exists x.\mathbf{v}) = \exists x.(\mathbf{u} \wedge \mathbf{v}) \tag{3.4}$$
$$(\exists x.\mathbf{v}) \wedge \mathbf{u} = \exists x.(\mathbf{v} \wedge \mathbf{u}) \tag{3.5}$$

and these are what we will actually implement. We choose to implement these easier rules because checking that each x_i is free in **u** would be an expensive exercise.

Interestingly, it is actually quite important to get the order of **u** and **v** right in the conjunction. For example, implementing

$$(\exists x.\mathbf{v}) \wedge \mathbf{u} = \exists x.(\mathbf{u} \wedge \mathbf{v})$$

instead of Statement 3.5 will seriously slow down the predicate *permute* (see Sect. 6.3).

73 ⟨term::function definitions 32e⟩+≡

```
bool term::simplifyConjunction2(term ∗ parent, unint id) {
        term ∗ t1 = lc()→rc(), ∗ t2 = rc();
        term ∗ sigma, ∗ other;
        if (t2→isApp() ∧ t2→lc()→isF(iSigma)) {
                sigma = t2; other = t1;
        } else if (t1→isApp() ∧ t1→lc()→isF(iSigma)) {
                sigma = t1; other = t2;
        } else return false;

        int var = sigma→rc()→fields[0]→cname;
        if (other→occursFree(var)) return false;

        ⟨simplifyConjunction2::create body 74⟩
        ⟨simplify update pointers 62a⟩
        return true;
}
```

Defines:
`simplifyConjunction2`, used in chunks 90 and 111.
Uses `iSigma` 145, `isApp` 30a, `isF` 30a, `lc` 30e, `occursFree` 47b, and `rc` 30e.

Comment 3.1.29. We could recycle $\exists x$ but choose not to.

74 ⟨simplifyConjunction2::create body 74⟩≡

```
term * con = newT2Args(F, iAnd);
if (sigma ≡ t2)
        con→initT2Args(other→reuse(), sigma→rc()→fields[1]→reuse());
else con→initT2Args(sigma→rc()→fields[1]→reuse(), other→reuse());
term * abs = new_term(ABS);
// abs->lc = new_term(V, var); abs->rc = con;
abs→insert(new_term(V, var)); abs→insert(con);
term * ret = new_term(APP);
// ret->lc = new_term(F, iSigma); ret->rc = abs;
ret→insert(new_term(F, iSigma));  ret→insert(abs);
```

Uses `iAnd` 145, `iSigma` 145, `initT2Args` 34a, `insert` 30e, `lc` 30e, `newT2Args` 33c, `new_term` 40a, `rc` 30e, and `reuse` 43d.

Comment 3.1.30. Example execution of `simplifyConjunction2`.

```
Query: ((&& (sigma \x1.(sigma \x2.v))) u)
Time = 1 Answer: (sigma \x1.((&& u) (sigma \x2.v)))
Time = 2 Answer: (sigma \x1.(sigma \x2.((&& u) v)))
```

Comment 3.1.31. The use of Statements 3.4 and 3.5 introduces a peculiar behaviour into Escher in that the same query, when asked using two different variable names, can result in two different computation sequences. To illustrate this, consider the following statement:

$$f : Int \times (Int \to \Omega) \to \Omega$$
$$f(x, s) = (x \leq 8) \wedge \exists z.(z \in s \wedge (prime\ z)).$$

Now, if we ask Escher to compute the value of $f(y, \{2, 3\})$, we get

$$\begin{aligned} f(y, \{2,3\}) &= (y \leq 8) \wedge \exists z.(z \in \{2,3\} \wedge (prime\ z)) \\ &= \exists z.((y \leq 8) \wedge z \in \{2,3\} \wedge (prime\ z)) \\ &= \ldots \\ &= \exists z.((y \leq 8) \wedge (z = 2)) \\ &= (y \leq 8). \end{aligned}$$

However, if we ask Escher to compute the value of $f(z, \{2, 3\})$, we will get

$$\begin{aligned} f(z, \{2,3\}) &= (z \leq 8) \wedge \exists z.(z \in \{2,3\} \wedge (prime\ z)) \\ &= \ldots \\ &= (z \leq 8) \wedge \exists z.(z = 2) \\ &= (z \leq 8) \wedge \top \\ &= (z \leq 8). \end{aligned}$$

The two computation sequences are different from the second step onwards. The second sequence takes one step longer than the first. Just about the only comforting thing is that the end results are equivalent. The questions then are

1. Should we retain Statements 3.4 and 3.5 in the booleans module? Judging from the (limitted set of) test programs I have, there is no actual need for the two statements. In general, I believe they make certain computations go faster, although one can also show instances where they actually make things go slightly slower.

2. If we retain the two statements, should we modify the convention so that a variable renaming is done to make them applicable in cases where they cannot be applied?

Comment 3.1.32. This function implements the following existential rules:

$$\exists x_1.\cdots\exists x_n.\top = \top \tag{3.6}$$

$$\exists x_1.\cdots\exists x_n.\bot = \bot \tag{3.7}$$

$$\exists x_1.\cdots\exists x_n.(\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y}) = \exists x_2.\cdots\exists x_n.(\mathbf{x}\{x_1/\mathbf{u}\} \wedge \top \wedge \mathbf{y}\{x_1/\mathbf{u}\}). \tag{3.8}$$

The other rules are implemented in the Booleans module. See Comment 6.1.1. I suppose they can be implemented here if we really need to maximise efficiency at the price of complicated code.

Comment 3.1.33. We first check whether the current term starts with $\exists x_1 \cdots \exists x_n$. We then move to the subterm after $\exists x_1 \cdots \exists x_n$ and perform surgery on it if possible.

75a ⟨term::function definitions 32e⟩+≡

```
bool term::simplifyExistential(term ∗ parent, unint id) {
        if (fields[0]→isF(iSigma) ≡ false) return false;

        term ∗ ret = NULL; term ∗ p = NULL;
        int var = rc()→fields[0]→cname;
        substitution bind;

        ⟨simplifyExistential::move to the body 75b⟩

        ⟨simplifyExistential::case one and two 75c⟩
        ⟨simplifyExistential::tricky case 76a⟩

simplifyExistential_cleanup:
        ⟨simplify update pointers 62a⟩
        return true;
}
```

Defines:
`simplifyExistential`, used in chunks 90 and 111.
Uses `iSigma` 145, `isF` 30a, `rc` 30e, and `substitution` 51.

Comment 3.1.34. The following allows us to move past the remaining $\exists x_i$ to get to the body of the term.

75b ⟨simplifyExistential::move to the body 75b⟩≡

```
term ∗ body = fields[1]→fields[1];
while (body→isApp() ∧ body→lc()→isF(iSigma)) {
        body = body→rc()→fields[1];
}
```

Uses `iSigma` 145, `isApp` 30a, `isF` 30a, `lc` 30e, and `rc` 30e.

Comment 3.1.35. This handles Statements 3.6 and 3.7. Completeness of specification is not an issue here. The two statements can be captured by repeated application of the statements

$$\exists x.\top = \top \text{ and } \exists x.\bot = \bot.$$

Making them part of the internal simplification routine gives us efficiency advantages.

75c ⟨simplifyExistential::case one and two 75c⟩≡

```
if (body→isD(iTrue) ∨ body→isD(iFalse)) {
        ret = body→reuse(); goto simplifyExistential_cleanup; }
```

Uses `iFalse` 145, `iTrue` 145, `isD` 30a, and `reuse` 43d.

Comment 3.1.36. We next discuss Statement 3.8. The pattern in the head of Statement 3.8 should be interpreted in the same way as the corresponding pattern in the conjunction rule described in Comment 3.1.16. Note that the statement is slightly different from that given in [Llo03], which takes the following form:

$$\exists x_1.\cdots\exists x_n.(\mathbf{x}\wedge(x_i=\mathbf{u})\wedge\mathbf{y}) = \exists x_1.\cdots\exists x_{i-1}.\exists x_{i+1}.\cdots\exists x_n.(\mathbf{x}\{x_i/\mathbf{u}\}\wedge\mathbf{y}\{x_i/\mathbf{u}\}).$$

First, restricting x_i to be x_1 as we did incurs a small computational cost in that we need to move to the subterm starting with $\exists x_i$ during pattern matching to apply Statement 3.8. In return, we can write simpler code. The second change is that instead of dropping the term $(x_1 = \mathbf{u})$, we put a $\top$ in its place. The two expressions are equivalent, of course. The advantage of that is the same: we can write simpler code. Another advantage of this latter change is that, unlike the original statement, we do end up with a natural special case. (See Comment 3.1.37.)

76a
```
⟨simplifyExistential::tricky case 76a⟩≡
  ⟨simplifyExistential::tricky case::special case 76b⟩
  ⟨simplifyExistential::tricky case::general case 76c⟩
```

Comment 3.1.37. A special case of Statement 3.8 is the following:

$$\exists x_1.\cdots\exists x_n.(x_1=\mathbf{u}) = \exists x_2.\cdots\exists x_n.\top. \tag{3.9}$$

The body of the statement can be further simplified to $\top$, of course.

76b
```
⟨simplifyExistential::tricky case::special case 76b⟩≡
  if (body→isEq(var)) { ret = new_term(D, iTrue);
                        goto simplifyExistential_cleanup; }
```
Uses `iTrue` 145, `isEq` 72a 77b, and `new_term` 40a.

Comment 3.1.38. In the general case, we first check that the body has the overal form $t_1 \wedge t_2$. Then we attempt to find in the body an equation that instantiates the first quantified variable and replaces it with $\top$. (This is performed all at the same time by `replaceEq`.) If that operation is successful, we perform term substitutions on the body and then get rid of the first quantification.

76c
```
⟨simplifyExistential::tricky case::general case 76c⟩≡
  if (body→isFunc2Args(iAnd) ≡ false) return false;
  p = body→replaceEq(var); if (p ≡ NULL) return false;

  bind.first = p→lc()→rc()→cname; bind.second = p→fields[1];
  body→subst(bind);
  p→freememory();

  ret = rc()→fields[1]→reuse();
```
Uses `iAnd` 145, `isFunc2Args` 32f, `lc` 30e, `rc` 30e, `replaceEq` 77a, `reuse` 43d, and `subst` 52a.

Comment 3.1.39. The function `replaceEq` finds a subterm of the form $(x = t)$ embedded conjunctively inside a term (with the help of `isEq`), replaces it with $\top$ and then returns a pointer to $(x = t)$.

Comment 3.1.40. We assume that the calling term is a conjunction of the form $t_1 \wedge t_2$. If t_1 is a variable-instantiating equation, we return. Otherwise, we recurse on t_1 if it has the right (conjunctive) form. Then we do the same on t_2. (See also Comment 3.1.22.)

77a ⟨term::function definitions 32e⟩+≡

```
term ∗ term::replaceEq(int var) {
        term ∗ p = NULL;
        term ∗ t1 = lc()→rc();
        if (t1→isEq(var)) {
                lc()→fields[1] = new_term(D, iTrue); return t1; }
        if (t1→isFunc2Args(iAnd)) { p = t1→replaceEq(var); if (p) return p; }

        term ∗ t2 = rc();
        if (t2→isEq(var)) { fields[1] = new_term(D, iTrue); return t2; }
        if (t2→isFunc2Args(iAnd)) { p = t2→replaceEq(var); if (p) return p; }
        return NULL;
}
```

Defines:
`replaceEq`, used in chunks 76c, 79b, and 111.
Uses `iAnd` 145, `iTrue` 145, `isEq` 72a 77b, `isFunc2Args` 32f, `lc` 30e, `new_term` 40a, and `rc` 30e.

Comment 3.1.41. This function checks whether the current term has the form $(x = t)$ where x is the input variable and t is a term such that x does not occur free in t. If the current term has the form $(t = x)$ where x does not occur free in t, we need to swap the two arguments because procedures that call `replaceEq` expect the variable x to be on the LHS of the equation.

77b ⟨term::function definitions 32e⟩+≡

```
bool term::isEq(int x) {
        if (isFunc2Args(iEqual) ≡ false) return false;
        term ∗ t1 = lc()→rc(), ∗ t2 = rc();

        if (t1→isVar(x) ∧ t2→occursFree(x) ≡ false) return true;
        if (t2→isVar(x) ∧ t1→occursFree(x) ≡ false) {
                ⟨isEq::switch t1 and t2 72b⟩
                return true; }
        return false;
}
```

Defines:
`isEq`, used in chunks 71b, 76–78, and 111.
Uses `iEqual` 145, `isFunc2Args` 32f, `isVar` 30a, `lc` 30e, `occursFree` 47b, and `rc` 30e.

Comment 3.1.42. Example execution of `simplifyExistential`.

```
Query: (sigma \x3.(sigma \x2.(sigma \x1.((== x3) t1))))
Time = 1 Answer: True

Query: (sigma \x3.(sigma \x.(sigma \y.(&& y (&& (== x T1) (&& (== t2 y) x))))))
Time = 1 Answer: (sigma \x3.(sigma \y.(&& y (&& True (&& (== t2 y) T1)))))
Time = 2 Answer: (sigma \x3.(&& t2 (&& True (&& True T1))))
Time = 3 Answer: (sigma \x3.(&& t2 (&& True T1)))
Time = 4 Answer: (sigma \x3.(&& t2 T1))
```

Comment 3.1.43. This function implements the following universal rules:

$$\forall x_1.\cdots\forall x_n.(\bot \to \mathbf{u}) = \top \tag{3.10}$$

$$\forall x_1.\cdots\forall x_n.(\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y} \to \mathbf{v}) = \forall x_2.\cdots\forall x_n.((\mathbf{x} \wedge \top \wedge \mathbf{y} \to \mathbf{v})\{x_1/\mathbf{u}\}). \tag{3.11}$$

Statement 3.11 is equivalent to the following rule given in [Llo03]:

$$\forall x_1.\cdots\forall x_n.(\mathbf{x} \wedge (x_1 = \mathbf{u}) \wedge \mathbf{y} \to \mathbf{v}) = \forall x_1.\cdots\forall x_{i-1}.\forall x_{i+1}.\cdots\forall x_n.((\mathbf{x} \wedge \mathbf{y} \to \mathbf{v})\{x_i/\mathbf{u}\}).$$

78a ⟨term::function definitions 32e⟩+≡

```
bool term::simplifyUniversal(term ∗ parent, unint id) {
        if (lc()→isF(iPi) ≡ false) return false;

        int var = rc()→fields[0]→cname;
        ⟨simplifyUniversal::check the form of body 78b⟩
        ⟨simplifyUniversal::true statement 78c⟩
        ⟨simplifyUniversal::special case 78d⟩
        ⟨simplifyUniversal::general case 79b⟩
}
```

Defines:
`simplifyUniversal`, used in chunks 90 and 111.
Uses `iPi` 145, `isF` 30a, `lc` 30e, and `rc` 30e.

Comment 3.1.44. We move past the remaining ∀s to get to the body and check whether it has the form $t_1 \to t_2$. If so, we move to t_1.

78b ⟨simplifyUniversal::check the form of body 78b⟩≡

```
term ∗ body = rc()→fields[1];
while (body→isApp() ∧ body→lc()→isF(iPi))
        body = body→rc()→fields[1];
if (body→isFunc2Args(iImplies) ≡ false) return false;
term ∗ t1 = body→lc()→rc();
```

Uses `iImplies` 145, `iPi` 145, `isApp` 30a, `isF` 30a, `isFunc2Args` 32f, `lc` 30e, and `rc` 30e.

Comment 3.1.45. This code chunk implements Statement 3.10.

78c ⟨simplifyUniversal::true statement 78c⟩≡

```
if (t1→isD(iFalse)) { term ∗ ret = new_term(D, iTrue);
                      ⟨simplify update pointers 62a⟩
                      return true; }
```

Uses `iFalse` 145, `iTrue` 145, `isD` 30a, and `new_term` 40a.

Comment 3.1.46. A special case of Statement 3.11 is the following:

$$\forall x_1.\cdots\forall x_n.((x_1 = \mathbf{u}) \to \mathbf{v}) = \forall x_2.\cdots\forall x_n.(\top \to \mathbf{v})\{x_1/\mathbf{u}\} = \forall x_2.\cdots\forall x_n.\mathbf{v}\{x_1/\mathbf{u}\}.$$

78d ⟨simplifyUniversal::special case 78d⟩≡

```
if (t1→isEq(var)) {
        term ∗ t2 = body→rc();
        substitution bind(t1→lc()→rc()→cname, t1→rc());
        t2→subst(bind);
        body→replace(t2→reuse());
        t2→freememory();
        ⟨simplifyUniversal::change end game 79a⟩
}
```

Uses `isEq` 72a 77b, `lc` 30e, `rc` 30e, `replace` 42b, `reuse` 43d, `subst` 52a, and `substitution` 51.

Comment 3.1.47. After changing the body, we remove the quantifier of x_1 and return.

79a ⟨simplifyUniversal::change end game 79a⟩≡
```
term ∗ ret = rc()→fields[1]→reuse();
⟨simplify update pointers 62a⟩
return true;
```
Uses `rc` 30e and `reuse` 43d.

Comment 3.1.48. We first check whether the LHS of → has the form $t_3 \wedge t_4$. If so, we seek to find an equation instantiating the first quantified variable and replace it with $\top$. (This is again done using `replaceEq`.) Then we make the necessary term substitutions and return.

79b ⟨simplifyUniversal::general case 79b⟩≡
```
if (t1→isFunc2Args(iAnd) ≡ false) return false;
term ∗ p = t1→replaceEq(var); if (p ≡ NULL) return false;

substitution bind(p→lc()→rc()→cname, p→fields[1]);
body→subst(bind);
p→freememory();

⟨simplifyUniversal::change end game 79a⟩
```
Uses `iAnd` 145, `isFunc2Args` 32f, `lc` 30e, `rc` 30e, `replaceEq` 77a, `subst` 52a, and `substitution` 51.

Comment 3.1.49. Example execution of `simplifyUniversal`.

```
Query: (pi \x2.(pi \x1.(pi \x3.((implies ((== x1) t1)) ((&& x1) x1)))))
Time = 1 Answer: (pi \x2.(pi \x3.((&& t1) t1)))

Query: (pi \x3.(pi \x1.(pi \x2.((implies ((&& ((== True) x2))
                                  ((&& ((== x1) True)) ((&& x1) x2)))) t1))))
Time = 1 Answer: (pi \x3.(pi \x2.((implies ((&& ((== True) x2))
                                  ((&& True) ((&& True) x2)))) t1)))
Time = 2 Answer: (pi \x3.((implies ((&& True) ((&& True) (&& True True)))) t1))
Time = 3 Answer: (pi \x3.((implies ((&& True) ((&& True) True))) t1))
Time = 4 Answer: (pi \x3.((implies ((&& True) True)) t1))
Time = 5 Answer: (pi \x3.((implies True) t1))
Time = 6 Answer: (pi \x3.t1)
```

Comment 3.1.50. This next function implements the rules

$\Box_i \mathbf{t} = \mathbf{t}$ if $\mathbf{t}$ is rigid.

$(\Box_i \mathbf{s}\ \mathbf{t}) = \Box_i(\mathbf{s}\ \mathbf{t})$ if $\mathbf{t}$ is rigid.

80 ⟨term::function definitions 32e⟩+≡

```
  bool term::simplifyModalTerms(term ∗ parent, unint id) {
        if (isModal()) {
              if (¬isRigid()) return false;
              term ∗ ret = fields[0]→reuse();
              ⟨simplify update pointers 62a⟩
              return true;
        }
        if (isApp() ∧ lc()→isModal() ∧ lc()→fields[0]→isF()) {
              if (¬rc()→isRigid()) return false;
              term ∗ ret = new_term(MODAL);
              ret→modality = lc()→modality;
              term ∗ temp = new_term(APP);
              ÷∗ temp→lc = lc→lc→reuse();
              temp→rc = rc→reuse();
              ret→lc = temp; ∗÷
              temp→insert(lc()→fields[0]→reuse());
              temp→insert(rc()→reuse());
              ret→insert(temp);
              ⟨simplify update pointers 62a⟩
              return true;
        }
        return false;
  }
```

Defines:
  `simplifyModalTerms`, used in chunk 111.

Uses `insert` 30e, `isApp` 30a, `isF` 30a, `isModal` 30a, `lc` 30e, `new_term` 40a, `rc` 30e, and `reuse` 43d.

3.1.2 Computing and Reducing Candidate Redexes

Comment 3.1.51. We now describe the function `reduce` that dynamically computes the candidate redexes inside a term (in the leftmost outermost order) and tries to reduce them.

Definition 3.1.52. A *redex* of a term t is an occurrence of a subterm of t that is α-equivalent to an instance of the head of a statement.

Comment 3.1.53. Informally, given a term t, every term s represented by a subtree of the syntax tree representing t, with the exception of the variable directly following a λ, is a subterm of t. The path expression leading from the root of the syntax tree representing t to the root of the syntax tree representing s is called the occurrence of s. For exact formal definitions of these concepts, see [Llo03, pp. 46].

Comment 3.1.54. There is an easy way to count the number of subterms in a term t. A token is either a left bracket '(', a variable, a constant, or an expression of the form λx for some variable x. The number of subterms in a term t is simply the number of tokens in (the string representation) of t. For example, the term $((f\ (1,(2,3),4))\ \lambda x.(g\ x))$ has 13 subterms.

Comment 3.1.55. There are obviously many subterms. For redex testing, it is important that we rule out as many of these as posible up front. The following result is a start.

Proposition 3.1.56. *Let t be a term. A subterm r of t cannot be a redex if any one of the following is true:*

1. *r is a variable;*
2. *$r = \lambda x.t$ for some variable x and term t;*
3. *$r = D\, t_1 \dots t_n$, $n \geq 0$, where D is a data constructor of arity $m \geq n$, and each t_i is a term;*
4. *$r = (t_1, \dots, t_n)$ for some $n \geq 0$.*

Proof. Consider any statement $h = b$ in the program. By definition, h has the form $f\ t_1 \dots t_n$, $n \geq 0$ for some function f. In each of the cases above, $r \neq h\theta$ for any θ and therefore r cannot be a redex. ☐

81a ⟨cannot possibly be a redex 81a⟩≡

```
if (isAString()) return false;
if (tag ≡ V ∨ tag ≡ D) return false;
if (tag ≡ ABS ∨ tag ≡ PROD ∨ tag ≡ MODAL ∨ isData()) goto not_a_redex;
```

Uses `isAString` 33a and `isData` 81b 82d.

Comment 3.1.57. This function checks whether the current term has the form $D\, t_1 \dots t_n$, $n \geq 1$, where D is a data constructor of arity $m \geq n$ and each t_i is a term.

81b ⟨term::function declarations 30a⟩+≡

```
bool isData();
```

Defines:
`isData`, used in chunk 81a.

Comment 3.1.58. When we see a term $t = D\, t_1 \dots t_n$, $n \geq 1$, where D is a data constructor of arity n and each t_i is a term, we can immediately deduce that any prefix of t cannot be a redex. The variable `is_data` is used to store this information.

81c ⟨term bool parts 31a⟩+≡

```
bool is_data;
```

81d ⟨term init 29d⟩+≡

```
is_data = false;
```

82a ⟨heap term init 29e⟩+≡

```
ret→is_data = false;
```

82b ⟨term clone parts 29f⟩+≡

```
ret→is_data = is_data;
```

Comment 3.1.59. We can probably sometimes recycle `t->is_data` here, but decided to always use the safe `false` value instead.

82c ⟨term replace parts 29g⟩+≡

```
is_data = false;
```

Comment 3.1.60. If the current term is a data term, then the left subterm of the current term is also a data term.

82d ⟨term::function definitions 32e⟩+≡

```
bool term::isData() {
        if (tag ≠ APP) return false;
        if (is_data) { fields[0]→is_data = true; return true; }
        if (spineTip()→isD()) {
                is_data = true; fields[0]→is_data = true; return true; }
        return false;
}
```

Defines:
`isData`, used in chunk 81a.
Uses `isD` 30a and `spineTip` 32e.

Comment 3.1.61. Proposition 3.1.56 allows us to focus on terms of the form $(f\ t_1 \ldots t_n)$, $n \geq 0$, in finding redexes. What else do we know that can be used to rule out as potential redexes subterms of this form?

Given a function symbol f, we define the effective arity of f to be the number of argument(s) f is applied to in the head of any statement in the program. Clearly, given a term $t = (f\ t_1 \ldots t_n)$, $n \geq 0$, if n is not equal to the effective arity of f, then t cannot possibly be a redex.

82e ⟨cannot possibly be a redex 2 82e⟩≡

```
if (tag ≡ F ∧ getFuncEArity(cname).first ≠ 0) return false;
if (isFuncNotRightArgs()) goto not_a_redex;
```

Uses `getFuncEArity` 155b and `isFuncNotRightArgs` 82f 83e.

Comment 3.1.62. This function checks whether the current term, which is an application node, is a function applied to the right number of arguments (its effective arity). The number of arguments can be more than the effective arity of the leftmost function symbol. The term $(((remove\ s)\ t)\ x)$ is one such example.

82f ⟨term::function declarations 30a⟩+≡

```
bool isFuncNotRightArgs();
```

Defines:
`isFuncNotRightArgs`, used in chunk 82e.

Comment 3.1.63. This is used to capture the fact that every prefix of a function application term that does not have enough arguments will not have enough arguments.

82g ⟨term bool parts 31a⟩+≡

```
bool notEnoughArgs;
```

83a
```
⟨term init 29d⟩+≡
  notEnoughArgs = false;
```

83b
```
⟨heap term init 29e⟩+≡
  ret→notEnoughArgs = false;
```

83c
```
⟨term clone parts 29f⟩+≡
  ret→notEnoughArgs = notEnoughArgs;
```

Comment 3.1.64. We can probably safely recycle `t->notEnoughArgs` here.

83d
```
⟨term replace parts 29g⟩+≡
  notEnoughArgs = false;
```

Comment 3.1.65. If we have an excess of arguments, return true. Otherwise, if we have an under supply of arguments, mark the `notEnoughArgs` flag of the left subterm and then return true.

83e
```
⟨term::function definitions 32e⟩+≡
  bool term::isFuncNotRightArgs() {
        if (tag ≠ APP) return false;
        if (notEnoughArgs) { fields[0]→notEnoughArgs = true; return true; }
        spineTip(); // can use spineTip(int & x) here
        int numargs = spinelength-1;
        if (spinetip→isF() ≡ false) return false;
        pair<int,int> arity = getFuncEArity(spinetip→cname);
        ⟨isFuncNotRightArgs::error handling 83f⟩
        // if (arity > numargs) {
        if (numargs < arity.first) {
              notEnoughArgs = true; fields[0]→notEnoughArgs = true;
              return true;
        }
        // return (arity < numargs);
        return (numargs > arity.second);
  }
```

Defines:
  `isFuncNotRightArgs`, used in chunk 82e.
Uses `getFuncEArity` 155b, `isF` 30a, and `spineTip` 32e.

Comment 3.1.66. If the function is unknown, then we just return true as a conservative measure.

83f
```
⟨isFuncNotRightArgs::error handling 83f⟩≡
  if (arity.first ≡ -1) return true;
```

Comment 3.1.67. We now describe the `reduce` function. We compute the subterms one by one in the left-to-right, outermost to innermost order. For each subterm, we first determine whether it can possibly be a candidate redex. If not, we proceed to the next subterm. Otherwise, we attempt to match and reduce it using `try_match_n_reduce`. If this is successful, we return true. Otherwise, we proceed to the next subterm. The parameter `tried` records the total number of candidate redexes actually tried by this function. All the other parameters are needed only by `try_match_n_reduce`.

84a ⟨term::function declarations 30a⟩+≡

```
bool reduce(vector<int> mpath, term * parent, unint cid,
            term * root, int & tried);
bool reduce(vector<int> mpath, term * parent, unint cid,
            term * root, int & tried, bool lresort);
bool reduceRpt(int maxstep, int & stepsTaken);
bool reduceRpt() { int x; return reduceRpt(0, x); }
```

Defines:
`reduce`, used in chunks 84–87, 89a, 91b, and 93b.
`reduceRpt`, never used.

84b ⟨term::function definitions 32e⟩+≡

```
bool term::reduce(vector<int> mpath, term * parent, unint cid,
                  term * root, int & tried) {
        if (reduce(mpath,parent,cid,root,tried,false)) return true;
        // cerr << "Trying last resort rules" << endl;// setSelector(osel);
        return reduce(mpath,parent,cid,root,tried,true);
}
```

Uses `reduce` 84a 85a and `setSelector` 164 165.

85a ⟨term::function definitions 32e⟩+≡

```
bool term::reduce(vector<int> mpath, term ∗ parent, unint cid,
                        term ∗ root, int & tried, bool lr) {
        ⟨cannot possibly be a redex 81a⟩
  #ifdef ESCHER
        ⟨cannot possibly be a redex 2 82e⟩
  #endif
        tried++;
        // if (try_disruptive(mpath, this, parent, cid, root, tried))
        //       return true;
        if (try_match_n_reduce(mpath,this,parent,cid,root,tried,lr))
              return true;

  not_a_redex:
        if (tag ≡ ABS)
              return fields[1]→reduce(mpath,this,1, root, tried,lr);
        if (tag ≡ MODAL) {
              // return false; // to begin with, Escher does not handle this.
              mpath.push_back(modality);
              return fields[0]→reduce(mpath,this,0,root, tried, lr);
        }
        if (tag ≡ APP) {
              ⟨reduce::small APP optimization 85b⟩
              if (lc()→reduce(mpath, this,0, root, tried, lr))
                    return true;
              return rc()→reduce(mpath,this,1, root, tried, lr);
        }
        if (tag ≡ PROD) {
              unint dimension = fieldsize;
              for (unint i=0; i≠dimension; i++)
                    if (fields[i]→reduce(mpath, this, i, root, tried, lr))
                          return true;
              return false;
        }
        if (tag ≡ F) return false;
  #ifdef ESCHER
        setSelector(STDERR); cerr ≪ "term = "; print(); ioprintln();
        cerr ≪ "tag = " ≪ tag ≪ endl; assert(false);
  #endif
        return false;
  }
```

Defines:
`reduce`, used in chunks 84–87, 89a, 91b, and 93b.
Uses `ioprintln` 164 165, `lc` 30e, `rc` 30e, `setSelector` 164 165, and `try_match_n_reduce` 87.

Comment 3.1.68. When we see a term of the form $(f\ t)$ where f has effective arity greater than 1, we can immediately deduce that f cannot be a redex. This would have been picked out if we recurse on f, but we can save a call to `getFuncEArity` by having a special case here.

85b ⟨reduce::small APP optimization 85b⟩≡

```
  #ifdef ESCHER
  if (lc()→isF() ∧ notEnoughArgs)
        return rc()→reduce(mpath, this, 1, root, tried, lr);
  #endif
```

Uses `isF` 30a, `lc` 30e, `rc` 30e, and `reduce` 84a 85a.

Comment 3.1.69. It is easy to add code to calculate the occurrence of each subterm if this information is desired.

Comment 3.1.70. The function `reduceRpt` reduces an expression repeatedly until no further reduction is possible. The return value is true if the term is modified in the process; false otherwise.

86a ⟨term::function definitions 32e⟩+≡

```
bool term::reduceRpt(int maxstep, int & stepsTaken) {
        int tried = 0; vector<int> modalPath;
        bool reduced = true; bool rewritten = false;
        int starttime = ltime;
        while (reduced) {
                reduced = reduce(modalPath, NULL, 0, this, tried);
                if (reduced) rewritten = true;
                if (tag ≡ D) break;
                if ((maxstep > 0) ∧ (ltime - starttime) > maxstep) break;
                if (interrupted) break;
        };
        stepsTaken = ltime - starttime;
        return rewritten;
}
```

Defines:
`reduceRpt`, never used.
Uses `reduce` 84a 85a.

Comment 3.1.71. The function `reduce` uses the following function to try and match and reduce a candidate redex. The function `try_match_n_reduce` works as follows. Given a candidate redex, we first examine whether it can be simplified using the internal simplification routines of Escher. If so, we are done and can return. Otherwise, we try to pattern match (using `redex_match`) the candidate redex with the head of suitable statements in the program. If the head of a statement $h = b$ is found to match with `candidate` using some term substitution θ, then we construct $b\theta$ and replace `candidate` with $b\theta$. Depending on whether `candidate` has a parent, we either only need to redirect a pointer or we need to replace in place.

86b ⟨terms.cc::local functions 33c⟩+≡

```
#include "global.h"
#include "pattern-match.h"
static int nestingdepth = 0;
⟨try match 87⟩
⟨try disruptive 93b⟩
```

```
87  ⟨try match 87⟩≡
      bool do_local_search = true;
      bool try_match_n_reduce(vector<int> mpath, term ∗ candidate,
                              term ∗ parent,unint cid, term ∗ root,
                              int & tried, bool lastresort) {
          vector<substitution> theta; ÷∗ this cannot be made global because of
                                         eager statements ∗÷
          ⟨debug matching 1 92d⟩
          ⟨try match::different simplifications 90⟩
          ⟨try match::try cached statements first 88a⟩

          candidate→spineTip();
          if (¬candidate→spinetip→isF()) return false;
          int anchor = candidate→spinetip→cname;
          if (anchor ≥ (int)grouped_statements.size()) return false;
          statementType ∗ sts = grouped_statements[anchor];
          if (sts ≡ NULL) return false;
          while (sts ≠ NULL) {
              if (sts→lastresort ≠ lastresort) { sts = sts→next; continue;}
              if ((candidate→spinelength -1) ≠ sts→numargs)
                  { sts = sts→next; continue; }
              theta.clear();
              term ∗ head = sts→stmt→lc()→rc();
              term ∗ body = sts→stmt→rc();
              ⟨debug matching 2 92e⟩
              if (redex_match(head, candidate, theta)) {
                  ⟨try match::eager statements 91b⟩
                  ltime++;
                  ⟨try match::unimportant things 91c⟩
                  term ∗ temp = NULL;
                  if (sts→noredex) temp = body→reuse();
                  else {
                      temp = body→clone();
                      temp→subst(theta);
                      ⟨try match::reduce temp to simplest form 89a⟩
                  }
                  ⟨try match::put reduct in place 88b⟩
                  ⟨try match::output answer 92b⟩
                  return true;
              }
              ⟨debug matching 4 93a⟩
              sts = sts→next;
          }

          ÷∗ int bm_size = statements.size();
          for (int j=0; j<bm_size; j++) {
              if (statements[j].lastresort ≠ lastresort) continue;
              <try match::find special cases where no matching is required>
              theta.clear();
              term ∗ head = statements[j].stmt→lc()→rc();
              term ∗ body = statements[j].stmt→rc();
              <debug matching 2>
              if (redex_match(head, candidate, theta)) {
                  <try match::eager statements>
```

```
            ltime++;
            <try match::unimportant things>
            term * temp = body→clone();
            temp→subst(theta);
            <try match::reduce temp to simplest form>
            <try match::put reduct in place>
            <try match::output answer>
            return true;
        }
        <debug matching 4>
    } *÷
    return false;
}
```

Defines:
`try_match_n_reduce`, used in chunk 85a.
Uses `isF` 30a, `lc` 30e, `rc` 30e, `redex_match` 99a 99b 100a, `reduce` 84a 85a, `reuse` 43d, `spineTip` 32e, `subst` 52a, and `substitution` 51.

Comment 3.1.72. Certain (sub)computations are cached in the vector `cachedStatements` during run-time. We try out these cached statements first in simplifying a redex. The pattern matching operation used in this special case is just identity checking. Maybe we need to check for α-equivalence in general.

88a ⟨try match::try cached statements first 88a⟩≡

```
int cs_size = cachedStatements.size();
for (int j=0; j≠cs_size; j++) {
    term * head = cachedStatements[j]→stmt→lc()→rc();
    term * body = cachedStatements[j]→stmt→rc();
    theta.clear(); // vector<substitution> theta;
    if (candidate→equal(head) ∨ redex_match(head, candidate, theta)) {
        // cerr << "Using cached computation." << endl;
        term * temp = body→clone();
        if (theta.size()) { temp→subst(theta); }
        ltime++; cltime++;
        ⟨try match::put reduct in place 88b⟩
        return true;
    }
}
```

Uses `equal` 34b, `lc` 30e, `rc` 30e, `redex_match` 99a 99b 100a, `subst` 52a, and `substitution` 51.

Comment 3.1.73. This is how we put the reduct (`temp`) in place of the redex (`candidate`). Depending on whether `candidate` has a parent, we either change some pointers or do an in-place replacement.

88b ⟨try match::put reduct in place 88b⟩≡

```
if (parent) { parent→fields[cid] = temp; candidate→freememory(); }
else { candidate→replace(temp); temp→freememory(); }
```

Uses `replace` 42b.

Comment 3.1.74. In the proposed leftmost outermost reduction scheme, we need to find the leftmost outermost redex in t^* immediately after rewriting a subterm r in t with s to obtain $t^* = t[s/r]$. We are probably better off doing localised surgeries on terms. After one rewriting step, instead of looking for the next redex in t^*, we will try to reduce s as much as possible before jumping out to consider reducing t^*. This simple change in the redex selection order speeds things up tremendously, at no cost to correctness. We will have problems with list/set comprehension though, in particular infinite lists and infinite sets.

```
89a    ⟨try match::reduce temp to simplest form 89a⟩≡
         #ifdef ESCHER
         // do_local_search = false;
         if (¬outermost ∧ do_local_search ∧ ¬lastresort ∧ temp→tag ≠ D) {
                 // cerr << "reduce temp to simplest form\n";
                 do_local_search = false;
                 term ∗ temp3 = NULL; // term * temp2 = NULL;
                 if (optimise) { // temp2 = head->clone(); temp2->subst(theta);
                               // temp2->unshare(NULL, 1);
                               temp3 = temp→clone(); }
                 nestingdepth++;
                 int time_old = ltime;
                 temp→unshare(NULL, 1); // we should not need to do this operation
                 bool reduced = true;
                 int interval = 0; // this makes sure the local operation doesn't go
                                 // too deep, which can happen if we do this too early
                 while (reduced) {
                         reduced = temp→reduce(mpath,NULL,0, temp,tried);
                         if (temp→tag ≡ D) break;
                         if (interval++ > 500) break;
                 }
                 nestingdepth−−;
                 if (optimise) { ⟨try match::cache computation 89b⟩ }
                 do_local_search = true;
         }
         #endif
```

Uses `reduce` 84a 85a and `subst` 52a.

```
89b    ⟨try match::cache computation 89b⟩≡
         if (ltime - time_old > 30 ∧ head→isApp() ∧
             cacheFuncs.find(head→lc()→cname) ≠ cacheFuncs.end()) {
                 // int osel = getSelector(); setSelector(STDERR);
                 //    temp3->print(); ioprint(" = "); temp->print();
                 //    ioprint("; – "); ioprint(ltime-time_old); ioprintln();
                 //    ioprintln(); setSelector(osel);
                 statementType ∗ st = new statementType();
                 st→stmt = newT2Args(F, iEqual);
                 st→stmt→initT2Args(temp3, temp→clone());
                 temp3→labelStaticBoundVars(); // temp->labelStaticBoundVars();
                 st→stmt→collectSharedVars();
                 cachedStatements.push_back(st);
         } else temp3→freememory();
```

Uses `collectSharedVars` 94a, `getSelector` 164 165, `iEqual` 145, `initT2Args` 34a, `ioprint` 164 165, `ioprintln` 164 165, `isApp` 30a, `labelStaticBoundVars` 46f, `lc` 30e, `newT2Args` 33c, and `setSelector` 164 165.

Comment 3.1.75. The different simplification routines described in 3.1.1 are used here. We check the form of `candidate` before attempting to apply suitable routines.

```
90   ⟨try match::different simplifications 90⟩≡
       int msg = -5;
       if (candidate→isFunc2Args()) {
               int f = candidate→spineTip()→cname;
               if (f ≡ iEqual) {
                       if (candidate→simplifyEquality(parent, cid))
                               { msg = 1; ⟨simpl output 91a⟩ }
               } else if (f ≡ iAnd) {
                       if (candidate→simplifyConjunction())
                               { msg = 2; ⟨simpl output 91a⟩ }
                       if (candidate→simplifyConjunction2(parent, cid))
                               { msg = 3; ⟨simpl output 91a⟩ }
               }
               if (candidate→simplifyInequalities(parent, cid))
                       { msg = 4; ⟨simpl output 91a⟩ }

               if (candidate→simplifyArithmetic(parent, cid))
                       { msg = 5; ⟨simpl output 91a⟩ }
       }
       if (candidate→isApp()) {
               if (candidate→simplifyExistential(parent, cid))
                       { msg = 6; ⟨simpl output 91a⟩ }

               if (candidate→simplifyUniversal(parent, cid))
                       { msg = 7; ⟨simpl output 91a⟩ }

               if (candidate→betaReduction(parent, cid))
                       { msg = 8; ⟨simpl output 91a⟩ }

               if (candidate→simplifyMath(parent, cid))
                       { msg = 9; ⟨simpl output 91a⟩ }
       }
```

Uses `betaReduction` 69a, `iAnd` 145, `iEqual` 145, `isApp` 30a, `isFunc2Args` 32f, `simplifyArithmetic` 65, `simplifyConjunction` 71a, `simplifyConjunction2` 73, `simplifyEquality` 61, `simplifyExistential` 75a, `simplifyInequalities` 67, `simplifyMath` 68, `simplifyUniversal` 78a, and `spineTip` 32e.

Comment 3.1.76. The redex is marked out in the answer. We do not print the term before the simplification; that would be too messy though.

```
91a  ⟨simpl output 91a⟩≡
       ltime++;
       //if (verbose && ltime % 100 == 0) {
       if (verbose) {
              int osel = getSelector(); setSelector(STDOUT);
              ioprint("Time = "); ioprintln(ltime);
              switch (msg) {
              case 1: ioprint(eqsimpl); break;
              case 2: ioprint(andsimpl); break;
              case 3: ioprint(and2simpl); break;
              case 4: ioprint(ineqsimpl); break;
              case 5: ioprint(arsimpl); break;
              case 6: ioprint(exsimpl); break;
              case 7: ioprint(uvsimpl); break;
              case 8: ioprint(betasimpl); break;
              case 9: ioprint(mathsimpl); break;
              }
              candidate→redex = true;
              ioprint("Answer: "); root→print(); ioprint("\n\n");
              candidate→redex = false; setSelector(osel);
        }
       return true;
```

Uses `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

Comment 3.1.77. We now look at how eager statements are handled. When we matched a subterm of the form $(f\ t_1 \ldots t_n)$ with the head of a statement that is to be evaluated eagerly, we proceed to evaluate the arguments t_1 to t_n first. The whole expression can only be rewritten if none of the t_is contain a redex.

```
91b  ⟨try match::eager statements 91b⟩≡
       if (sts→eager ∧ candidate→isApp()) {
              ÷∗ try reduce the arguments first, return
                true if any one can be reduced ∗÷
              for (int i=candidate→spinelength-1; i≠0; i−−) {
                     /∗ go to argument spinelength - i argument ∗/
                     term ∗ arg = candidate;
                     for (int j=1; j≠i; j++) arg = arg→lc();
                     if (arg→rc()→reduce(mpath,arg,1,root, tried))
                            return true;
              }
       }
```

Uses `isApp` 30a, `lc` 30e, `rc` 30e, and `reduce` 84a 85a.

Comment 3.1.78. We are done talking about important things. We now list the not-so-important things like reporting and debugging checks.

```
91c  ⟨try match::unimportant things 91c⟩≡
       ⟨try match::debugging code 1 92c⟩
       ⟨debug matching 3 92f⟩
       ⟨try match::output pattern matching information 92a⟩
```

92a ⟨try match::output pattern matching information 92a⟩≡

```
            //if (verbose && ltime % 100 == 0) {
  if (verbose) {
        int osel = getSelector(); setSelector(STDOUT);
        ioprint("Time = "); ioprintln(ltime);
        ioprint("Matched "); head→print(); ioprintln(); // ioprint(" and ");
        // // candidate->print();
        // // ioprint("\nReplacing with "); body->print(); ioprint(' ');
        // // printTheta(theta);

        candidate→redex = true;
        ioprint("Query:  "); root→print(); ioprintln();
        candidate→redex = false; setSelector(osel);
  }
```

Uses `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, `printTheta` 109e 110a, and `setSelector` 164 165.

92b ⟨try match::output answer 92b⟩≡

```
  //if (verbose && ltime % 100 == 0) {
  if (verbose) {
        int osel = getSelector(); setSelector(STDOUT);
        ioprint("Answer: "); root→print(); ioprint("\n\n"); setSelector(osel);}
```

Uses `getSelector` 164 165, `ioprint` 164 165, and `setSelector` 164 165.

Comment 3.1.79. This is a simple check to make sure the candidate redex and the instantiated head are really α-equivalent.

92c ⟨try match::debugging code 1 92c⟩≡

```
  #ifdef MAIN_DEBUG1

  term * head1 = head→clone();
  head1→subst(theta);
  head1→applySubst();
  assert(head1→equal(candidate));

  #endif
```

Uses `equal` 34b and `subst` 52a.

Comment 3.1.80. These code allows us to track what is going on during matching.

92d ⟨debug matching 1 92d⟩≡

```
  if (verbose ≡ 3) {
        setSelector(STDOUT);
        ioprint("Trying to redex match "); candidate→print(); ioprintln();
  }
```

Uses `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

92e ⟨debug matching 2 92e⟩≡

```
  if (verbose ≡ 3) { ioprint("\tand "); head→print(); ioprint(" ... "); }
```

Uses `ioprint` 164 165.

92f ⟨debug matching 3 92f⟩≡

```
  if (verbose ≡ 3) ioprint("\t[succeed]\n");
```

Uses `ioprint` 164 165.

93a ⟨debug matching 4 93a⟩≡

```
  if (verbose ≡ 3) ioprint("\t[failed]\n");
```

Uses `ioprint` 164 165.

Comment 3.1.81. We now look at disruptive operations.

93b ⟨try disruptive 93b⟩≡

```
  ÷∗
  bool try_disruptive(vector<int> mpath, term ∗ candidate,
                      term ∗ parent, unint cid, term ∗ root,
                      int & tried) {
         int osel = getSelector(); setSelector(SILENT);
         if (candidate→isFunc2Args(iAssign)) {
                term ∗ arg1 = candidate→lc()→rc();
                // reduce arg2
                ioprint("simplifying "); candidate→rc()→print(); ioprintln();
                int tried2 = 0;
                bool reduced = true;
                while (reduced) {
                       reduced = candidate→rc()→reduce(mpath,NULL,
                                                   0, root, tried2);
                       if (candidate→rc()→tag ≡ D) break;
                };
                ioprint("result = "); candidate→rc()→print(); ioprintln();
                // update statement
                for (unint i=0; i≠statements.size(); i++) {
                    if (statements[i].anchor ≡ arg1→cname) {
                           assert(statements[i].persistent);
                           statements[i].stmt→rc()→freememory();
                           statements[i].stmt→fields[1] = candidate→rc();

                           statements[i].stmt→print(); ioprintln();
                           setSelector(osel);
                           break;
                    }
                }
                // candidate = Succeed
                candidate→fields[1] = NULL;
                term ∗ temp = new_term(D, iSucceeded);
                if (parent) { parent→fields[cid] = temp;
                              candidate→freememory(); }
                else { candidate→replace(temp); temp→freememory(); }
                return true;
         }
         return false;
  }
  ∗÷
```

Uses `getSelector` 164 165, `iAssign` 145, `iSucceeded` 145, `ioprint` 164 165, `ioprintln` 164 165, `isFunc2Args` 32f, `lc` 30e, `new_term` 40a, `rc` 30e, `reduce` 84a 85a, `replace` 42b, and `setSelector` 164 165.

3.1.3 Pattern Matching

3.1.3.1 Preprocessing of Statements

Comment 3.1.82. During pattern matching, the name of bound variables in the head of a program statement $s = t$ needs to be changed repeatedly. The corresponding variables in t must be changed accordingly to preserve the original meaning of the statement. As this is a key operation that needs to be done repeatedly very many times, an efficient algorithm is needed. The key idea here is that we can use the same variable node to represent corresponding variables in s and t. This way, when we change a variable in s during pattern matching, all corresponding variables in s and t get changed automatically. The term representations produced by the parser are trees without shared node. The following function `collectSharedVars` implements this kind of sharing. The procedure is simple. We first collect together all the shared variables in s and t separately using `shareLambdaVars`. Then we redirect shared variables in t to their corresponding variables in s using `shareHeadLambdaVars`.

Note that only bound variables are shared by this operation. The correctness of the function `labelStaticBoundVars` (see Comment 2.2.43) is thus not affected.

94a ⟨term::function definitions 32e⟩+≡

```
void term::collectSharedVars() {
        term ∗ head = fields[0]→fields[1];
        term ∗ body = fields[1];
        vector<term ∗> headlvars;
        head→shareLambdaVars(headlvars, true);
        body→shareLambdaVars(headlvars, false);
        body→shareHeadLambdaVars(headlvars);
}
```

Defines:
`collectSharedVars`, used in chunks 89b and 111.
Uses `shareHeadLambdaVars` 96a and `shareLambdaVars` 94b.

Comment 3.1.83. The input vector `lvars` is used to collect all the lambda variables in a term. We only need to do this for the head. The input parameter `use` controls this. The procedure of `shareLambdaVars` is as follows: every time we see a term of the form $\lambda x.t$, we use `shareVar` to redirect all occurrences of x in t to point to the x straight after the λ sign.

94b ⟨term::function definitions 32e⟩+≡

```
void term::shareLambdaVars(vector<term ∗> & lvars, bool use) {
        if (tag ≡ ABS) {
                if (use) lvars.push_back(fields[0]);
                fields[1]→shareVar(fields[0], this, 1);
                fields[1]→shareLambdaVars(lvars, use);
                return;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                fields[i]→shareLambdaVars(lvars, use);
}
```

Defines:
`shareLambdaVars`, used in chunks 94a and 111.
Uses `shareVar` 95a.

Comment 3.1.84. The procedure `shareVar` with input variable x is only ever called within the correct scope t of a term $\lambda x.t$. (If a subterm $\lambda x.t_2$ occurs inside t, we will skip that subterm.) This guarantees that all the variables that get redirected in the `(tag == V)` case are exactly those variables bound by the input variable `var`. The pointer `parent->fields[id]` points to the current term.

95a ⟨term::function definitions 32e⟩+≡

```
void term::shareVar(term ∗ var, term ∗ parent, unint id) {
        if (tag ≡ SV ∨ tag ≡ D ∨ tag ≡ F) return;
        if (tag ≡ ABS) { if (var→cname ≡ fields[0]→cname) return;
                        fields[1]→shareVar(var, this, 1); return; }
        if (tag ≡ V) {
                if (cname ≡ var→cname) {
                        parent→fields[id] = var→reuse();
                        var→parents.push_back(&parent→fields[id]);
                        this→freememory(); }
                return;
        }
        unint size = fieldsize;
        for (unint i=0; i≠size; i++) fields[i]→shareVar(var, this, i);
}
```

Defines:
`shareVar`, used in chunks 94b and 111.
Uses `reuse` 43d.

Comment 3.1.85. Pointers to term schema pointers that got redirected in `shareVar` are stored in `parents`. These are then used for further redirection in `shareHeadLambdaVars`. At present, this is the only place where `parents` is used. The `parents` parameter need not be initialized during term construction. It need not be copied during cloning. Its value also does not get affected during replacing.

95b ⟨term vector parts 30b⟩+≡

```
vector<term ∗∗> parents;
```

Comment 3.1.86. The procedure for `shareHeadLambdaVars` is as follows. Every time we see a term of the form $\lambda x.t$, we redirect x and all occurrences of x in t pointing to it (these are recorded in `parents`) if x is in `hlvars` and then we recurse on t.

96a ⟨term::function definitions 32e⟩+≡

```
void term::shareHeadLambdaVars(vector<term *> & hlvars) {
        if (hlvars.empty()) return;
        if (tag ≡ ABS) {
                int size = hlvars.size();
                for (int i=0; i≠size; i++) {
                        if (fields[0]→cname ≠ hlvars[i]→cname) continue;
                        int psize = fields[0]→parents.size();
                        for (int j=0; j≠psize; j++) {
                                ∗(fields[0]→parents[j]) = hlvars[i]→reuse();
                                fields[0]→freememory();
                        }
                        fields[0]→freememory();
                        fields[0] = hlvars[i]→reuse();
                        break;
                }
                fields[1]→shareHeadLambdaVars(hlvars);
                return;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                fields[i]→shareHeadLambdaVars(hlvars);
}
```

Defines:
`shareHeadLambdaVars`, used in chunks 94a and 111.
Uses `reuse` 43d.

Comment 3.1.87. Free variables in program statements may also need to be changed during pattern matching. To do away with the need for tree traversal, we employ the same trick to share corresponding free variables in the head and body of statements. This is done in a preprocessing step. The following function performs this task. It works as follows. Every time we see a free variable x in the head, we traverse the body to redirect all free occurrences of x to the one in the head. Redirection is accomplished using `shareFreeVar`. We assume that `labelStaticBoundVars` has been called to label the variables.

96b ⟨term::function definitions 32e⟩+≡

```
void term::collectFreeVars(term ∗ bodyparent, unint id) {
        if (tag ≡ V ∧ isFree())
                bodyparent→fields[id]→shareFreeVar(this, bodyparent, id);
        if (tag ≡ SV ∨ tag ≡ D ∨ tag ≡ F) return;
        if (tag ≡ ABS) fields[1]→collectFreeVars(bodyparent, id);
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                fields[i]→collectFreeVars(bodyparent, id);
}
```

Defines:
`collectFreeVars`, used in chunk 111.
Uses `isFree` 46e and `shareFreeVar` 97a.

Comment 3.1.88. The return value of `shareFreeVar` can be used to implement the idea described in Comment 3.1.90.

97a ⟨term::function definitions 32e⟩+≡

```
bool term::shareFreeVar(term ∗ v, term ∗ parent, unint id){
        if (tag ≡ V ∧ isFree() ∧ cname ≡ v→cname) {
                freememory(); parent→fields[id] = v→reuse(); return true; }

        if (tag ≡ SV ∨ tag ≡ D ∨ tag ≡ F) return false;
        if (tag ≡ ABS) return fields[1]→shareFreeVar(v, this, 1);
        bool ret = false;
        int size = fieldsize;
        for (int i=0; i≠size; i++)
                if (fields[i]→shareFreeVar(v, this, i)) ret = true;
        return ret;
}
```

Defines:
`shareFreeVar`, used in chunks 96b and 111.
Uses `isFree` 46e and `reuse` 43d.

Comment 3.1.89. The following function pre-computes all the free variables inside a subterm and put them in the vector `preFVars`. Pointers to terms instead of strings are used to allow us to rename free variables directly without doing another traversal.

97b ⟨term parts 29b⟩+≡

```
vector<term ∗> preFVars;
```

97c ⟨term::function definitions 32e⟩+≡

```
bool term::precomputeFreeVars() {
        if (tag ≡ SV ∨ tag ≡ D ∨ tag ≡ F) return false;
        if (tag ≡ V ∧ isFree()) {
           preFVars.push_back(this);
                return true;
        }
        if (tag ≡ ABS) {
               bool res = fields[1]→precomputeFreeVars();
               if (res)
                   preFVars = fields[1]→preFVars;
               return res;
        }
        int size = fieldsize;
        for (int i=0; i≠size; i++) {
                bool res = fields[i]→precomputeFreeVars();
                if (¬res) continue;
                int size2 = fields[i]→preFVars.size();
                for (int j=0; j≠size2; j++)
                    preFVars.push_back(fields[i]→preFVars[j]);
        }
        return ¬preFVars.empty();
}
```

Defines:
`precomputeFreeVars`, used in chunk 111.
Uses `isFree` 46e.

Comment 3.1.90. UNIMPLEMENTED IDEA: A variable that occurs in the head but not in the body of a statement can be flagged so that we do not have to put its substitution in θ.

3.1.3.2 Redex Determination

Definition 3.1.91. A *redex* of a term t is an occurrence of a subterm of t that is α-equivalent to an instance of the head of a statement.

Fact 3.1.92. *Two α-equivalent terms can only differ in the names of their bound variables. (See also [Llo03, pp. 71].)*

Algorithm 3.1.93. To determine whether a term t is a redex with respect to the head h of a statement, we need to determine whether there exists a term substitution θ such that $h\theta$ is α-equivalent to t. There is a simple algorithm for doing that:

```
$\theta \leftarrow \{\}$
while ($h\theta \neq t$) do
    $o \leftarrow$ leftmost innermost occurrence in $t$ such that $o$ is also in $h$ and $h\theta_{|o} \neq t_{|o}$;
    if $h\theta_{|o}$ and $t_{|o}$ are both $\lambda$-terms then
        change name of bound variable in $h\theta_{|o}$ to that in $t_{|o}$, renaming free
        variables in $h\theta_{|o}$ to avoid free-variable capture whenever necessary;
    else if $h\theta_{|o}$ is a free occurrence of a variable $x$ in $h$ and no free variable in $t_{|o}$
    would be captured by the substitution $\{x/t_{|o}\}$ then
        $\theta \leftarrow \theta \cup \{x/t_{|o}\}$;
    else return failure;
return $\theta$;
```

Comment 3.1.94. The no-free-variable-capture condition in the else if case is needed to prevent matching on statements like $h = \lambda y.x$ and $t = \lambda y.(g\,y)$. Without the condition, we would bind $(g\,y)$ to x, but the end result of doing $h\{x/(g\,y)\}$ is actually $\lambda z.(g\,y)$, which is not equal to t. (See Definition 2.5.3 in [Llo03].) If this kind of matching is desired, a syntactical variable must be used.

Comment 3.1.95. Algorithm 3.1.93 does not take syntatical variables into account. Conceptually, given an equation with syntatical variables in it, we should first initialise the syntactical variables to obtain a valid statement. This will then allow us to use Algorithm 3.1.93 to do pattern matching on it. In practice, we do the instantiation of syntactical variables and pattern matching at the same time. The following modified algorithm is used.

Algorithm 3.1.96. Given terms h with syntactical variables in it and a candidate redex t, the algorithm decides whether there exists θ such that $h\theta$ is α-equivalent to t.

```
$\theta \leftarrow \{\}$
while ($h\theta \neq t$) do
    $o \leftarrow$ leftmost innermost occurrence in $t$ such that $o$ is also in $h$ and $h\theta_{|o} \neq t_{|o}$;
    if $h\theta_{|o}$ and $t_{|o}$ are both $\lambda$-terms then
        change name of bound variable in $h\theta_{|o}$ to that in $t_{|o}$, renaming free
        variables in $h\theta_{|o}$ to avoid free-variable capture whenever necessary;
    else if $h\theta_{|o}$ is a free occurrence of a variable $x$ in $h$ and no free variable in $t_{|o}$
    would be captured by the substitution $\{x/t_{|o}\}$ then
        $\theta \leftarrow \theta \cup \{x/t_{|o}\}$;
    else if $h\theta_{|o}$ is a syntactical variable $\mathbf{x}$ in $h$
        $\theta \leftarrow \theta \cup \{\mathbf{x}/t_{|o}\}$;
    else return failure;
return $\theta$;
```

Provided syntactical variables only ever occur at places where a (normal) variable can appear, I think the algorithm is complete in the sense that if there is a way to instantiate the syntactical variables so that a matching can occur, Algorithm 3.1.96 will find it.

Comment 3.1.97. Algorithm 3.1.96 renames variables as necessary when both $h\theta_{|o}$ and $t_{|o}$ are λ-terms. Renaming of free variables in h is safe only because the head of a statement cannot contain more than one occurrence of a free variable.

Comment 3.1.98. Typically, the h considered in Algorithm 3.1.96 is the head of a statement $h = b$. When we rename free variables in h, we also need to rename the corresponding variables in b so as not to change the original meaning of the statement. How about the bound variables? When we change a bound variable in h, do we need to rename its corresponding variables in b?

In the presence of syntactical variables, the answer is a definite yes. Consider the statement $(f\ \lambda x.\mathbf{u}) = \lambda x.\mathbf{u}$. Given candidate redex $(f\ \lambda y.(g\ y))$, we will get the incorrect answer $\lambda x.(g\ y)$ if we do not rename the x in the body of the statement during pattern matching. Efficient algorithms for doing such renaming of variables are described in Comments 3.1.82 and 3.1.87.

Comment 3.1.99. There is a simple way to realise Algorithm 3.1.96. Start with two pointers p_t and p_h pointing, respectively, at t and h. Denote by $[p_t]$ and $[p_h]$ the subterms of t and h pointed to by p_t and p_h. Move the pointers forward one step at a time to the next subterm in the left-to-right, outermost-to-innermost order. At each time step, if $[p_h] \neq [p_t]$ then:

1. if $[p_h]$ is a syntactical variable, add $\{[p_h]/[p_t]\}$ to θ;
2. else if $[p_h]$ is a variable free in h and the free variable capture condition does not occur, add $\{[p_h]/[p_t]\}$ to θ;
3. else if $[p_t]$ and $[p_h]$ are both lambda terms and x_t and x_h are the corresponding lambda variables, then set all occurrences of x_h in $[p_h]$ to x_t, renaming as necessary free variables that get captured as a result;
4. else return failure.

99a ⟨pattern-match::function declarations 99a⟩≡

```
bool redex_match(term * head, term * body, vector<substitution> & theta);
bool redex_match(term * head, term * body, vector<substitution> & theta,
                 vector<term *> bindingAbss, term * orig_head);
```

Defines:
`redex_match`, used in chunks 57a, 87, 88a, and 102–104.
Uses `substitution` 51.

99b ⟨pattern-match::functions 99b⟩≡

```
bool redex_match(term * head, term * body, vector<substitution> & theta) {
        vector<term *> bindingAbss;
        return redex_match(head, body, theta, bindingAbss, head);
}
```

Defines:
`redex_match`, used in chunks 57a, 87, 88a, and 102–104.
Uses `substitution` 51.

100a

```
⟨pattern-match::functions 99b⟩+≡
  bool redex_match(term ∗ head, term ∗ body, vector<substitution> & theta,
                   vector<term ∗> bindingAbss, term ∗ orig_head) {
      kind head_tag = head→tag;
      kind term_tag = body→tag;

      if (head_tag ≡ SV) { ⟨redex-match::case of SV 100b⟩  }
      if (head_tag ≡ V) { ⟨redex-match::case of V 101c⟩ }
      if (head_tag ≠ term_tag) return false;

      ⟨redex-match::case of constant 102b⟩
      if (head_tag ≡ APP) { ⟨redex-match::case of APP 102c⟩ }
      if (head_tag ≡ PROD) { ⟨redex-match::case of PROD 103a⟩ }
      if (head_tag ≡ ABS) { ⟨redex-match::case of ABS 103b⟩ }
      if (head_tag ≡ MODAL) { ⟨redex-match::case of MODAL 104⟩ }
      assert(false); return false;
  }
```

Defines:
`redex_match`, used in chunks 57a, 87, 88a, and 102–104.
Uses `substitution` 51.

Comment 3.1.100. Here we consider matching on syntactical variables. A syntactical variable matches anything, if all the constraints are obeyed, that is.

100b

```
⟨redex-match::case of SV 100b⟩≡
  ⟨redex-match::case of SV::check constraints 101a⟩
  substitution sub(head→cname, body);
  theta.push_back(sub);
  return true;
```

Uses `substitution` 51.

Comment 3.1.101. The constraint `/VAR/` means that the term bound to the current syntactical variable must be a variable. The constraint `/CONST/` means that the term bound to the current syntactical variable must be a data constructor or a function symbol. The constraint `/EQUAL,x_SV/`, where `x_SV` is another syntactical variable appearing before the current one, means that the term bound to the current syntactical variable must be equal to the term bound to `x_SV`.

101a ⟨redex-match::case of SV::check constraints 101a⟩≡

```
condition ∗ constraint = head→cond;
if (constraint) {
        int ctag = constraint→tag;

        if (ctag ≡ CVAR ∧ term_tag ≠ V) return false; // problematic?
        else if (ctag ≡ CCONST ∧ term_tag ≠ D ∧ term_tag ≠ F) return false;
        else if (ctag ≡ CEQUAL) {
                // if (term_tag != D && term_tag != V) return false;
                term ∗ bound = findBinding(constraint→svname, theta);
                ⟨error handling::get previously bound 101b⟩
                if (body→equal(bound) ≡ false) return false;
        } else if (ctag ≡ CNOTEQUAL) {
                // if (term_tag != D) return false;
                term ∗ bound = findBinding(constraint→svname, theta);
                ⟨error handling::get previously bound 101b⟩
                if (body→equal(bound) ≡ true) return false;
        }
        // assert(ctag != CVAR); assert(ctag != CNOTEQUAL);
}
```

Uses `equal` 34b and `findBinding` 110b 110c.

101b ⟨error handling::get previously bound 101b⟩≡

```
if (bound ≡ NULL) {
        setSelector(STDERR);
        ioprint("The constraint EQUAL or NOTEQUAL on syntactical "
             "variables is used incorrectly; it appears before "
             "its argument is instantiated.\n");
        assert(false);
}
```

Uses `ioprint` 164 165 and `setSelector` 164 165.

Comment 3.1.102. We next examine the case of variables. We do not have to do anything if `head` is identical to `body`. If `head` is a bound variable, then `body` must be identical to `head` for matching to succeed.

101c ⟨redex-match::case of V 101c⟩≡

```
if (term_tag ≡ V ∧ head→cname ≡ body→cname) return true;
if (head→isFree() ≡ false) return false;
if (head→cname ≡ iWildcard) return true;

⟨redex-match::case of V::check free variable capture condition 102a⟩

substitution sub(head→cname, body);
theta.push_back(sub);
return true;
```

Uses `iWildcard` 145, `isFree` 46e, and `substitution` 51.

Comment 3.1.103. We need to check that no variable in `body` would be captured by the substitution `head/body`.

102a ⟨redex-match::case of V::check free variable capture condition 102a⟩≡

```
int captd;
if (body→captured(bindingAbss, captd)) {
        setSelector(STDERR);
        cerr ≪" ** Matching Failed: Free variable capture in redex-match.\n";
        ioprint("head = "); head→print(); ioprintln();
        ioprint("term = "); body→print(); ioprintln();
        assert(orig_head);
        ioprint("orig head = "); orig_head→print(); ioprintln();
        return false; }
```

Uses `captured` 48b, `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

Comment 3.1.104. We now look at the case when `head` is a constant.

102b ⟨redex-match::case of constant 102b⟩≡

```
if (head_tag ≡ F) return (head→cname ≡ body→cname);
if (head_tag ≡ D) {
        if (head→isfloat ≠ body→isfloat) return false;
        if (head→isint ≠ body→isint) return false;
        if (head→isfloat ∧ body→isfloat) return (head→numf ≡ body→numf);
        else if (head→isint ∧ body→isint) return (head→numi ≡ body→numi);
        return (head→cname ≡ body→cname);
}
```

Comment 3.1.105. The case of applications is particularly simple. We first try to match the left child. If successful, we match the right child.

102c ⟨redex-match::case of APP 102c⟩≡

```
if (¬redex_match(head→lc(),body→lc(), theta, bindingAbss, orig_head))
        return false;
return redex_match(head→rc(), body→rc(),theta,bindingAbss,orig_head);
```

Uses `lc` 30e, `rc` 30e, and `redex_match` 99a 99b 100a.

Comment 3.1.106. These can be used to debug `redex_match`.

102d ⟨redex-match::case of APP::debug matching 1 102d⟩≡

```
if (verbose ≡ 3) {
    ioprint("\n\t\tmatching "); head→lc()→print();
    ioprint(" and "); body→lc()→print(); ioprint(" ... ");
}
```

Uses `ioprint` 164 165 and `lc` 30e.

102e ⟨redex-match::case of APP::debug matching 2 102e⟩≡

```
if (verbose ≡ 3) {
    ioprint(" successful\n");
    ioprint("\t\tmatching "); head→rc()→print(); ioprint(" and ");
    body→rc()→print(); ioprint(" ... ");
}
```

Uses `ioprint` 164 165 and `rc` 30e.

Comment 3.1.107. We now look at the case of products. We cannot assume that the dimensions of `head` and `body` are equal even when the type-checker says they have the same types. Why? Well, sometimes we use a function name to represent data.

103a ⟨redex-match::case of PROD 103a⟩≡

```
unint size = head→fieldsize;
if (size ≠ body→fieldsize) return false;

for (unint i=0; i≠size; i++)
        if (¬redex_match(head→fields[i],body→fields[i],theta,
                         bindingAbss,orig_head))
                return false;
return true;
```

Uses `redex_match` 99a 99b 100a.

Comment 3.1.108. The last case is that of abstraction. We change the name of lambda variables to avoid having to worry about α-equivalence later on.

103b ⟨redex-match::case of ABS 103b⟩≡

```
if (head→fields[0]→tag ≡ SV) {
      redex_match(head→fields[0],body→fields[0],theta,bindingAbss,orig_head);
 } else { ⟨redex-match::case of ABS::change variable name 103c⟩ }
bindingAbss.push_back(head);
return redex_match(head→fields[1],body→fields[1],theta,bindingAbss,orig_head);
```

Uses `redex_match` 99a 99b 100a.

Comment 3.1.109. If necessary, we need to change the name of the bound variable in `head` so that it is the same as the bound variable in `body`. In so doing, we may inadvertently capture a free variable inside `head`. (This is an extremely rare scenario. I have never seen it happen in any non-simulated computation.) Another variable renaming is necessary in this case.

Thanks to the preprocessing we did (see Comments 3.1.82 and 3.1.87), we need to set only the name of one variable in each case.

103c ⟨redex-match::case of ABS::change variable name 103c⟩≡

```
int term_var = body→fields[0]→cname;
if (head→fields[0]→cname ≠ term_var) {
        int size = head→preFVars.size();
        for (int i=0; i≠size; i++)
                if (term_var ≡ head→preFVars[i]→cname) {
                        ⟨redex-match::write a small warning message 103d⟩
                        head→preFVars[i]→cname = newPVar();
                }
        head→fields[0]→cname = term_var;
}
```

Uses `newPVar` 148c.

103d ⟨redex-match::write a small warning message 103d⟩≡

```
int osel = getSelector(); setSelector(STDOUT);
ioprint(" ** Trouble. Variable "); head→preFVars[i]→print();
ioprint(" captured after lambda variable renaming.\n");
ioprint("head = "); head→print(); ioprintln();
ioprint("term = "); body→print(); ioprintln();
setSelector(osel);
```

Uses `captured` 48b, `getSelector` 164 165, `ioprint` 164 165, `ioprintln` 164 165, and `setSelector` 164 165.

104 ⟨redex-match::case of MODAL 104⟩≡

```
  if (head→modality ≠ body→modality) return false;
  return redex_match(head→fields[0],body→fields[0],theta,bindingAbss,orig_head);
```

Uses `redex_match` 99a 99b 100a.

Comment 3.1.110. We now look at some instructive test cases for the procedure. Evaluating the following program

```
(f \y.x) = True
: (f \y.(g y y))
```

will result in

```
  ** Matching Error: Free variable capture in redex-match.
Final Answer: (f \y.((g y) y)).
```

To force a matching here, we can use the statement `(f \y.x_SV) = True` instead. Evaluating the same query will then result in `True`.

Evaluating the program

```
(f \x.(g x y)) = (g y y)
: (f \y.(g y y))
```

will result in

```
 ** Trouble. Variable y captured after lambda variable renaming.
 ** Matching Failed: Free variable capture in redex-match.
Final Answer: (f \y.((g y) y)).
```

The lambda variable x in the head of the statement is successfully renamed at first. Matching fails when we subsequently try to match the free variable y in the head of the statement with the bound variable y in the query. The reader should convince herself that matching should indeed fail in this case.

Evaluating this next prggram

```
(f \x.(g y x)) = (g y y)
: (f \y.(g z y))
```

will produce the answer `((g z) z)`.

3.2 Interaction with the Theorem Prover

3.2.1 Rank k Computations

105 ⟨term::function definitions::unused 105⟩≡

```
  #ifndef ESCHER
  #include "tableaux.h"
  bool term::simplifyWithTP() {
        return false; // disable this function to begin with
        if (isVar() ∨ isD() ∨ isF()) return false;
        if (isAbs()) return fields[1]→simplifyWithTP();
        if (isProd()) {
              for (unint i=0; i≠fieldsize; i++)
                    if (fields[i]→simplifyWithTP()) return true;
              return false;
        }
        /∗ if done previously, return ∗/
        if (isApp() ∧ lc()→isApp() ∧ lc()→lc()→isD(iTpTag))
              return false;
        /∗ we don't do terms with free variables inside ∗/
```

```
    getFreeVars();
    if (frvarsize > 0) {
        assert(isApp() ∨ isModal());
        if (isApp()) {
            if (lc()→simplifyWithTP()) return true;
            return rc()→simplifyWithTP();
        } else if (isModal()) { assert(false); }
    }
    /∗ check that the type is Bool ∗/
    pair<type ∗, vector<term_type> > res = mywellTyped(this);
    if (res.first→getTag() ≠ "Bool") {
        assert(isApp() ∨ isModal());
        if (isApp()) {
            if (lc()→simplifyWithTP()) return true;
            return rc()→simplifyWithTP();
        } else if (isModal()) { assert(false); }
    }
    if (¬(isApp() ∧ lc()→isF(iTpHelp))) {
        assert(isApp() ∨ isModal());
        if (isApp()) {
            if (lc()→simplifyWithTP()) return true;
            return rc()→simplifyWithTP();
        } else if (isModal()) { assert(false); }
    }
    ⟨simplifyWithTP::call theorem prover 107a⟩
}

bool term::simplifyWithTP2() {
    ⟨simplifyWithTP::call theorem prover 107a⟩
}
#endif
```

Defines:
`simplifyWithTP`, used in chunk 111.
`simplifyWithTP2`, used in chunk 111.

Uses `getFreeVars` 45b, `iTpHelp` 145, `iTpTag` 145, `isAbs` 30a, `isApp` 30a, `isD` 30a, `isF` 30a, `isModal` 30a, `isProd` 30a, `isVar` 30a, `lc` 30e, `mywellTyped` 28b, `rc` 30e, and `term_type` 17b.

107a ⟨simplifyWithTP::call theorem prover 107a⟩≡

```
term ∗ theorem = rc();
vector<pair<term ∗,bool> > tlist;
// insert goal formula into tableau
term ∗ goal = new_term(APP);
// goal->lc = new_term(F, iNot); goal->rc = theorem->clone();
goal→insert(new_term(F, iNot)); goal→insert(theorem→clone());
pair<term ∗,bool> gent(goal→normalise(), false);
tlist.push_back(gent);

// prove theorem
backchain = true;
Tableaux tab(tlist);
TruthValue ret = tab.expand();
if (verbose) {
        ioprint("Attempted Proof:\n "); tab.print();
        ioprint("  Answer : "); ret.print(); ioprintln(" ;");
}
tab.freememory();
if (ret.value ≡ MYTRUE) {
        lc()→freememory(); rc()→freememory(); fieldsize = 0;
        tag = D; cname = iTrue;
        return true;
} else {
        term ∗ temp = new_term(APP);
        ÷∗ temp→lc = new_term(APP);
        temp→rc = theorem→clone();
        temp→lc→lc = new_term(D, iTpTag);
        temp→lc→rc = new_term(D, iDontKnow); ∗÷
        temp→insert(new_term(APP));
        temp→insert(theorem→clone());
        temp→lc()→insert(new_term(D, iTpTag));
        temp→lc()→insert(new_term(D, iDontKnow));
        replace(temp);
        temp→freememory();
}
return false;
```

Uses `iDontKnow` 145, `iNot` 145, `iTpTag` 145, `iTrue` 145, `insert` 30e, `ioprint` 164 165, `ioprintln` 164 165, `lc` 30e, `new_term` 40a, `normalise` 57b, `rc` 30e, and `replace` 42b.

3.2.2 Free Variable Instantiation

Comment 3.2.1. In the theorem proving mode, we sometimes need to replace universally quantified variables with so-called free variables. These needs to be instantiated at some stage. One important mechanism used to instantiate them is to find matching between subformulas that would allow us to close off branches. The matching algorithm is very similar to `redex_match`, except that we concentrate only on free variables and do not do α-conversion.

107b ⟨pattern-match::function declarations 99a⟩+≡

```
bool freevar_match(term ∗ fml1, term ∗ fml2, vector<substitution> & theta);
bool freevar_match(term ∗ fml1, term ∗ fml2, vector<substitution> & theta,
                vector<term ∗> bindingAbss);
```

Uses `freevar_match` 108a and `substitution` 51.

108a ⟨pattern-match::functions 99b⟩+≡

```
bool freevar_match(term ∗ fml1, term ∗ fml2,
              vector<substitution> & theta) {
      vector<term ∗> bindingAbss;
      return freevar_match(fml1, fml2, theta, bindingAbss);
}
```

Defines:
  `freevar_match`, used in chunks 107–109.
Uses `substitution` 51.

108b ⟨pattern-match::functions 99b⟩+≡

```
bool freevar_match(term ∗ head, term ∗ body, vector<substitution> & theta,
              vector<term ∗> bindingAbss) {
      kind head_tag = head→tag;
      kind term_tag = body→tag;
      assert(head_tag ≠ SV /∗ && head_tag != MODAL ∗/);
      if (head_tag ≡ V) {
            if (isUVar(head→cname) ∧ ¬body→occursFree(head→cname))
                  { term ∗ orig_head = NULL;
                    head→validfree = true; head→free = true;
                    ⟨redex-match::case of V 101c⟩ }
      }
      if (term_tag ≡ V) {
            if (isUVar(body→cname) ∧ ¬head→occursFree(body→cname))
                  { term ∗ headtemp = head;
                    head = body;
                    body = headtemp;
                    term ∗ orig_head = NULL;
                    head→validfree = true; head→free = true;
                    ⟨redex-match::case of V 101c⟩
                  }
      }
      if (head_tag ≠ term_tag) return false;
      if (head_tag ≡ V ∧ term_tag ≡ V) {
            if (head→cname ≠ body→cname) return false;
            return true;
      }

      ⟨redex-match::case of constant 102b⟩
      if (head_tag ≡ APP) { ⟨freevar-match::case of APP 109b⟩ }
      if (head_tag ≡ PROD) { ⟨freevar-match::case of PROD 109d⟩ }
      if (head_tag ≡ ABS) { ⟨freevar-match::case of ABS 109a⟩ }
      if (head_tag ≡ MODAL) { ⟨freevar-match::case of MODAL 109c⟩ }
      assert(false); return false;
}
```

Uses `freevar_match` 108a, `isUVar` 55a, `occursFree` 47b, and `substitution` 51.

Comment 3.2.2. One significant case where `redex_match` and `freevar_match` differs is in the case of abstractions. In `redex_match`, we will rename the name of bound variables in the `head` if necessary. (We are looking for a substitution that achieves α-equivalence). This is not done in `freevar_match`. We may or may not want to do this in the future.

109a ⟨freevar-match::case of ABS 109a⟩≡

```
if (head→fields[0]→cname ≠ body→fields[0]→cname) return false;
bindingAbss.push_ back(head);
return freevar_ match(head→fields[1], body→fields[1], theta, bindingAbss);
```

Uses `freevar_match` 108a.

Comment 3.2.3. These cases are identical for `redex_match` and `freevar_match`.

109b ⟨freevar-match::case of APP 109b⟩≡

```
if (¬freevar_ match(head→lc(),body→lc(),theta,bindingAbss)) return false;
return freevar_ match(head→rc(), body→rc(), theta, bindingAbss);
```

Uses `freevar_match` 108a, `lc` 30e, and `rc` 30e.

109c ⟨freevar-match::case of MODAL 109c⟩≡

```
if (head→modality ≠ body→modality) return false;
return freevar_ match(head→fields[0], body→fields[0],theta, bindingAbss);
```

Uses `freevar_match` 108a.

109d ⟨freevar-match::case of PROD 109d⟩≡

```
unint size = head→fieldsize;
if (size ≠ body→fieldsize) return false;

for (unint i=0; i≠size; i++) {
        setSelector(SILENT);
        ioprint("unifying "); head→fields[i]→print();
        ioprint(" and "); body→fields[i]→print(); ioprint(" ");
        if (¬freevar_ match(head→fields[i],body→fields[i],theta,bindingAbss)){
                setSelector(SILENT); ioprint(" false\n"); setSelector(SILENT);
                return false;
        }
        setSelector(SILENT); ioprint(" true\n"); setSelector(SILENT);
}
return true;
```

Uses `freevar_match` 108a, `ioprint` 164 165, and `setSelector` 164 165.

3.2.2.1 Manipulating Substitutions

109e ⟨pattern-match::function declarations 99a⟩+≡

```
void printTheta(vector<substitution> & theta);
```

Defines:
  `printTheta`, used in chunk 92a.
Uses `substitution` 51.

110a ⟨pattern-match::functions 99b⟩+≡

```
void printTheta(vector<substitution> & theta) {
        if (getSelector() ≡ SILENT) return;
        ioprint('{');
        int size = theta.size();
        if (size ≡ 0) { ioprint("}\n"); return; }
        for (int i=0; i≠size-1; i++) {
                ioprint('(');
                if (theta[i].first ≥ 5000) {
                        ioprint(pve); ioprint(theta[i].first-5000); }
                else ioprint(getString(theta[i].first));
                ioprint('/');
                theta[i].second→print(); ioprint("), ");
        }
        ioprint('(');
        if (theta[size-1].first ≥ 5000) {
                ioprint(pve); ioprint(theta[size-1].first-5000); }
        else ioprint(getString(theta[size-1].first));
        ioprint('/');
        theta[size-1].second→print(); ioprint(')');
        ioprint("}\n");
}
```

Defines:
  `printTheta`, used in chunk 92a.
Uses `getSelector` 164 165, `getString` 147, `ioprint` 164 165, and `substitution` 51.

110b ⟨pattern-match::function declarations 99a⟩+≡

```
term ∗ findBinding(int svname, vector<substitution> & theta);
```

Defines:
  `findBinding`, used in chunk 101a.
Uses `substitution` 51.

110c ⟨pattern-match::functions 99b⟩+≡

```
term ∗ findBinding(int svname, vector<substitution> & theta) {
        int size = theta.size();
        for (int i=0; i≠size; i++)
                if (theta[i].first ≡ svname) return theta[i].second;
        return NULL;
}
```

Defines:
  `findBinding`, used in chunk 101a.
Uses `substitution` 51.

3.2.2.2 File Organization

111 ⟨terms.h 111⟩≡

```
#ifndef _TERM_H
#define _TERM_H

#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <utility>
#include <cassert>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include "io.h"
using namespace std;

#define unint unsigned int  // defined in stdlib.h

struct term;
class type;

⟨term::definitions 38a⟩
⟨term::supporting types 38b⟩
typedef vector<int> occurrence;
⟨term::type defs 29a⟩

extern const string & getString(int code);
extern bool isUVar(term * t);
extern bool isUVar(int cn);

struct term {
        ⟨term bool parts 31a⟩
        ⟨term parts 29b⟩
        term * next;
        ⟨term vector parts 30b⟩
        ⟨term::function declarations 30a⟩
        term * clone();
        void freememory();
        void replace(term * t);
        bool equal(term * t);
        bool isFunc2Args();
        bool isFunc2Args(int f);
        term * spineTip();
        term * spineTip(int & x);
        bool isChar();
        bool isString();
        bool isAString();
        bool isRigid();
        void print();
        void printVertical(unint level);
        void getFreeVars();
        void unmarkValidfree();
```

```
    void labelStaticBoundVars();
    void labelBound(int x);
    bool occursFree(int var);
    bool occursFreeNaive(int var);
    bool occursFreeNaive(int var, vector<int> boundv);
    bool captured(vector<term *> & bvars, int & captd);
    void rename(int var1, int var2);
    void renameLambdaVar(int var1, int var2);
    void subst(vector<substitution> & subs);
    void subst(substitution & sub);
    void subst2(vector<substitution> & subs, vector<term *> bv,
            term ** pointer);
    // bool containsQuantifiers();
    bool isNegation();
    bool isNegationOf(term * t2);
    bool isDiamond();
    void stripNegations();
    bool containsFreeVariable();
    void collectFreeVariables(set<int> & fvars);
    term * normalise();
    term * normalise1();
    term * normalise2();
    void collectFunctionNames(set<int> & x);
    bool termReplace(term * s, term * r,
              term * parent,int id);
    bool matchReplace(term * s, term * r,
               term * parent,int id);
    bool simplifyEquality(term * parent, unint id);
    bool simplifyArithmetic(term * parent, unint id);
    bool simplifyInequalities(term * parent, unint id);
    bool simplifyMath(term * parent, unint id);
    bool betaReduction(term * parent, unint id);
    bool simplifyIte(term * parent, unint id);
    bool simplifyConjunction();
    bool simplifyConjunction2(term * parent, unint id);
    bool simplifyExistential(term * parent, unint id);
    bool simplifyUniversal(term * parent, unint id);
    bool simplifyModalTerms(term * parent, unint id);
    term * findEq(term * root);
    bool isEq(term * root);
    bool isEq(int var);
    term * replaceEq(int var);
    void collectSharedVars();
    void shareLambdaVars(vector<term *> & lvars, bool use);
    void shareVar(term * var, term * parent, unint id);
    void shareHeadLambdaVars(vector<term *> & hlvars);
    void collectFreeVars(term * bodyparent, unint id);
    void collectLambdaVars(multiset<int> & ret);
    bool shareFreeVar(term * v, term * parent, unint id);
    bool precomputeFreeVars();
    // bool simplifyWithTP();
    // bool simplifyWithTP2();
};
```

```
⟨term::memory management 39a⟩
extern int newPVar();
extern int newUVar();
extern term ∗ newT2Args(kind k, const string & f);
extern term ∗ newT2Args(kind k, int f);

#endif
```

Uses `betaReduction` 69a, `captured` 48b, `collectFreeVariables` 55a, `collectFreeVars` 96b, `collectSharedVars` 94a, `containsFreeVariable` 55a, `equal` 34b, `findEq` 71b, `getFreeVars` 45b, `getString` 147, `isAString` 33a, `isChar` 33a, `isDiamond` 55c, `isEq` 72a 77b, `isFunc2Args` 32f, `isNegation` 55b, `isNegationOf` 56a, `isString` 33a, `isUVar` 55a, `labelBound` 47a, `labelStaticBoundVars` 46f, `matchReplace` 57a, `newPVar` 148c, `newT2Args` 33c, `newUVar` 148c, `normalise` 57b, `normalise1` 58, `normalise2` 60a, `occursFree` 47b, `occursFreeNaive` 47c, `precomputeFreeVars` 97c, `printVertical` 37a, `rename` 49b, `renameLambdaVar` 50, `replace` 42b, `replaceEq` 77a, `shareFreeVar` 97a, `shareHeadLambdaVars` 96a, `shareLambdaVars` 94b, `shareVar` 95a, `simplifyArithmetic` 65, `simplifyConjunction` 71a, `simplifyConjunction2` 73, `simplifyEquality` 61, `simplifyExistential` 75a, `simplifyInequalities` 67, `simplifyIte` 69b, `simplifyMath` 68, `simplifyModalTerms` 80, `simplifyUniversal` 78a, `simplifyWithTP` 105, `simplifyWithTP2` 105, `spineTip` 32e, `stripNegations` 56b, `subst` 52a, `subst2` 52b, `substitution` 51, and `termReplace` 56c.

113a ⟨terms.cc 113a⟩≡

```
#include "terms.h"
⟨terms.cc::local functions 33c⟩
⟨term::function definitions 32e⟩
```

113b ⟨pattern-match.h 113b⟩≡

```
#ifndef _PATTERN_MATCH_H
#define _PATTERN_MATCH_H

#include "terms.h"

⟨pattern-match::function declarations 99a⟩

#endif
```

113c ⟨pattern-match.cc 113c⟩≡

```
#include <iostream>
#include <utility>
#include <vector>
#include "io.h"
#include "pattern-match.h"
#include "global.h"

⟨pattern-match::functions 99b⟩
```

Chapter 4

Parsing

4.1 Parsing using Lex and Yacc

4.1.1 Scanner

115 ⟨escher-scan.l 115⟩≡

```
%{
#include <iostream>
#include <string.h>
#include <stack>
#include "terms.h"
#include "unification.h"
#include "y.tab.h"
using namespace std;

char linebuf[5000];
int tokenpos = 0;
int import_level = 0;
bool quiet;
bool interactive;
%}
⟨flex options 118b⟩
%x CMNT
%%
\{\-       BEGIN CMNT;
<CMNT>.|\n ;
<CMNT>\-\} BEGIN INITIAL;
[\t ]+             { ⟨lex:tpos 117c⟩ }
\-\-.*             { // cout « "-\n";
                     ⟨lex:tpos 117c⟩ }
\n.*               { ⟨lex error reporting hackery 117b⟩ tokenpos = 0;}
LastResort         { ⟨lex:tpos 117c⟩ return LASTRESORT; }
Cache              { ⟨lex:tpos 117c⟩ return CACHE; }
Eager              { ⟨lex:tpos 117c⟩ return EAGER; }
Persistent         { ⟨lex:tpos 117c⟩ return PERSISTENT; }
type               { ⟨lex:tpos 117c⟩ return TYPE; }
prove              { ⟨lex:tpos 117c⟩ return PROVE; }
KB                 { ⟨lex:tpos 117c⟩ return KB; }
Bool               { ⟨lex:tpos 117c⟩ return BOOL; }
Int                { ⟨lex:tpos 117c⟩ return INT; }
Float              { ⟨lex:tpos 117c⟩ return FLOAT; }
Char               { ⟨lex:tpos 117c⟩ return CHAR; }
String             { ⟨lex:tpos 117c⟩ return STRING; }
ListString         { ⟨lex:tpos 117c⟩ return LISTRING; }
\-\>               { ⟨lex:tpos 117c⟩ return ARROW; }
import             { ⟨lex:tpos 117c⟩ return IMPORT; }
quit               { ⟨lex:tpos 117c⟩ return QUIT; }
VAR                { ⟨lex:tpos 117c⟩ return VAR; }
CONST              { ⟨lex:tpos 117c⟩ return CONST; }
EQUAL              { ⟨lex:tpos 117c⟩ return EQUAL; }
NOTEQUAL           { ⟨lex:tpos 117c⟩ return NOTEQUAL; }
StrList            { ⟨lex:tpos 117c⟩ return STRLIST; }
add                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return ADD; }
sub                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return SUB; }
max                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MAX; }
```

```
min                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MIN; }
mul                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MUL; }
div                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return DIV; }
mod                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MOD; }
sin                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return SIN; }
cos                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return COS; }
sqrt                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return SQRT; }
exp                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return EXP; }
atan2               { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return ATAN2; }
if                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return IF; }
then                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return THEN; }
else                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return ELSE; }
ite                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return ITE; }
\&\&                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return AND; }
\|\|                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return OR; }
not                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return NOT; }
implies             { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return IMPLIES; }
iff                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return IFF; }
sigma               { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return SIGMA; }
pi                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return PI; }
exists              { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return EXISTS; }
forall              { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return FORALL; }
box                 { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return BOX; }
\|\-                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return TURNSTILE; }
assign              { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return ASSIGN; }
\=                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYEQ; }
\/\=                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYNEQ; }
\<\=                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYLTE; }
\<                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYLT; }
\>\=                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYGTE; }
\>                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return MYGT; }
True                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return TRUE; }
False               { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return FALSE; }
\#                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return CONS;}
\[\]                { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return EMPTYLIST;}
-?[0-9]+            { ⟨lex:tpos 117c⟩ yylval.numint = atoi(yytext);
                      return DATA_CONSTRUCTOR_INT; }
-?[0-9]+\.[0-9]+    { ⟨lex:tpos 117c⟩ yylval.num = atof(yytext);
                      return DATA_CONSTRUCTOR_FLOAT; }
\'[^']\'            { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩
                      return DATA_CONSTRUCTOR_CHAR; /*'*/}
\"[^"]*\"           { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩
                      return DATA_CONSTRUCTOR_STRING; /*"*/ }
[a-zA-Z\/0-9\_\.\-]+\.es  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return FILENAME; }
\_                  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return VARIABLE; }
[a-z][0-9\_]*       { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return VARIABLE; }
\:[a-z]+[0-9\_]*    { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return VARIABLE; }
pv(e|t)[0-9]*       { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return VARIABLE; }
[a-zA-Z][0-9]*\_SV  { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return SYN_VARIABLE; }
[a-z][a-zA-Z0-9\-\_\']* { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return IDENTIFIER1; }
[A-Z][a-zA-Z0-9\-\_\']* { ⟨lex:tpos 117c⟩ ⟨lex:copy yytext 117a⟩; return IDENTIFIER2; }
.                       { ⟨lex:tpos 117c⟩ return yytext[0]; }

%%
```

```
#define YY_NO_UNPUT 1 // suppose to get rid of warning message
⟨facilities for handling multiple input files 118a⟩
```

117a ⟨lex:copy yytext 117a⟩≡

```
yylval.name = new char[strlen(yytext)+1];
strcpy(yylval.name, yytext);
```

Comment 4.1.1. I learned this trick for achieving better error recovery from [LMB92, p. 246]. The regular expression
`n.*` matches a newline and the next line, which is saved in `linebuf` before being returned to the scanner by `yyless`. The variable `tokenpos` remembers the current position on the current line.

117b ⟨lex error reporting hackery 117b⟩≡

```
if (!quiet) cerr « "prompt> ";
assert(strlen(yytext+1) <= 5000);
strcpy(linebuf, yytext+1); yyless(1);
```

117c ⟨lex:tpos 117c⟩≡

```
tokenpos += yyleng;
```

117d ⟨yacc token definitions 117d⟩≡

```
%token IMPORT QUIT ARROW PROVE KB TYPE
%token EAGER CACHE LASTRESORT PERSISTENT
%token DATA VAR CONST EQUAL NOTEQUAL BOX EXISTS FORALL TURNSTILE
%token AND OR NOT IMPLIES IFF ADD SUB MAX MIN MUL DIV MOD SIN COS SQRT EXP ATAN2
%token IF THEN ELSE ITE
%token SIGMA PI MYLT MYLTE MYGT MYGTE MYEQ MYNEQ ASSIGN
%token TRUE FALSE CONS EMPTYLIST
%token BOOL INT FLOAT CHAR STRING LISTRING STRLIST
%token <name> FILENAME
%token <name> VARIABLE
%token <name> DATA_CONSTRUCTOR
%token <numint> DATA_CONSTRUCTOR_INT
%token <num> DATA_CONSTRUCTOR_FLOAT
%token <name> DATA_CONSTRUCTOR_STRING
%token <name> DATA_CONSTRUCTOR_CHAR
%token <name> SYN_VARIABLE
%token <name> IDENTIFIER1
%token <name> IDENTIFIER2
```

Comment 4.1.2. Escher allows nested import statements in program files. Unfortunately, we cannot simply switch input files every time we see an import statement to read from the correct file because flex scanners do a lot of buffering. That is to say, the next token comes from the buffer, not the file `yyin`.

The solution provided by flex is a mechanism to create and switch between input buffers, and this is what we used here. A stack of input buffers is used to handle multiply nested import statements. Every time we see an import statement, we call `switchBuffer` to push the current buffer onto stack, and then create a new buffer and switch to it. When we are done with the current buffer, the scanner will call `yywrap` to delete the existing buffer and then revert to the previous buffer stored on top of the stack.

See, for more details on flex, [Pax95].

118a ⟨facilities for handling multiple input files 118a⟩≡

```
stack<YY_BUFFER_STATE> import_stack;

void switchBuffer(FILE * in) {
        YY_BUFFER_STATE current = YY_CURRENT_BUFFER;
        import_stack.push(current);
        // cout « "Switching to new file.\n";
        YY_BUFFER_STATE newf = yy_create_buffer(in, YY_BUF_SIZE);
        yy_switch_to_buffer(newf);
        import_level++;
}
int yywrap() {
        import_level-;
        cerr « "done "; if (import_level == 0) cerr « endl;
        if (import_level == 0 && interactive) quiet = false;
        if (!quiet) cerr « "prompt> ";
        YY_BUFFER_STATE current = YY_CURRENT_BUFFER;
        yy_delete_buffer(current);
        if (import_stack.size()) {
                yy_switch_to_buffer(import_stack.top());
                import_stack.pop();
                return 0;
        }
        return 1;
}
int mywrap() {
        if (interactive) quiet = false;
        if (!quiet) cerr « "prompt> ";
        // YY_BUFFER_STATE current = YY_CURRENT_BUFFER;
        // yy_delete_buffer(current);
        if (import_stack.size()) {
                yy_switch_to_buffer(import_stack.top());
                import_stack.pop();
                return 0;
        }
        return 1;
}
```

Defines:
`mywrap`, used in chunk 119.
`switchBuffer`, used in chunk 120.

Comment 4.1.3. It forces input to be read one character at a time. The option `yylineno` allows us to track down the line number of an offending command.

118b ⟨flex options 118b⟩≡

```
%option yylineno
```

4.1.2 Parser

119 ⟨escher-parser.y 119⟩≡

```
%{
#include <iostream>
#include <vector>
#include <stack>
#include "global.h"
#include "io.h"
#include "terms.h"
#include "unification.h"
#include "pattern-match.h"
using namespace std;

#define YYMAXDEPTH 50000

extern int yylex(); extern FILE * yyin;
extern bool quiet; extern bool interactive;
int mywrap();
int yyparse();
⟨parser::function declarations 120d⟩
⟨parser::variables 120c⟩
%}

%union { char * name; int cname;
         float num;
         int numint;
         term * trm;
         type * c_type;
         condition * cond;   }

⟨yacc token definitions 117d⟩
%type <trm> term_schema
%type <cond> sv_condition
%type <c_type> type
%type <name> dataconstructor
%type <cname> functionsymbol
%type <numint> stmt_ctrl_directive
%%

input : program_statements ;
program_statements : /* empty */ | program_statements program_statement ;
program_statement : import | type_info | statement_schema | query | quit ;

⟨parser::quit 120a⟩
⟨parser::import 120b⟩
⟨parser::query 120e⟩
⟨parser::type info 132⟩
⟨parser::statement schema 122a⟩
⟨parser::term schema 125⟩

%%
⟨parser::error reporting 137b⟩
⟨escher main program 136a⟩
```

Uses `mywrap` 118a.

Comment 4.1.4. Quiting is easy. We clean up the memory occupied by the program and then exit.

120a ⟨parser::quit 120a⟩≡

```
quit : QUIT ’;’ { cout « "Quiting...\n"; cleanup(); exit(0); } ;
```

Uses `cleanup` 137c 137d.

Comment 4.1.5. We allow nested import statements. See Comment 4.1.2 on how this works.

120b ⟨parser::import 120b⟩≡

```
import : IMPORT FILENAME ’;’
        { if (imported.find($2) == imported.end()) {
            FILE * in = fopen($2, "r");
            if (!in) {
                  cerr « "Error reading from " « $2 « endl; assert(false); }
            quiet = true;
            switchBuffer(in); cerr « "  Reading " « $2 « "...";
            imported.insert($2);
          }
          // if (!quiet) cerr « "prompt> ";
        }
       ;
```

Uses `switchBuffer` 118a.

120c ⟨parser::variables 120c⟩≡

```
#include <set>
set<string> imported;
```

120d ⟨parser::function declarations 120d⟩≡

```
extern int switchBuffer(FILE * in);
```

Uses `switchBuffer` 118a.

Comment 4.1.6. This is where a computation starts.

120e ⟨parser::query 120e⟩≡

```
query : ’:’ term_schema ’;’           { ⟨parser::perform a computation 121a⟩
                                        if (answer) answer->freememory(); }
      | ’:’ term_schema ’|’ CACHE ’;’ { term * oquery = $2->clone();
                                        ⟨parser::perform a computation 121a⟩
                                        ⟨parser::cache computed result 121b⟩ }
      ;
```

Comment 4.1.7. Given a query, we repeatedly simplify it using `reduce` until nothing can be done anymore. If the end result is a data constructor, we print it. The variable `tried` is the total number of redexes tried throughout the computation.

121a ⟨parser::perform a computation 121a⟩≡

```
  bool program_okay = typeCheck();
  type * ret = NULL;
  term * answer = NULL;
  if (program_okay && typecheck) ret = wellTyped($2);
  if (program_okay) {
          if (ret) delete_type(ret);
          ⟨parser::query::output query 121c⟩
          $2->reduceRpt();
          answer = $2; //$
          ⟨parser::query::output result 121d⟩
  } else {
          cerr « "  Error: Query not evaluated.\n";
          if (!quiet) cerr « "prompt> ";
  }
```

Comment 4.1.8. Here we form a statement from the original query and the answer obtained and insert that into the `cacheStatements` vector.

121b ⟨parser::cache computed result 121b⟩≡

```
  statementType * st = new statementType();
  term * head = oquery;
  term * body = answer;
  term * p = newT2Args(F, iEqual); p->initT2Args(head, body);
  st->stmt = p;
  ⟨parser::preprocess statements 124a⟩

  term * leftmost = head->spineTip(st->numargs);
  if (leftmost->isF()) {
          st->anchor = leftmost->cname;
          insert_ftable(leftmost->cname, st->numargs); }
  cachedStatements.push_back(st);
```

121c ⟨parser::query::output query 121c⟩≡

```
  int osel = getSelector(); setSelector(STDOUT);
  ioprint("  Query: "); $2->print(); ioprintln(); /*$*/
  setSelector(osel);
```

121d ⟨parser::query::output result 121d⟩≡

```
  // cout « "Total candidate redexes tried = " « tried « endl;
  int osel2 = getSelector(); setSelector(STDOUT);
  ioprint("Steps = "); ioprint(ltime);
  if (optimise) { ioprint(" ("); ioprint(cltime); ioprint(" cached step(s))"); }
  ioprint("\n  Answer: ");
  answer->print(); ioprintln(" ;");
  setSelector(osel2);
  // answer->freememory();
  ltime = 0; cltime = 0;
  cerr « "prompt> ";
```

Comment 4.1.9. We now move on to the parsing of Escher statements. Two different kinds of input are supported. We can accept a vanilla Escher statement with syntactic variables in it. We can also accept Bach input equations. System defined statements are used in Alkemy for other purposes.

122a ⟨parser::statement schema 122a⟩≡

```
statement_schema :
         term_schema MYEQ term_schema ';'
           { ⟨escher-parser::statement schema 122b⟩ }
       | term_schema MYEQ term_schema ';' stmt_ctrl_directive ';'
           { ⟨escher-parser::statement schema 122b⟩
             ⟨statement schema::control directives 124c⟩
           }
       ;

stmt_ctrl_directive : EAGER { $$ = EAGER ; }
                    | LASTRESORT { $$ = LASTRESORT; }
                    | CACHE { $$ = CACHE; }
                    | PERSISTENT { $$ = PERSISTENT; }
                    ;
```

Comment 4.1.10. In the case of Escher statements, we just put the head and the body together and do the necessary preprocessing.

122b ⟨escher-parser::statement schema 122b⟩≡

```
statementType * st = new statementType();
term * head = $1; term * body = $3;
⟨parser::make sure statement head has the right form 123⟩
st->stmt = newT2Args(F, iEqual); st->stmt->initT2Args(head, body);
⟨parser::preprocess statements 124a⟩
insert_statement(st);
```

Comment 4.1.11. Here, we check that

1. Every free variable appearing in the body also appears in the head. This is part of the type-weaker condition on statements.
2. Every free variable in the head appears exactly once.
3. Every lambda variable in the head is unique. This extra condition is needed to make sure the preprocessing code behaves alright.

123 ⟨parser::make sure statement head has the right form 123⟩≡

```
term * leftmost = head->spineTip(st->numargs); //$
assert(leftmost->isF());
// make sure all free variables in body appears in the head
// head->labelVariables(0); body->labelVariables(0);
head->getFreeVars(); body->getFreeVars();

for (int i=0; i!=body->frvarsize; i++) {
        if (body->frvars[i] == -5) continue;
        bool done = false;
        for (int j=0; j!=$1->frvarsize; j++)
                if (body->frvars[i] == head->frvars[j]) {
                        done = true; break; }
        if (!done) {
                setSelector(STDERR);
                cerr « " *** Error parsing statement: ";
                head->print(); cerr « " = "; body->print(); cerr « endl;
                cerr « "Variable " « getString(body->frvars[i]) «
                        " free in body but not free in head.\n";
                assert(false);
        }
 }
// make sure every free variable occurs only once in the head
for (int i=0; i!=head->frvarsize; i++) {
        if (head->frvars[i] == -5) continue;
        if (head->frvars[i] == iWildcard) continue;
        for (int j=i+1; j!=head->frvarsize; j++)
                if (head->frvars[i] == head->frvars[j]) {
                        setSelector(STDERR);
                        cerr « " *** Error parsing statement: ";
                        head->print(); cerr « " = "; body->print(); cerr«endl;
                        cerr « "Variable " « head->frvars[i] «
                                " occurs multiple times in head.\n";
                        assert(false);
                }
 }

// make sure lambda variables in the head are unique
multiset<int> lvars; head->collectLambdaVars(lvars);
multiset<int>::iterator p = lvars.begin();
while (p != lvars.end()) {
        if (lvars.count(*p) > 1) {
                setSelector(STDERR);
                cerr « " *** Error parsing statement: ";
                head->print(); cerr « " = "; body->print(); cerr«endl;
                cerr « "Lambda variable " « getString(*p) «
```

```
                        " occurs multiple times in head.\n";
                assert(false);
        }
        p++;
}
// everything is okay now
st->anchor = leftmost->cname;
insert_ftable(leftmost->cname, st->numargs);
```

Comment 4.1.12. Here we perform different kinds of preprocessing on statements talked about in §3.1.3.1 and other places.

124a ⟨parser::preprocess statements 124a⟩≡

```
head->labelStaticBoundVars(); body->labelStaticBoundVars();
st->stmt->collectSharedVars();
head->collectFreeVars(st->stmt, 1);
head->precomputeFreeVars();
```

Comment 4.1.13. We have some control directives that can be used to control the evaluation order of Bach.

We provide a mechanism to specify that certain statements should be evaluated in an eager fashion. The default Escher evaluation strategy is a lazy one: the leftmost outermost redex is picked at any step. The eager strategy stipulates that the leftmost innermost redex be picked at any step. Eager statements are processed in the same way as normal statements. We just set a tag to say the statement is eager. The `try_match_n_reduce` function will take this tag into account when doing computations.

Some statements in the booleans module increase the length of the resulting term. (This is usually the case for user-defined statements, but statements in the booleans module, in most cases, should not be like that.) We provide here a mechanism to delay the application of such statements as a last resort.

We provide a control direction to specify that computations involving certain functions should be cached to improve efficiency.

We need persistent objects in database applications. This is done by labelling certain statements as persistent. The RHS of a persistent statement can change during run-time.

124b ⟨escher-parser::statement schema cache 124b⟩≡
⟨escher-parser::statement schema 122b⟩

```
cacheFuncs.insert(leftmost->cname);
```

124c ⟨statement schema::control directives 124c⟩≡

```
if ($5 == EAGER) st->eager = true;
if ($5 == LASTRESORT) st->lastresort = true;
if ($5 == PERSISTENT) st->persistent = true;
if ($5 == CACHE) cacheFuncs.insert(leftmost->cname);
```

Comment 4.1.14. In addition to side conditions on syntactic variables, we also have side conditions on the type of subterms residing in the head of statements. This mechanism is designed to allow us to overload a function with definitions that are type dependent. For example, suppose we want to write a function $card\ :\ (a->Bool)->Nat$ to compute the cardinality of a given set. Depending on the actual type of a, we may have different definitions. For example, we may have

$$(card\ \lambda x.v) = (card\ (enumerate2D\ \lambda x.v));\ \ \text{if } \lambda x.v : (a \times b) \to \Omega$$
$$(card\ \lambda x.v) = (card\ (enumerate3D\ \lambda x.v));\ \ \text{if } \lambda x.v : (a \times b \times c) \to \Omega$$

if we have different ways of enumerating sets of tuples.

Here, we just add the type condition to the statement. This will be checked during pattern matching; see Comment 3.1.71.

Comment 4.1.15. We next look at term schemas.

125 ⟨parser::term schema 125⟩≡

```
term_schema : SYN_VARIABLE { $$ = new_term(SV, insert_symbol($1)); }
            | SYN_VARIABLE sv_condition
                           { $$=new_term(SV, insert_symbol($1)); $$->cond = $2;}
            | VARIABLE     { if ($1[0] == '_')
                                  $$ = new_term(V, iWildcard);
                             else $$ = new_term(V, insert_symbol($1)); }
            | SIGMA        { $$ = new_term(F, iSigma); }
            | PI           { $$ = new_term(F, iPi); }
            | AND          { $$ = new_term(F, iAnd); }
            | OR           { $$ = new_term(F, iOr); }
            | NOT          { $$ = new_term(F, iNot); }
            | IMPLIES      { $$ = new_term(F, iImplies); }
            | ITE          { $$ = new_term(F, iIte); }
            | IFF          { $$ = new_term(F, iIff); }
            | ADD          { $$ = new_term(F, iAdd); }
            | SUB          { $$ = new_term(F, iSub); }
            | MAX          { $$ = new_term(F, iMax); }
            | MIN          { $$ = new_term(F, iMin); }
            | MUL          { $$ = new_term(F, iMul); }
            | DIV          { $$ = new_term(F, iDiv); }
            | MOD          { $$ = new_term(F, iMod); }
            | SIN          { $$ = new_term(F, iSin); }
            | COS          { $$ = new_term(F, iCos); }
            | SQRT         { $$ = new_term(F, iSqrt); }
            | EXP          { $$ = new_term(F, iExp); }
            | ATAN2        { $$ = new_term(F, iAtan2); }
            | MYLT         { $$ = new_term(F, iLT); }
            | MYLTE        { $$ = new_term(F, iLTE); }
            | MYGT         { $$ = new_term(F, iGT); }
            | MYGTE        { $$ = new_term(F, iGTE); }
            | MYEQ         { $$ = new_term(F, iEqual); }
            | MYNEQ        { $$ = new_term(F, iNEqual); }
            | ASSIGN       { $$ = new_term(F, iAssign); }
            | TRUE         { $$ = new_term(D, iTrue); }
            | FALSE        { $$ = new_term(D, iFalse); }
            | CONS         { $$ = new_term(D, iHash); }
            | EMPTYLIST    { $$ = new_term(D, iEmptyList); }
            | DATA_CONSTRUCTOR_INT    { $$ = new_term_int($1); }
            | DATA_CONSTRUCTOR_FLOAT  { $$ = new_term_float($1); }
            | DATA_CONSTRUCTOR_CHAR
               { int code = insert_symbol($1); $$ = new_term(D, code); }
            | ⟨term schema::strings 127a⟩
            | IDENTIFIER1  { $$ = new_term(F, insert_symbol($1)); }
            | IDENTIFIER2  { $$ = new_term(D, insert_symbol($1)); }
            | '\\' VARIABLE '.' term_schema
                  { $$ = new_term(ABS);
                    // $$->lc = new_term(V,insert_symbol($2));
                    // $$->rc = $4;
```

```
            $$->insert(new_term(V,insert_symbol($2)));
            $$->insert($4);
          }
      | '\\' SYN_VARIABLE '.' term_schema
          { $$ = new_term(ABS);
            // $$->lc = new_term(SV,insert_symbol($2));
            // $$->rc = $4;
            $$->insert(new_term(SV,insert_symbol($2)));
            $$->insert($4);
          }
      | ⟨term schema::if-then-else statements 127b⟩
      | ⟨term schema::existential statements 128a⟩
      | ⟨term schema::universal statements 128b⟩
      | BOX DATA_CONSTRUCTOR_INT term_schema
           { $$ = new_term(MODAL); $$->modality = $2;
             /* $$->lc = $3; */ $$->insert($3); }
      | '(' term_schema term_schema ')'
           { $$ = new_term(APP);
             // $$->lc = $2; $$->rc = $3;
             $$->insert($2); $$->insert($3);
           }
      | ⟨term schema::syntactic sugar 129a⟩
      | '(' ')'   { $$ = new_term(PROD); }
      | ⟨term schema::products 129c⟩
      | ⟨term schema::sets 130⟩
      | ⟨term schema::lists 131⟩
      ;
```

⟨parser::term schemas 129b⟩
⟨parser::term schema products 129d⟩
⟨parser::sv condition 128c⟩

Comment 4.1.16. We have two kinds of strings: the first kind atomic; the second, composite. An atomic string is a single data constructor; just like characters, one can only define equality function on atomic strings, nothing else.

To define functions like substring that access the individual characters of a string, we need to represent a string as a composite object. A natural thing to do here is to represent a string as a list of characters. An atomic string is written "This is a string". A composite string is written StrList "This is a string".

127a ⟨term schema::strings 127a⟩≡

```
DATA_CONSTRUCTOR_STRING
{ int code = insert_symbol($1); strings.insert(code);
  $$ = new_term(D, code); }
| STRLIST DATA_CONSTRUCTOR_STRING
   { string x($2); int size = x.size();
     term * elist = new_term(D, iEmptyList);
     for (int i=size-2; i!=0; i-) {
             term * temp = newT2Args(D, iHash);
             string character("’"); character += x[i]; character += "’";
             int code = insert_symbol(character);
             temp->initT2Args(new_term(D, code), elist);
             elist = temp;
     }
     $$ = elist; /*$*/ }
```

Comment 4.1.17. We provide syntactic sugar for writing if-then-else statements here. The function have the following signature:

$$\mathit{if} - \mathit{then} - \mathit{else} : \Omega \times a \times a \rightarrow a.$$

Note that the domain is a tuple – this function should not be written in curried form.

127b ⟨term schema::if-then-else statements 127b⟩≡

```
IF term_schema THEN term_schema ELSE term_schema
{ $$ = new_term(APP);
  // $$->lc = new_term(F, iIte);
  $$->insert(new_term(F, iIte));
  term * temp = new_term(PROD);
  /* temp->tuple = $2;
  temp->tuple->next = $4;
  temp->tuple->next->next = $6;
  temp->tuple->next->next->next = NULL; */
  temp->insert($2); temp->insert($4); temp->insert($6);
  // $$->rc = temp;
  $$->insert(temp);
}
```

Comment 4.1.18. We provide syntactic sugars for writing existentially and universally quantified statements in a natural way.

128a ⟨term schema::existential statements 128a⟩≡

```
'\\' EXISTS VARIABLE '.' term_schema
   { $$ = new_term(APP);
     // $$->lc = new_term(F, iSigma);
     $$->insert(new_term(F, iSigma));
     term * abs = new_term(ABS);
     // abs->lc = new_term(V, insert_symbol($3)); abs->rc = $5;
     abs->insert(new_term(V, insert_symbol($3))); abs->insert($5);
     // $$->rc = abs;
     $$->insert(abs);
   }
```

128b ⟨term schema::universal statements 128b⟩≡

```
'\\' FORALL VARIABLE '.' term_schema
   { $$ = new_term(APP);
     // $$->lc = new_term(F, iPi);
     $$->insert(new_term(F, iPi));
     term * abs = new_term(ABS);
     // abs->lc = new_term(V, insert_symbol($3)); abs->rc = $5;
     abs->insert(new_term(V, insert_symbol($3))); abs->insert($5);
     // $$->rc = abs;
     $$->insert(abs);
   }
```

Comment 4.1.19. There is a small language for imposing side conditions on syntactical variables. See Comment 2.2.25.

128c ⟨parser::sv condition 128c⟩≡

```
sv_condition : '/' VAR '/' { $$ = new condition; $$->tag = CVAR; }
             | '/' CONST '/' { $$ = new condition; $$->tag = CCONST; }
             | '/' EQUAL ',' SYN_VARIABLE '/'
               { $$ = new condition; $$->tag = CEQUAL;
                 $$->svname = insert_symbol($4); }
             | '/' NOTEQUAL ',' SYN_VARIABLE '/'
               { $$ = new condition; $$->tag = CNOTEQUAL;
                 $$->svname = insert_symbol($4); }
             ;
```

Comment 4.1.20. A function applied to multiple arguments is painful to write. Here we introduce a syntactic sugar to allow users to write terms of the form $(f\ \ t_1 \ldots t_n)$ to mean $(\cdots(f\ t_1)\cdots t_n)$. The following variable is needed to remember terms.

128d ⟨parser::variables 120c⟩+≡

```
vector<term *> temp_fields;
```

129a ⟨term schema::syntactic sugar 129a⟩≡

```
  '(' term_schema term_schema term_schemas ')'
          { $$ = new_term(APP);
            // $$->lc = $2; $$->rc = $3;
            $$->insert($2); $$->insert($3);

            int size = temp_fields.size(); int psize = 0;
            while (temp_fields[size-1-psize] != NULL) psize++;

            term * temp;
            for (int i=size-psize; i!=size; i++) {
                    temp = new_term(APP);
                    // temp->lc = $$; temp->rc = temp_fields[i];
                    temp->insert($$);
                    temp->insert(temp_fields[i]);
                    $$ = temp;
            }
            while (psize+1) { temp_fields.pop_back(); psize-; }
          }
```

129b ⟨parser::term schemas 129b⟩≡

```
  term_schemas : term_schema
                 { temp_fields.push_back(NULL); // start a new mult app
                   temp_fields.push_back($1); }
               | term_schemas term_schema
                 { temp_fields.push_back($2); }
               ;
```

Comment 4.1.21. Products are handled in about the same way, except that we do not have to construct application nodes.

129c ⟨term schema::products 129c⟩≡

```
  '(' term_schemas_product ')'
          { $$ = new_term(PROD);
            int size = temp_fields.size(); int psize = 0;
            while (temp_fields[size-1-psize] != NULL) psize++;

            for (int i=size-psize; i!=size; i++) {
                $$->insert(temp_fields[i]);
            }
            while (psize+1) { temp_fields.pop_back(); psize-; }
          }
```

129d ⟨parser::term schema products 129d⟩≡

```
  term_schemas_product : term_schema
                         { temp_fields.push_back(NULL); // start a new product
                           temp_fields.push_back($1); }
                       | term_schemas_product ',' term_schema
                         { temp_fields.push_back($3); }
                       ;
```

Comment 4.1.22. We also provide syntactic sugar for extensional sets.

130 ⟨term schema::sets 130⟩≡

```
  ’{’ ’}’
       {  $$ = new_term(ABS);
          // $$->lc = new_term(V, newPVar());
          // $$->rc = new_term(D, iFalse);
          $$->insert(new_term(V, newPVar()));
          $$->insert(new_term(D, iFalse));
       }
  | ’{’ term_schemas_product ’}’
       {  int pv = newPVar();
          $$ = new_term(ABS); $$->insert(new_term(V, pv));

          term * arg2 = new_term(D, iFalse);
          int i = temp_fields.size()-1;
          while (temp_fields[i] != NULL) {
                  term * ite = new_term(APP);
                  ite->insert(new_term(F, iIte));
                  ite->insert(new_term(PROD));

                  term * eq = newT2Args(F, iEqual);
                  eq->initT2Args(new_term(V, pv), temp_fields[i]);

                  ite->fields[1]->insert(eq);
                  ite->fields[1]->insert(new_term(D, iTrue));
                  ite->fields[1]->insert(arg2);

                  arg2 = ite;
                  i-;
          }
          $$->insert(arg2); // $
          // setSelector(STDOUT); $$->print(); setSelector(SILENT);

          int size = temp_fields.size(); int psize = 0;
          while (temp_fields[size-1-psize] != NULL) psize++;
          while (psize+1) { temp_fields.pop_back(); psize-; }
        }
```

Comment 4.1.23. In the good tradition of functional programmingm we provide syntactic sugar for lists as well.

131 ⟨term schema::lists 131⟩≡

```
’[’ ’]’ {}
| ’[’ term_schemas_product ’]’
    {   int tsize = temp_fields.size();
        term * tail = newT2Args(D, iHash);
        tail->initT2Args(temp_fields[tsize-1], new_term(D, iEmptyList));
        int i = tsize - 2;
        while (temp_fields[i] != NULL) {
                term * current = newT2Args(D, iHash);
                current->initT2Args(temp_fields[i], tail);
                tail = current;
                i-;
        }
        $$ = tail;
        int size = temp_fields.size(); int psize = 0;
        while (temp_fields[size-1-psize] != NULL) psize++;
        while (psize+1) { temp_fields.pop_back(); psize-; }
    }
```

Comment 4.1.24. We need to declare the signature of every constant we use. This information is need for proper type checking of the program.

132 ⟨parser::type info 132⟩≡

```
type_info : functionsymbol ':' type ';'  { insert_constant($1, $3); }
          | constructordecl
          | syndecl
          ;
functionsymbol : IDENTIFIER1 { $$ = insert_symbol($1); }
              | SIGMA        { $$ = iSigma; }
              | PI           { $$ = iPi; }
              | AND          { $$ = iAnd; }
              | OR           { $$ = iOr; }
              | NOT          { $$ = iNot; }
              | IMPLIES      { $$ = iImplies; }
              | ITE          { $$ = iIte; }
              | IFF          { $$ = iIff; }
              | ADD          { $$ = iAdd; }
              | SUB          { $$ = iSub; }
              | MAX          { $$ = iMax; }
              | MIN          { $$ = iMin; }
              | MUL          { $$ = iMul; }
              | DIV          { $$ = iDiv; }
              | MOD          { $$ = iMod; }
              | MYLT         { $$ = iLT; }
              | MYLTE        { $$ = iLTE; }
              | MYGT         { $$ = iGT; }
              | MYGTE        { $$ = iGTE; }
              | MYEQ         { $$ = iEqual; }
              | MYNEQ        { $$ = iNEqual; }
              | ASSIGN       { $$ = iAssign; }
              | TRUE         { $$ = iTrue; }
              | FALSE        { $$ = iFalse; }
              | CONS         { $$ = iHash; }
              | EMPTYLIST    { $$ = iEmptyList; }
              ;
```

Comment 4.1.25. A collection of data constructors with a common signature can be declared by listing them followed by their common signature. We need to insert the signature as a user-defined type here so that we can recognise it when we see it again later. Each data constructor together with its signature is recorded for later type checking use as well.

133a ⟨parser::type info 132⟩+≡

```
constructordecl : dataconstructors ':' type ';'
                  { string tname($3->getName());
                    type * t = new type_udefined(tname, vec_constants);
                    if ($3->isUdefined())
                           insert_type(tname, UDEFINED, t);
                    insert_constant(insert_symbol(vec_constants[0]), $3);
                    for (unint i=1; i!=vec_constants.size(); i++)
                            insert_constant(insert_symbol(vec_constants[i]),
                                            $3->clone());
                    vec_constants.clear();
                    // if (!quiet) cerr « "prompt> ";
                  }
                ;
dataconstructors : dataconstructor  { vec_constants.push_back($1); }
                 | dataconstructors ',' dataconstructor
                   { vec_constants.push_back($3); }
                 ;
dataconstructor : IDENTIFIER2 { $$ = $1; }
                | DATA_CONSTRUCTOR { $$ = $1; }
                ;
```

Comment 4.1.26. We record the list of data constructors in this temporary vector.

133b ⟨parser::variables 120c⟩+≡

```
vector<string> vec_constants;
```

133c ⟨parser::type info 132⟩+≡

```
syndecl : TYPE IDENTIFIER2 MYEQ type ';'
           { string t($2); insert_type(t,SYNONYM,$4); }
        ;
```

Comment 4.1.27. We now give the grammar for types. The `->` function-forming operator is right associative; the `*` product-forming operator is left associative.

There are six system-defined types: `Bool`, `Int`, `Float`, `Char`, `String` and `ListString`. The first five are atomic types. The type `ListString` is translated into `(List Char)` by the system.

134a ⟨parser::type info 132⟩+≡

```
type : VARIABLE { string tname($1); $$ = new type_parameter(tname); }
     | IDENTIFIER1 { string tname($1); $$ = new type_parameter(tname); }
     | BOOL     { $$ = new type("Bool"); }
     | INT      { $$ = new type("Int"); }
     | FLOAT    { $$ = new type("Float"); }
     | CHAR     { $$ = new type("Char"); }
     | STRING   { $$ = new type("String"); }
     | LISTRING { $$ = new type_alg("List"); $$->addAlpha(new type("Char")); }
     | IDENTIFIER2
        { string tname($1);
          pair<int,type *> p = get_type(tname);
          if (p.second == NULL) $$ = new type_udefined(tname);
          else {  if (p.first == UDEFINED) $$ = p.second->clone();
                  else $$ = new type_synonym(tname, p.second->clone()); }
        }
     | '(' IDENTIFIER2 types ')'
        { string tname($2);
          type_tuple * rem = dcast<type_tuple *>(tempTuples.top());
          tempTuples.pop();
          $$ = new type_alg(tname, rem);
          delete_type(rem);
        }
     | '(' products ')' { $$ = tempTuples.top(); tempTuples.pop(); }
     | type ARROW type { $$ = new type_abs($1, $3); }
     | '(' type ')' { $$ = $2; }
     ;

products : products '*' type { tempTuples.top()->addAlpha($3); }
         | type '*' type     { tempTuples.push(new type_tuple);
                               tempTuples.top()->addAlpha($1);
                               tempTuples.top()->addAlpha($3);  }
         ;
types : type
         { tempTuples.push(new type_tuple); tempTuples.top()->addAlpha($1); }
      | types type
         { tempTuples.top()->addAlpha($2); }
      ;
```

134b ⟨parser::variables 120c⟩+≡

```
stack<type *> tempTuples;
```

134c ⟨yacc token definitions 117d⟩+≡

```
%right ARROW
%left '*'
```

Comment 4.1.28. We may want to use the `data` statement of Haskell for declaring data constructors in the future. There are some issues that need to be resolved first, however.

An algebraic data type declaration in Haskell does not end with a delimiter. That is a bit strange because I need to put a delimiter (a semi-colon) to make the grammar unambiguous. The problem is related to the limitted lookahead mechanism of Yacc. Consider the following two statements.

```
data List a = Nillist | Cons a (List a)
f x = 2 * x
```

Yacc cannot tell whether `f` is a parameter for `Cons` or the start of another statement. It cannot know this until it sees the `=` sign two tokens down the track.

The Haskell 98 report [Pey02] gives the following grammar for constructors:

$$\textit{constr} \rightarrow \textit{con}\ [!]\,\textit{atype}_1 \ldots [!]\,\textit{atype}_k.$$

It is not clear to me what [!] means here. Maybe that holds the key to proper parsing without the need for a delimiter.

135 ⟨parser::type info::unused 135⟩≡

```
algebraic_type : "data" IDENTIFIER2 parameters MYEQ data_constructors ;

parameters : /* nothing */ | parameters IDENTIFIER1 ;

data_constructors : data_constructor
                  | data_constructor '|' data_constructors ;
data_constructor : IDENTIFIER2 brtypes ;

brtypes : /* nothing */ | brtypes brtype ;

brtype : IDENTIFIER1 | IDENTIFIER2 | '(' IDENTIFIER2 types ')' | '(' type ')' ;
```

4.1.3 Escher Main Program

136a ⟨escher main program 136a⟩≡
```
#ifndef __APPLE__
#ifndef __sun
#include <getopt.h>
#endif
#endif

#include <unistd.h>
#include <signal.h>

static void handle_signal(int sig) {
        cout « "Interrupted....\n";
        if (interrupted) { cleanup(); exit(1); }
        interrupted = true;
}

int main(int argc, char ** argv) {
        interactive = false; quiet = true;
        char c;
        while ((c = getopt(argc, argv, "vitobds")) != EOF) {
                switch (c) {
                case 'v': verbose++; break;
                case 'i': interactive = true; break;
                case 't': typecheck = false; break;
                case 'o': optimise = true; break;
                case 'b': backchain = true; break;
                case 'd': outermost = true; break;
                case 's': stepByStep = true; break;
                }
        }
        if (verbose) setSelector(STDOUT); else setSelector(SILENT);
        makeHeap();
        initFuncTable();
        initialise_constants();

        signal(SIGINT, handle_signal);

        logcache = fopen("log.cache", "r+"); assert(logcache);

        if (interactive) cerr « "prompt> ";
        do { yyparse(); } while (!feof(yyin));

        fclose(logcache);

        cleanup();
        return 0;
}
```
Uses `cleanup` 137c 137d.

Comment 4.1.29. Error reporting is not quite right yet. The line number reported is wrong because of nested imports.

136b ⟨parser::variables 120c⟩+≡

```
extern int yylineno; extern char * yytext; extern char linebuf[500];
extern int tokenpos;
```

137a ⟨parser::function declarations 120d⟩+≡

```
void yyerror(const char * s);
```

137b ⟨parser::error reporting 137b⟩≡

```
void yyerror(const char * s) {
        cerr « yylineno « ": " « s « " at " « yytext « " in this line\n";
        cerr « linebuf « endl;
        for (int i=0; i!=tokenpos-1; i++) cerr « " ";
        cerr « "^" « endl;
        if (!quiet) cerr « "prompt> ";
}
```

Comment 4.1.30. This function frees the memory held by the program modules.

137c ⟨parser::function declarations 120d⟩+≡

```
void cleanup();
```

Defines:
`cleanup`, used in chunks 120a and 136a.

137d ⟨escher main program 136a⟩+≡

```
void cleanup() {
        cleanup_statements(); cleanup_formulas();
        cleanup_constants(); cleanup_synonyms();
        mem_report();
}
```

Defines:
`cleanup`, used in chunks 120a and 136a.

Chapter 5

Global Data Structures

138 ⟨global.h 138⟩≡

```
#ifndef _ESCHER_GLOBAL_H_
#define _ESCHER_GLOBAL_H_

#include <algorithm>
#include <vector>
#include <string>
#include <set>
#include <math.h>
#include <stdlib.h>
#include "terms.h"
#include "types.h"
#include "unification.h"
using namespace std;

⟨global:data types 139a⟩
⟨global:external variables 143c⟩
⟨global:external functions 148b⟩

// extern vector<vector<term_type> > stat_term_types;
extern set<int> cacheFuncs;
extern set<int> strings;
extern const string pve;

⟨global symbol constants 148a⟩

#define UDEFINED 0
#define SYNONYM  1

#endif
```

Uses `term_type` 17b.

Comment 5.0.31. The variable `ltime` records the total number of computation steps taken to simplify the query. The variable `cltime` records the total number of steps computed using cached information. Statements in the input Escher program are stored in a vector. Each statement is stored in a structure called `statementType`. The fields `numargs` and `anchor` are used to pick out unsuitable statements during pattern matching.

139a ⟨global:data types 139a⟩≡

```
  // these are the escher statements
  struct statementType {
        vector<int> modalContext; // this is used in Bach only
        vector<int> quantifiedVars; // this is used in Bach only
        term ∗ stmt;
        int numargs;
        int anchor;
        bool typechecked;
        bool lastresort;
        bool eager;
        bool persistent;
        bool noredex;
        bool collectstats; int usestats;
        statementType ∗ next;
        statementType() {
              anchor = -5; typechecked = false; lastresort = false;
              eager = false; persistent = false; noredex = false;
              collectstats = false; usestats = 0;
              next = NULL;
        }
        void freememory() { stmt→freememory(); }
        void print() { stmt→print(); ioprintln(); if (next) next→print(); }
  };
  extern vector<statementType ∗> grouped_statements;
  extern vector<statementType ∗> statements;
  extern vector<statementType ∗> cachedStatements;
```

Uses `ioprintln` 164 165.

139b ⟨global:data types 139a⟩+≡

```
  struct formulaType {
        term ∗ fml;
        bool globalass;
        bool typechecked;
        formulaType() { globalass = false; typechecked = false; }
        void freememory() { fml→freememory(); }
  };
  extern vector<formulaType> formulas;
```

Comment 5.0.32. The following is the data structure for storing edits.

140a
```
⟨global:data types 139a⟩+≡
  struct BN_node {
      int vname; term ∗ density; BN_node ∗ next;
      BN_node() { density = NULL; next = NULL; }
      BN_node ∗ clone() { ⟨BN node:clone 141c⟩ }
      void freememory() {
          if (density) density→freememory();
          if (next) next→freememory(); }
      void print() { ⟨BN node:print 141a⟩ }
      void subst(vector<substitution> & theta) {
          density→subst(theta); if (next) next→subst(theta); }
  };
```

Defines:
  `BN_node`, used in chunks 140 and 141c.
Uses `subst` 52a and `substitution` 51.

140b
```
⟨global:data types 139a⟩+≡
  struct editType {
      term ∗ head;
      term ∗ body;
      type ∗ htype;
      string htype_name;
      BN_node ∗ bnodes;
      editType ∗ next;
      editType() { head = NULL; body = NULL; bnodes = NULL; next = NULL; }
      editType ∗ clone() { ⟨editType:clone 141d⟩ }
      void freememory() { ⟨editType:freememory 141e⟩ }
      void subst(vector<substitution> & theta) { ⟨editType:subst 140c⟩ }
      void print() { ⟨editType:print 141b⟩ }
  };
  extern vector<editType ∗> edits;
```

Defines:
  `editType`, used in chunks 141–43.
Uses `BN_node` 140a, `subst` 52a, and `substitution` 51.

Comment 5.0.33. We do the appropriate term substitutions on the parts of the conditional edit, taking care that the domain of the input substitution does not overlap the variables defined in the bayes net.

140c
```
⟨editType:subst 140c⟩≡
  // make sure theta does not bound variables in bnodes
  for (unint i=0; i≠theta.size(); i++) {
      BN_node ∗ pt = bnodes;
      while (pt≠NULL) { assert(theta[i].first ≠ pt→vname); pt = pt→next; }
  }
  // do the substitution
  body→subst(theta);
  if (bnodes) bnodes→subst(theta);
  if (next) next→subst(theta);
```

Uses `BN_node` 140a, `subst` 52a, and `substitution` 51.

Comment 5.0.34. Following are print routines for the two data structures above.

141a ⟨BN node:print 141a⟩≡

```
ioprint(getString(vname)); ioprint(" ~ "); density→print();
if (next ≠ NULL) { ioprint(", "); next→print(); }
```

Uses `getString` 147 and `ioprint` 164 165.

141b ⟨editType:print 141b⟩≡

```
head→print(); ioprint(" ~> "); body→print();
if (bnodes ≠ NULL) { ioprint(" ["); bnodes→print(); ioprint(" ]"); }
ioprint(" of type "); ioprint(htype→getName()); ioprint("\n");
if (next) next→print();
```

Uses `ioprint` 164 165.

141c ⟨BN node:clone 141c⟩≡

```
BN_node * ret = new BN_node;
ret→vname = vname; ret→density = density→clone();
if (next) ret→next = next→clone();
return ret;
```

Uses `BN_node` 140a.

Comment 5.0.35. In cloning an `editType`, we always have to work out the free variables because this calculation is relied upon by the pattern-matching routine.

141d ⟨editType:clone 141d⟩≡

```
editType * ret = new editType;
ret→head = head→clone(); ret→body = body→clone();
ret→head→labelStaticBoundVars(); ret→body→labelStaticBoundVars();
ret→htype = htype→clone();
if (bnodes) ret→bnodes = bnodes→clone();
if (next) ret→next = next→clone();
return ret;
```

Uses `editType` 140b and `labelStaticBoundVars` 46f.

141e ⟨editType:freememory 141e⟩≡

```
if (head) head→freememory();
if (body) body→freememory();
if (htype) delete_type(htype);
if (bnodes) bnodes→freememory();
if (next) next→freememory();
```

Uses `delete_type` 10a 10b.

Comment 5.0.36. This is a data structure for storing conditional edit grammars.

```
142  ⟨global:data types 139a⟩+≡
  struct CEG_node {
        term ∗ cond;
        editType ∗ editg;
        CEG_node ∗ lt, ∗ rt;
        CEG_node() { cond = NULL; editg = NULL; lt = NULL; rt = NULL; }
        CEG_node ∗ clone() {
              CEG_node ∗ ret = new CEG_node;
              if (cond ≠ NULL) ret→cond = cond→clone();
              if (editg ≠ NULL) ret→editg = editg→clone();
              if (lt ≠ NULL) { ret→lt = lt→clone(); ret→rt = rt→clone();}
              return ret;
        }
        void freememory() {
              if (cond) cond→freememory();
              if (editg) editg→freememory();
              if (lt) lt→freememory();
              if (rt) rt→freememory();
        }
        void print() {
              setSelector(STDERR);
              if (cond) {
                    ioprint("if "); cond→print(); ioprint(" then \n");
                    lt→print(); ioprint("\n");
                    ioprint("else\n");
                    rt→print(); ioprint("\n");
              } else editg→print();
        }
  };
  extern CEG_node ∗ condEditG;
  extern CEG_node ∗ instEditG;
```

Uses `editType` 140b, `ioprint` 164 165, and `setSelector` 164 165.

143a ⟨global.cc 143a⟩≡

```
#include "global.h"
#include <stdlib.h>
#include <cassert>
using namespace std;

vector<statementType *> grouped_statements;
vector<statementType *> statements;
// vector<vector<term_type> > stat_term_types;
vector<formulaType> formulas;
vector<statementType *> cachedStatements;
vector<editType *> edits;
CEG_node * condEditG = NULL, * instEditG = NULL;
set<int> cacheFuncs;
set<int> strings; // used to record strings in Bach programs

⟨run-time options 143b⟩
⟨string constants 144a⟩

⟨symbols and their integer representations 145⟩
⟨statements and type checking 149a⟩
⟨constants and their signatures 151⟩
⟨function symbol table 154a⟩
⟨nonrigid constants 156c⟩
⟨type name to type objects mapping 157a⟩
⟨statements insertion and printing 158a⟩
⟨misc functions 159a⟩
```

Uses `editType` 140b and `term_type` 17b.

Comment 5.0.37. These are variables that record the run-time options specified by the user.

143b ⟨run-time options 143b⟩≡

```
int ltime = 0; int cltime = 0;
int verbose = 0; bool typecheck = true; bool optimise = false;
bool backchain = false; bool outermost = false; bool externalIO = false;
FILE * logcache = NULL;
bool interrupted = false;
bool stepByStep = false;
vector<int> queryModalContext;
```

143c ⟨global:external variables 143c⟩≡

```
extern int ltime; extern int cltime;
/* options */
extern int verbose; extern bool typecheck;
extern bool optimise; extern bool backchain; extern bool outermost;
extern bool externalIO;
extern vector<int> queryModalContext;

extern bool interrupted;
extern bool stepByStep;
/* log files */
extern FILE * logcache;
```

Comment 5.0.38. Strings for the type system.

144a ⟨string constants 144a⟩≡

```
const string underscore = "_";
const string alpha = "alpha";
const string Parameter = "Parameter";
const string Tuple = "Tuple";
const string Arrow = "Arrow";
const string gBool = "Bool", gChar = "Char", gString = "String";
const string gInt = "Int", gFloat = "Float";
const string pve = "pve";
```

Comment 5.0.39. Output strings for system-level equational simplification routines.

144b ⟨string constants 144a⟩+≡

```
const string eqsimpl = "Equalities simplification\n";
const string andsimpl = "And rule simplification\n";
const string and2simpl = "And2 rule simplification\n";
const string ineqsimpl = "Inequalities simplification\n";
const string arsimpl = "Arithmetic simplification\n";
const string exsimpl = "Existential rule simplification\n";
const string uvsimpl = "Universal rule simplification\n";
const string betasimpl = "Beta reduction\n";
const string mathsimpl = "Math library function call\n";
const string itesimpl = "If-then-else rule simplification\n";
const string modalsimpl = "Modal term simplification\n";
```

144c ⟨global:external variables 143c⟩+≡

```
extern const string eqsimpl, andsimpl, and2simpl, ineqsimpl, arsimpl,
      exsimpl, uvsimpl, betasimpl, mathsimpl, itesimpl, modalsimpl;
```

Comment 5.0.40. Output strings for tableaux rules in the theorem prover.

144d ⟨string constants 144a⟩+≡

```
const string substitutionRuleId = "=";
const string negationRuleId = "~~";
const string conjunctionRuleId = "&";
const string disjunctionRuleId = "v";
const string reflexiveRuleId = "Id";
const string existentialRuleId = "E";
const string universalRuleId = "U";
const string universalSPImpliesRuleId = "USI";
const string bachRuleId = "Bc";
const string closureRuleId = "C";
const string uclosureRuleId = "UI";
const string diamondRuleId = "<>";
const string boxRuleId = "[]";
const string kRuleId = "K";
```

144e ⟨global:external variables 143c⟩+≡

```
extern const string substitutionRuleId, negationRuleId, conjunctionRuleId,
      disjunctionRuleId, reflexiveRuleId, kRuleId,
      existentialRuleId, universalRuleId, universalSPImpliesRuleId,
      bachRuleId, closureRuleId, uclosureRuleId, diamondRuleId, boxRuleId;
```

Comment 5.0.41. For efficiency reasons, we do not want to deal with the string representations of symbols in the system. Each symbol is mapped to an integer, and the mappings are recorded here.

145 ⟨symbols and their integer representations 145⟩≡

```
const int iNot = 1001;        const string gNot = "not";
const int iAnd = iNot + 1;    const string gAnd = "&&";
const int iOr = iAnd + 1;     const string gOr = "||";
const int iImplies = iOr + 1; const string gImplies = "implies";
const int iIff = iImplies +1; const string gIff = "iff";
const int iPi = iIff + 1;     const string gPi = "pi";
const int iSigma = iPi + 1;   const string gSigma = "sigma";
const int iEqual = iSigma +1; const string gEqual = "=";
const int iIte = iEqual + 1;  const string gIte = "ite";
const int iTrue = iIte + 1;   const string gTrue = "True";
const int iFalse = iTrue + 1; const string gFalse = "False";
const int iHash = iFalse + 1; const string gHash = "#";
const int iEmptyList = iHash + 1;  const string gEmptyList = "[]";
const int iInfinity = iEmptyList + 1; const string gInfinity = "Infinity";
const int iAdd = iInfinity + 1;   const string gAdd = "add";
const int iSub = iAdd + 1;     const string gSub = "sub";
const int iMax = iSub + 1;     const string gMax = "max";
const int iMin = iMax + 1;     const string gMin = "min";
const int iMul = iMin + 1;     const string gMul = "mul";
const int iDiv = iMul + 1;     const string gDiv = "div";
const int iMod = iDiv + 1;     const string gMod = "mod";
const int iAtan2 = iMod + 1;  const string gAtan2 = "atan2";
const int iLT = iAtan2 + 1;   const string gLT = "<";
const int iLTE = iLT + 1;      const string gLTE = "<=";
const int iGT = iLTE + 1;      const string gGT = ">";
const int iGTE = iGT + 1;      const string gGTE = ">=";
const int iNEqual = iGTE +1;  const string gNEqual = "/=";
const int iAssign = iNEqual+1;  const string gAssign = ":=";
const int iTpHelp = iAssign+1;  const string gTpHelp = "tpHelp";
const int iTpTag = iTpHelp +1;  const string gTpTag = "TpTag";
const int iSucceeded = iTpTag+1; const string gSucceeded = "Succeeded";
const int iFailed = iSucceeded+1;  const string gFailed = "Failed";
const int iDontKnow = iFailed+1;   const string gDontKnow = "DontKnow";
const int iSin = iDontKnow+1; const string gSin = "sin";
const int iCos = iSin + 1;    const string gCos = "cos";
const int iSqrt = iCos + 1;   const string gSqrt = "sqrt";
const int iExp = iSqrt + 1;   const string gExp = "exp";
const int iUniform = iExp + 1; const string gUniform = "uniform";
const int iCategorical = iUniform+1; const string gCategorical = "categorical";
const int iGaussian = iCategorical + 1; const string gGaussian = "gaussian";
const int iPoint = iGaussian + 1; const string gPoint = "point";
const int iDGaussian = iPoint + 1; const string gDGaussian = "dgaussian";
const int iWildcard = iDGaussian + 1; const string gWildcard = "_";
```

Defines:

`iAdd`, used in chunks 65, 66, 147, 148a, 152a, and 154c.
`iAnd`, used in chunks 58, 59b, 63a, 64a, 71b, 74, 76c, 77a, 79b, 90, 147, and 148a.
`iAssign`, used in chunks 93b, 147, 148a, 152c, and 154c.
`iAtan2`, used in chunks 65, 147, 148a, 152a, and 154c.
`iCategorical`, used in chunks 147, 148a, 151, and 163a.
`iCos`, used in chunks 68, 147, 148a, 152a, and 154c.
`iDGaussian`, used in chunks 147, 148a, 151, and 163a.

`iDiv`, used in chunks 66, 147, 148a, 152a, and 154c.
`iDontKnow`, used in chunks 107a, 147, and 148a.
`iEmptyList`, used in chunks 36, 62d, 147, 148a, 151, 159c, and 160.
`iEqual`, used in chunks 63a, 64a, 69b, 72a, 77b, 89b, 90, 147, 148a, 151, and 154c.
`iExp`, used in chunks 68, 147, 148a, 152a, and 154c.
`iFailed`, used in chunks 147, 148a, and 152c.
`iFalse`, used in chunks 56a, 58, 62d, 64a, 67, 75c, 78c, 147, 148a, and 151.
`iGT`, used in chunks 67, 147, 148a, 152b, and 154c.
`iGTE`, used in chunks 67, 147, 148a, 152b, and 154c.
`iGaussian`, used in chunks 147, 148a, 151, and 163a.
`iHash`, used in chunks 33a, 36b, 147, 148a, 151, 159c, and 160.
`iIff`, used in chunks 58, 147, and 148a.
`iImplies`, used in chunks 58, 78b, 147, and 148a.
`iInfinity`, used in chunks 65, 67, 147, 148a, and 151.
`iIte`, used in chunks 36c, 69b, 147, and 148a.
`iLT`, used in chunks 67, 147, 148a, 152b, and 154c.
`iLTE`, used in chunks 67, 147, 148a, 152b, and 154c.
`iMax`, used in chunks 65, 147, 148a, 152a, and 154c.
`iMin`, used in chunks 65, 147, 148a, 152a, and 154c.
`iMod`, used in chunks 65, 147, 148a, 152a, and 154c.
`iMul`, used in chunks 66, 147, 148a, 152a, and 154c.
`iNEqual`, used in chunks 147, 148a, and 154c.
`iNot`, used in chunks 55b, 56a, 58–60, 107a, 147, and 148a.
`iOr`, used in chunks 58, 59, 147, and 148a.
`iPi`, used in chunks 35, 58, 78, 147, and 148a.
`iPoint`, used in chunks 147, 148a, 151, 154c, and 163a.
`iSigma`, used in chunks 35, 58, 73–75, 147, and 148a.
`iSin`, used in chunks 68, 147, 148a, 152a, and 154c.
`iSqrt`, used in chunks 68, 147, 148a, 152a, and 154c.
`iSub`, used in chunks 66, 147, 148a, 152a, and 154c.
`iSucceeded`, used in chunks 93b, 147, 148a, and 152c.
`iTpHelp`, used in chunks 105, 147, and 148a.
`iTpTag`, used in chunks 105, 107a, 147, and 148a.
`iTrue`, used in chunks 56a, 58, 62–64, 67, 75–78, 107a, 147, 148a, and 151.
`iUniform`, used in chunks 147, 148a, 151, and 163a.
`iWildcard`, used in chunks 24a, 101c, 147, and 148a.

146 ⟨symbols and their integer representations 145⟩+≡

```
vector<string> symbolsMap;
vector<string> charsMap; // characters are encoded using numbers in the range
                         // [2000,2999]
int insert_symbol(const string & symbol) {
        if (symbol[0] ≡ ’\’’) {
                for (unint i=0; i≠charsMap.size(); i++)
                        if (charsMap[i] ≡ symbol) return 2000+i;
                charsMap.push_back(symbol);
                int csize = charsMap.size(); assert(csize ≤ 1000);
                return 2000+csize-1;
        }
        for (unint i=0; i≠symbolsMap.size(); i++)
                if (symbolsMap[i] ≡ symbol) return i+1;
        symbolsMap.push_back(symbol);
        int csize = symbolsMap.size(); assert(csize < 1000);
        return csize;
}
```

Defines:
`insert_symbol`, used in chunk 148b.

Comment 5.0.42. The next function returns the string encoded by the input integer.

147 ⟨symbols and their integer representations 145⟩+≡

```
const string gError = "Error";
const string & getString(int code) {
        if (0 < code ∧ code ≤ (int)symbolsMap.size())
                return symbolsMap[code-1];
        if (2000 ≤ code ∧ code < 2000+(int)charsMap.size())
                return charsMap[code-2000];
        switch (code) {
        case iNot: return gNot; case iAnd: return gAnd;
        case iOr: return gOr;   case iImplies: return gImplies;
        case iIff: return gIff; case iPi: return gPi;
        case iSigma: return gSigma; case iEqual: return gEqual;
        case iIte: return gIte;     case iTrue: return gTrue;
        case iFalse: return gFalse; case iHash: return gHash;
        case iEmptyList: return gEmptyList;
        case iInfinity: return gInfinity;
        case iAdd: return gAdd;
        case iSub: return gSub; case iMax: return gMax;
        case iMin: return gMin; case iMul: return gMul;
        case iDiv: return gDiv; case iMod: return gMod;
        case iLT: return gLT;   case iLTE: return gLTE;
        case iGT: return gGT;   case iGTE: return gGTE;
        case iNEqual: return gNEqual; case iAssign: return gAssign;
        case iTpHelp: return gTpHelp; case iTpTag: return gTpTag;
        case iSucceeded: return gSucceeded; case iFailed: return gFailed;
        case iDontKnow: return gDontKnow;
        case iSin: return gSin;
        case iCos: return gCos;
        case iSqrt: return gSqrt;
        case iExp: return gExp;
        case iAtan2: return gAtan2;
        case iUniform: return gUniform;
        case iCategorical: return gCategorical;
        case iGaussian: return gGaussian;
        case iDGaussian: return gDGaussian;
        case iPoint: return gPoint;
        case iWildcard: return gWildcard;
        }
        cerr ≪ "code = " ≪ code ≪ endl; assert(false);
        return gError;
}
```

Defines:
`getString`, used in chunks 35–37, 110a, 111, 141a, 148b, 153, and 155b.

Uses `iAdd` 145, `iAnd` 145, `iAssign` 145, `iAtan2` 145, `iCategorical` 145, `iCos` 145, `iDGaussian` 145, `iDiv` 145, `iDontKnow` 145, `iEmptyList` 145, `iEqual` 145, `iExp` 145, `iFailed` 145, `iFalse` 145, `iGT` 145, `iGTE` 145, `iGaussian` 145, `iHash` 145, `iIff` 145, `iImplies` 145, `iInfinity` 145, `iIte` 145, `iLT` 145, `iLTE` 145, `iMax` 145, `iMin` 145, `iMod` 145, `iMul` 145, `iNEqual` 145, `iNot` 145, `iOr` 145, `iPi` 145, `iPoint` 145, `iSigma` 145, `iSin` 145, `iSqrt` 145, `iSub` 145, `iSucceeded` 145, `iTpHelp` 145, `iTpTag` 145, `iTrue` 145, `iUniform` 145, and `iWildcard` 145.

148a ⟨global symbol constants 148a⟩≡

```
extern const int iNot, iAnd, iOr, iImplies, iIff, iPi, iSigma, iEqual, iIte,
   iTrue, iFalse, iHash, iEmptyList, iInfinity, iAdd, iSub, iMax, iMin, iMul,
   iDiv, iMod, iLT, iLTE, iGT, iGTE, iNEqual, iAssign, iTpHelp, iTpTag,
   iSucceeded, iFailed, iDontKnow, iSin, iCos, iSqrt, iExp, iAtan2,
   iUniform, iCategorical, iGaussian, iDGaussian, iPoint, iWildcard;
```

Uses `iAdd` 145, `iAnd` 145, `iAssign` 145, `iAtan2` 145, `iCategorical` 145, `iCos` 145, `iDGaussian` 145, `iDiv` 145, `iDontKnow` 145, `iEmptyList` 145, `iEqual` 145, `iExp` 145, `iFailed` 145, `iFalse` 145, `iGT` 145, `iGTE` 145, `iGaussian` 145, `iHash` 145, `iIff` 145, `iImplies` 145, `iInfinity` 145, `iIte` 145, `iLT` 145, `iLTE` 145, `iMax` 145, `iMin` 145, `iMod` 145, `iMul` 145, `iNEqual` 145, `iNot` 145, `iOr` 145, `iPi` 145, `iPoint` 145, `iSigma` 145, `iSin` 145, `iSqrt` 145, `iSub` 145, `iSucceeded` 145, `iTpHelp` 145, `iTpTag` 145, `iTrue` 145, `iUniform` 145, and `iWildcard` 145.

148b ⟨global:external functions 148b⟩≡

```
/* symbol table */
extern int insert_symbol(const string & symbol);
extern const string & getString(int code);
```

Uses `getString` 147 and `insert_symbol` 146.

Comment 5.0.43. We now see how variables are handled. System-generated variables have integer representations above 5000. Standard variables generated by the system are encoded using values in the range 5000 to 99999. Fresh variables of this type are obtained using `newPVar()`. A variable with code 5013, for example, corresponds to a variable `pve13`. Free universal variables generated by the universal rule in the theorem proving part of the system are encoded using values above 100000.

148c ⟨symbols and their integer representations 145⟩+≡

```
static unsigned int varInt = 5000;
static unsigned int uvarInt = 100000;
int newPVar() { assert(varInt < 100000); return varInt++; }
int newUVar() { return uvarInt++; }
```

Defines:
`newPVar`, used in chunks 54a, 103c, and 111.
`newUVar`, used in chunk 111.

Comment 5.0.44. Here we just systematically go through the statements and type check each one. We need to record the inferred type for each subterm of the statement. There is a check to make sure that the indices for `statements` and `stat_term_types` matches; that is, the *i*-th element in the latter contains information about the *i*-th element in the former.

149a ⟨statements and type checking 149a⟩≡

```
bool typeCheck() {
        if (¬typecheck) return true;
        cerr ≪ "Type checking statements...";
        int size = grouped_statements.size();
        for (int i=0; i≠size; i++) {
                if (grouped_statements[i] ≡ NULL) continue;
                statementType ∗ sts = grouped_statements[i];
                while (sts ≠ NULL) {
                        if (sts→typechecked) { sts = sts→next; continue; }
                        type ∗ res = wellTyped(sts→stmt);
                        if (res) { delete_type(res);
                                   sts→typechecked = true;
                        } else return false;
                        sts = sts→next;
                }
        }

        size = statements.size();
        for (int i=0; i≠size; i++) {
                if (statements[i]→typechecked) continue;
                type ∗ res = wellTyped(statements[i]→stmt);
                if (res) { delete_type(res);
                           statements[i]→typechecked = true;
                } else return false;
        }
        cerr ≪ "done.\n";
        // cerr << "Type checking formulas...";
        size = formulas.size();
        for (int i=0; i≠size; i++) {
                if (formulas[i].typechecked) continue;
                type ∗ res = wellTyped(formulas[i].fml);
                if (res) { delete_type(res); formulas[i].typechecked = true; }
                else return false;
        }
        // cerr << "done.\n";
        return true;
}
```

Defines:
  `typeCheck`, used in chunk 149b.
Uses `delete_type` 10a 10b and `wellTyped` 28a.

149b ⟨global:external functions 148b⟩+≡

```
extern bool typeCheck();
```

Uses `typeCheck` 149a.

Comment 5.0.45. Here we release the memory occupied by the statements and the data structures supporting side conditions on them. We do not have to free the term part of `stat_term_types` because they point to subterms of terms residing in the `statements` vector.

150a ⟨statements and type checking 149a⟩+≡

```
void cleanup_statements() {
        cerr ≪ "Cleaning up statements...";
        for (unint i=0; i≠grouped_statements.size(); i++) {
                if (grouped_statements[i] ≡ NULL) continue;
                statementType ∗ sts = grouped_statements[i];
                while (sts ≠ NULL) {
                        sts→freememory();
                        sts = sts→next;
                }
        }
        for (unint i=0; i≠statements.size(); i++)
                statements[i]→freememory();
        cerr ≪ "Done.\n";
        cerr ≪ "Cleaning up " ≪ cachedStatements.size() ≪
                " cached statements...";
        for (unint i=0; i≠cachedStatements.size(); i++)
                cachedStatements[i]→freememory();
        cerr ≪ "Done.\n";
}
```

Defines:
`cleanup_statements`, used in chunk 150c.

150b ⟨statements and type checking 149a⟩+≡

```
void cleanup_formulas() {
        cerr ≪ "Cleaning up formulas...";
        for (unint i=0; i≠formulas.size(); i++) formulas[i].freememory();
        cerr ≪ "Done.\n";
}
```

Defines:
`cleanup_formulas`, used in chunk 150c.

150c ⟨global:external functions 148b⟩+≡

```
extern void cleanup_statements();
extern void cleanup_formulas();
```

Uses `cleanup_formulas` 150b and `cleanup_statements` 150a.

Comment 5.0.46. We now describe a facility that supports the storage and retrieval of the declared signatures of constants.

151 ⟨constants and their signatures 151⟩≡

```
struct constant_sig { int name; type ∗ signature; };
vector<constant_sig> constants;

void initialise_constants() {
        constant_sig temp;
        temp.name = iTrue; temp.signature = new type(gBool);
        constants.push_back(temp);
        temp.name = iFalse; temp.signature = new type(gBool);
        constants.push_back(temp);
        type ∗ a = new type_parameter("a");
        type ∗ lista = new type_alg("List"); lista→addAlpha(a);
        temp.name = iEmptyList; temp.signature = lista;
        constants.push_back(temp);
        temp.name = iHash;
        temp.signature = new type_abs(a→clone(),
                          new type_abs(lista→clone(), lista→clone()));
        constants.push_back(temp);

        temp.name = iEqual;
        temp.signature = new type_abs(a→clone(),
                             new type_abs(a→clone(), new type(gBool)));
        constants.push_back(temp);

        temp.name = iInfinity; temp.signature = new type_parameter("number");
        constants.push_back(temp);

        temp.name = iUniform;
        temp.signature = new type_abs(lista→clone(),
                                new type_abs(a→clone(), new type(gFloat)));
        constants.push_back(temp);

        temp.name = iCategorical;
        temp.signature = temp.signature→clone();
        constants.push_back(temp);

        temp.name = iGaussian;
        temp.signature = new type_abs(new type(gFloat),
                           new type_abs(new type(gFloat),
                                      new type_abs(new type(gFloat),
                                                 new type(gFloat))));
        constants.push_back(temp);

        temp.name = iDGaussian;
        temp.signature = new type_abs(new type(gFloat),
                           new type_abs(new type(gFloat),
                                      new type_abs(new type(gFloat),
                                                 new type(gFloat))));
        constants.push_back(temp);

        temp.name = iPoint;
```

```
        temp.signature = new type_abs(a→clone(),
                                      new type_abs(a→clone(), new type(gFloat)));
        constants.push_back(temp);

        ⟨initialise constants::arithmetic operations 152a⟩
        ⟨initialise constants::relational operations 152b⟩
        ⟨initialise constants::disruptive operations 152c⟩
    }
```

Defines:
`initialise_constants`, used in chunk 153e.
Uses `iCategorical` 145, `iDGaussian` 145, `iEmptyList` 145, `iEqual` 145, `iFalse` 145, `iGaussian` 145, `iHash` 145, `iInfinity` 145, `iPoint` 145, `iTrue` 145, and `iUniform` 145.

152a ⟨initialise constants::arithmetic operations 152a⟩≡

```
    type ∗ number = new type_parameter("number");
    type ∗ number2 = new type_parameter("number2");
    type ∗ number3 = new type_parameter("number3");
    type ∗ algtype = new type_abs(number, new type_abs(number2,number3));
    temp.name = iAdd; temp.signature = algtype; constants.push_back(temp);
    temp.name = iSub; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iMax; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iMin; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iMul; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iMod; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iDiv; temp.signature = algtype→clone(); constants.push_back(temp);
    temp.name = iAtan2;temp.signature = algtype→clone(); constants.push_back(temp);
    type ∗ tempsig = new type_abs(number2→clone(),number3→clone()) ;
    temp.name = iSin; temp.signature = tempsig; constants.push_back(temp);
    temp.name = iCos; temp.signature = tempsig→clone(); constants.push_back(temp);
    temp.name = iSqrt; temp.signature = tempsig→clone(); constants.push_back(temp);
    temp.name = iExp; temp.signature = tempsig→clone(); constants.push_back(temp);
```

Uses `iAdd` 145, `iAtan2` 145, `iCos` 145, `iDiv` 145, `iExp` 145, `iMax` 145, `iMin` 145, `iMod` 145, `iMul` 145, `iSin` 145, `iSqrt` 145, and `iSub` 145.

152b ⟨initialise constants::relational operations 152b⟩≡

```
    type ∗ reltype = new type_abs(number→clone(),
                                  new type_abs(number2→clone(), new type(gBool)));
    temp.name = iGT; temp.signature = reltype; constants.push_back(temp);
    temp.name = iGTE; temp.signature = reltype→clone(); constants.push_back(temp);
    temp.name = iLT; temp.signature = reltype→clone(); constants.push_back(temp);
    temp.name = iLTE; temp.signature = reltype→clone(); constants.push_back(temp);
```

Uses `iGT` 145, `iGTE` 145, `iLT` 145, and `iLTE` 145.

Comment 5.0.47. The following constants are for disruptive operations, that is, operations that changes persistent objects.

152c ⟨initialise constants::disruptive operations 152c⟩≡

```
    temp.name = iSucceeded; temp.signature = new type("Success");
    constants.push_back(temp);
    temp.name = iFailed; temp.signature = new type("Success");
    constants.push_back(temp);
    temp.name = iAssign;
    temp.signature = new type_abs(a→clone(),
                                  new type_abs(a→clone(), new type("Success")));
    constants.push_back(temp);
```

Uses `iAssign` 145, `iFailed` 145, and `iSucceeded` 145.

153a ⟨constants and their signatures 151⟩+≡

```
void insert_constant(int name, type ∗ sig) {
        assert(name > 0);
        for (unint i=0; i≠constants.size(); i++)
                if (constants[i].name ≡ name)
                        { ⟨insert constant:error message 153b⟩ return; }

        constant_sig temp; temp.name = name; temp.signature = sig;
        constants.push_back(temp);
}
```

Defines:
`insert_constant`, used in chunk 153e.

153b ⟨insert constant:error message 153b⟩≡

```
int osel = getSelector(); setSelector(STDERR);
cerr ≪ "The constant "≪ getString(name) ≪
        " has been defined before with type " ≪
        constants[i].signature→getName() ≪ ".\nInstruction ignored.\n";
setSelector(osel);
```

Uses `getSelector` 164 165, `getString` 147, and `setSelector` 164 165.

153c ⟨constants and their signatures 151⟩+≡

```
type ∗ get_signature(int name) {
        for (unint i=0; i≠constants.size(); i++)
                if (constants[i].name ≡ name) return constants[i].signature;
        cerr ≪ "Unknown constant: " ≪ getString(name) ≪ endl;
        // assert(false);
        return NULL;
}
```

Defines:
`get_signature`, used in chunks 23a and 153e.
Uses `getString` 147.

153d ⟨constants and their signatures 151⟩+≡

```
void cleanup_constants() {
        cerr ≪ "Cleaning up constants...";
        for (unint i=0; i≠constants.size(); i++)
                delete_type(constants[i].signature);
        cerr ≪ "Done.\n";
}
```

Defines:
`cleanup_constants`, used in chunk 153e.
Uses `delete_type` 10a 10b.

153e ⟨global:external functions 148b⟩+≡

```
extern void initialise_constants();
extern void insert_constant(int name, type ∗ sig);
extern type ∗ get_signature(int name);
extern void cleanup_constants();
```

Uses `cleanup_constants` 153d, `get_signature` 153c, `initialise_constants` 151, and `insert_constant` 153a.

Comment 5.0.48. Information about function symbols (collected during parsing) are stored in a hash table for quick and easy access. We now describe this function symbol table.

154a ⟨function symbol table 154a⟩≡

```
struct fEntry {
        int name;
        int minEffectArity;
        int maxEffectArity;
        fEntry(int n, int min, int max) {
                name = n; minEffectArity = min; maxEffectArity = max; }
};

#define TABLESIZE 501
static vector<fEntry> func_info[TABLESIZE];
```

Comment 5.0.49. Clearly, we want a hash function that can be computed efficiently. Looking at the first and last characters in the function name seemed a reasonable idea. (Looking at every character seemed expensive, but there is probably not much in it.) We need to add `size` to make sure functions that begin and end with the same characters are hashed to different indices with high probability.

154b ⟨function symbol table 154a⟩+≡

```
static int eshash(int name) {
        // int size = name.size();
        // int ret = name[0] * name[size-1] - (name[0] + name[size-1]) + size;
        // ret = ret % TABLESIZE;
        // return ret;
        return name % TABLESIZE;
}
```

Defines:
  `hash`, never used.

Comment 5.0.50. We can probably have a scheme whereby we try out different hash functions at run time and decide on one that induces the best distribution of functions in the table.

Comment 5.0.51. Here we need to initialise information for functions that are implemented inside the code.

154c ⟨function symbol table 154a⟩+≡

```
void initFuncTable() {
        insert_ftable(iAdd, 2); insert_ftable(iSub, 2);
        insert_ftable(iMax, 2); insert_ftable(iMin, 2);
        insert_ftable(iMul, 2); insert_ftable(iDiv, 2);
        insert_ftable(iMod, 2); insert_ftable(iAtan2, 2);
        insert_ftable(iSin, 1); insert_ftable(iCos, 1);
        insert_ftable(iSqrt, 1); insert_ftable(iExp, 1);
        insert_ftable(iLT, 2); insert_ftable(iLTE, 2);
        insert_ftable(iGT, 2); insert_ftable(iGTE, 2);
        insert_ftable(iAssign, 2); insert_ftable(iEqual, 2);
        insert_ftable(iNEqual, 2);
        insert_ftable(iPoint, 1);
}
```

Defines:
  `initFuncTable`, used in chunk 156b.
Uses `iAdd` 145, `iAssign` 145, `iAtan2` 145, `iCos` 145, `iDiv` 145, `iEqual` 145, `iExp` 145, `iGT` 145, `iGTE` 145, `iLT` 145, `iLTE` 145, `iMax` 145, `iMin` 145, `iMod` 145, `iMul` 145, `iNEqual` 145, `iPoint` 145, `iSin` 145, `iSqrt` 145, `iSub` 145, and `insert_ftable` 155a.

Comment 5.0.52. Basic insertion is okay. We first check whether `func` is already present before inserting.

155a ⟨function symbol table 154a⟩+≡

```
void insert_ftable(int func, int earity) {
        int index = eshash(func);
        int size = func_info[index].size();
        for (int i=0; i≠size; i++)
                if (func_info[index][i].name ≡ func) {
                        if (earity < func_info[index][i].minEffectArity)
                                func_info[index][i].minEffectArity = earity;
                        else if (earity > func_info[index][i].maxEffectArity)
                                func_info[index][i].maxEffectArity = earity;
                        return;
                }
        fEntry f(func, earity, earity);
        func_info[index].push_back(f);
        // print_ftable();
}
```

Defines:
  `insert_ftable`, used in chunks 154c and 156b.
Uses `print_ftable` 156a.

155b ⟨function symbol table 154a⟩+≡

```
pair<int,int> getFuncEArity(int func) {
        assert(func > 0);
        pair<int,int> ret(-1,-1);
        int index = eshash(func);
        int size = func_info[index].size();
        for (int i=0; i≠size; i++)
                if (func ≡ func_info[index][i].name) {
                        ret.first = func_info[index][i].minEffectArity;
                        ret.second = func_info[index][i].maxEffectArity;
                        return ret;
                }
        cerr ≪ "Error: Function " ≪ getString(func) ≪ " unknown. "
                "Effective arity could not be determined.\n";
        // assert(false);
        return ret;
}
```

Defines:
  `getFuncEArity`, used in chunks 82e, 83e, and 156b.
Uses `getString` 147.

156a ⟨function symbol table 154a⟩+≡

```
void print_ftable() {
        for (int j=0; j≠TABLESIZE; j++) {
                cout ≪ j ≪ ": ";
                int size = func_info[j].size();
                for (int i=0; i≠size; i++)
                        cout ≪ "(" ≪ func_info[j][i].name ≪ " "
                            ≪ func_info[j][i].minEffectArity ≪ " "
                            ≪ func_info[j][i].maxEffectArity ≪ ")\t";
                cout ≪ endl;
        }
}
```

Defines:
`print_ftable`, used in chunks 155a and 156b.

156b ⟨global:external functions 148b⟩+≡

```
/* function symbol table */
extern void initFuncTable();
extern void insert_ftable(int func, int earity);
extern pair<int,int> getFuncEArity(int func);
extern void print_ftable();
```

Uses `getFuncEArity` 155b, `initFuncTable` 154c, `insert_ftable` 155a, and `print_ftable` 156a.

156c ⟨nonrigid constants 156c⟩≡

```
set<int> nonrigid_constants;

void insert_nonrigid_constant(int name) { nonrigid_constants.insert(name); }
bool is_rigid_constant(int name) {
        return (nonrigid_constants.find(name) ≡ nonrigid_constants.end());
}
```

Uses `insert` 30e.

156d ⟨global:external functions 148b⟩+≡

```
extern void insert_nonrigid_constant(int name);
extern bool is_rigid_constant(int name);
```

Comment 5.0.53. This facility is used to provide mappings from type names to type objects. The initial assignment was performed in the parser.

157a ⟨type name to type objects mapping 157a⟩≡

```
#include <map>
static map<string, pair<int, type ∗> > type_fac;

void insert_type(const string & tname, int x, type ∗ tp) {
        assert(type_fac.find(tname) ≡ type_fac.end());
        pair<int, type ∗> temp(x, tp);
        type_fac[tname] = temp;
}

pair<int, type ∗> get_type(const string & tname) {
        map<string, pair<int,type ∗> >::iterator p = type_fac.find(tname);
        if (p ≡ type_fac.end()) { pair<int,type ∗> ret(-5,NULL); return ret; }
        return p→second;
}

void cleanup_synonyms() {
        cerr ≪ "Cleaning up type synonyms...";
        map<string, pair<int, type ∗> >::iterator p = type_fac.begin();
        while (p ≠ type_fac.end()) { delete_type(p→second.second); p++; }
        cerr ≪ "Done.\n";
}
```

Defines:
`cleanup_signatures`, never used.
`get_type`, used in chunk 157b.
`insert_type`, used in chunk 157b.
Uses `delete_type` 10a 10b.

157b ⟨global:external functions 148b⟩+≡

```
extern void insert_type(const string & tname, int x, type ∗ tp);
extern pair<int, type ∗> get_type(const string & tname);
extern void cleanup_synonyms();
```

Uses `get_type` 157a and `insert_type` 157a.

Comment 5.0.54. We next look at how statements are stored in the system. We use a vector of linked-lists of statements, indexed by the leftmost function symbol on the LHS of each statement. This allows us to jump straight to the relevant statements in constant time when doing pattern matching.

158a ⟨statements insertion and printing 158a⟩≡

```
void insert_statement(statementType ∗ st) {
        assert(st→anchor ≥ 0);
        int gsize = grouped_statements.size();
        /∗ grow vector if it is not big enough ∗/
        if (st→anchor > gsize-1)
                for (int i=0; i≠st→anchor+1; i++)
                        grouped_statements.push_back(NULL);
        assert(st→anchor < (int)grouped_statements.size());
        /∗ insert statement ∗/
        if (grouped_statements[st→anchor] ≡ NULL) {
                grouped_statements[st→anchor] = st;
                return;
        }
        statementType ∗ p = grouped_statements[st→anchor];
        while (p→next ≠ NULL) p = p→next;
        p→next = st;
}
```

Defines:
`insert_statement`, used in chunk 158c.
Uses `insert` 30e.

158b ⟨statements insertion and printing 158a⟩+≡

```
void print_grouped_statements() {
        setSelector(STDOUT);
        for (int i=0; i≠(int)grouped_statements.size(); i++) {
                if (grouped_statements[i] ≡ NULL) continue;
                ioprint("*****\n");
                grouped_statements[i]→print();
                ioprint("---\n");
        }
}
```

Defines:
`print_grouped_statements`, used in chunk 158c.
Uses `ioprint` 164 165 and `setSelector` 164 165.

158c ⟨global:external functions 148b⟩+≡

```
extern void insert_statement(statementType ∗ st);
extern void print_grouped_statements();
```

Uses `insert_statement` 158a and `print_grouped_statements` 158b.

Comment 5.0.55. Here are some functions for checking container membership.

159a ⟨misc functions 159a⟩≡

```
bool inVector(int x, vector<int> & v) {
        vector<int>::iterator p = find(v.begin(), v.end(), x);
        return (p ≠ v.end());
}
bool subset(vector<int> v1, vector<int> v2) {
        int size = v1.size();
        for (int i=0; i≠size; i++)
                if (¬inVector(v1[i], v2)) return false;
        return true;
}
```

Defines:
`inVector`, used in chunks 48a and 159b.
`subset`, used in chunk 159b.

159b ⟨global:external functions 148b⟩+≡

```
bool inVector(int x, vector<int> & v);
bool subset(vector<int> v1, vector<int> v2);
```

Uses `inVector` 159a and `subset` 159a.

Comment 5.0.56. The following implements uniform sampling from a collection of terms represented in a (non-empty) list (the input argument). We have to go through the list first to find out the size of the collection. If the size is given as an argument, we can just flip a coin and go straight to the desired term.

159c ⟨misc functions 159a⟩+≡

```
vector<term *> usamplingset;
pair<term *, float> uniformSampling(term * items) {
        usamplingset.clear();
        assert(items→isApp() ∧ items→lc()→lc()→isD(iHash));
        while (¬items→isD(iEmptyList)) {
                usamplingset.push_back(items→lc()→rc());
                items = items→rc();
        }
        int ssize = usamplingset.size();
        int i = random() % ssize;
        pair<term *, float> ret(usamplingset[i], 1.0 ÷ ssize);
        return ret;
}
```

Uses `iEmptyList` 145, `iHash` 145, `isApp` 30a, `isD` 30a, `lc` 30e, and `rc` 30e.

Comment 5.0.57. The next function implements sampling from a categorical distribution. We assume the input argument has the form

$$[(t_1, n_1), \ldots, (t_k, n_k),$$

where each t_i is a term and n_i its probability.

160 ⟨misc functions 159a⟩+≡

```
vector<pair<term ∗,float> > msamplingset;
pair<term ∗,float> categoricalSampling(term ∗ arg) {
        msamplingset.clear();
        float counter = 0;
        assert(arg→isApp() ∧ arg→lc()→lc()→isD(iHash));
        while (¬arg→isD(iEmptyList)) {
                pair<term ∗,float> element;
                element.first = arg→lc()→rc()→fields[0];
                counter += arg→lc()→rc()→fields[1]→numf;
                element.second = counter;
                msamplingset.push_ back(element);
                arg = arg→rc();
        }
        assert(msamplingset.size() > 0); assert(counter ≡ 1);
        int r = random() % 100;
        unint i = 0;
        while (r÷100.0 > msamplingset[i].second) { i++; }
        float prob;
        if (i ≡ 0) prob = msamplingset[i].second;
        else prob = msamplingset[i].second - msamplingset[i-1].second;
        pair<term ∗,float> ret(msamplingset[i].first,prob);
        return ret;
}
```

Uses `iEmptyList` 145, `iHash` 145, `isApp` 30a, `isD` 30a, `lc` 30e, and `rc` 30e.

Comment 5.0.58. The next function implements sampling from a normal distribution. We use the Box-Muller-Marsaglia polar method described in [Knu97].

161 ⟨misc functions 159a⟩+≡

```
float gaussDens(float mu, float sigma, float x) {
        assert(sigma > 0);
        float s1 = 1.0 ÷ (sigma ∗ sqrt( 2 ∗ 3.14159));
        float s2 = (x - mu) ∗ (x - mu) ÷ (2 ∗ sigma ∗ sigma);
        float ret = s1 ∗ exp ( -1.0 ∗ s2);
        return ret;
}

pair<term ∗,float> ret;
pair<term ∗,float> gaussianSampling(term ∗ m, term ∗ s) {
        assert(m→isfloat ∧ s→isfloat);
        float u1, u2, v1, v2, S;
        do {
                u1 = (random() % 100) ÷ 100.0;
                u2 = (random() % 100) ÷ 100.0;
                v1 = 2∗u1 - 1;
                v2 = 2∗u2 - 1;
                S = u1∗u1 + u2∗u2;
        } while (S ≥ 1.0);
        float x1; // float x2;
        if (S ≡ 0) { x1 = 0; /∗ x2 = 0; ∗/ }
        else { x1 = v1 ∗ sqrt( -2∗log(S)÷S );
                /∗ x2 = v2 * sqrt( -2*log(S)/S ); ∗/ }

        ret.first = new_term_float(m→numf + s→numf ∗ x1);
        ret.second = gaussDens(m→numf, s→numf, ret.first→numf);

        return ret;
}
```

Uses `new_term_float` 40a.

```
162    ⟨misc functions 159a⟩+≡
         ÷∗ vector<pair<term ∗,float> > dgsamplingset;
         pair<term ∗,float> dgaussianSampling(term ∗ m, term ∗ s) {
                 assert(m→isfloat ∧ s→isfloat);
                 dgsamplingset.clear();

                 float mu = m→numf; float sigma = s→numf;
                 pair<term ∗,float> cent;
                 cent.first = new_term_float(mu);
                 cent.second = gaussDens(mu,sigma,mu);

                 float total = 0;
                 for (int i=0; i≠6; i++) {
                         pair<term ∗,float> cent;
                         cent.first = new_term_float(mu + i∗sigma);
                         cent.second = gaussDens(mu,sigma,mu + i∗sigma);
                         dgsamplingset.push_back(cent);
                         total += cent.second;

                         if (i≡0) continue;

                         cent.first = new_term_float(mu - i∗sigma);
                         cent.second = gaussDens(mu,sigma,mu + i∗sigma);
                         dgsamplingset.push_back(cent);
                         total += cent.second;
                 }

                 float offset = 0;
                 for (unint j=0; j≠dgsamplingset.size(); j++) {
                         dgsamplingset[j].second =
                                 offset + dgsamplingset[j].second ÷ total;
                         offset = dgsamplingset[j].second;
                 }

                 for (unint i=0; i≠dgsamplingset.size(); i++) {
                         setSelector(STDOUT);
                         dgsamplingset[i].first→print();
                         cout ≪ "," ≪ dgsamplingset[i].second ≪ "  ";
                         // revertSelector();
                 } cout ≪ endl;


                 int r = random() % 1000;
                 unint i = 0;
                 while (r÷1000.0 > dgsamplingset[i].second) { i++; }
                 float prob;
                 if (i ≡ 0) prob = dgsamplingset[i].second;
                 else prob = dgsamplingset[i].second - dgsamplingset[i-1].second;
                 pair<term ∗,float> ret(dgsamplingset[i].first, prob);
                 return ret;
         } ∗÷
```

Uses `new_term_float` 40a and `setSelector` 164 165.

Comment 5.0.59. This is the public function called to invoke the appropriate sampling routine.

163a ⟨misc functions 159a⟩+≡

```
pair<term ∗, float> sample(term ∗ density) {
        assert(density→isApp());
        pair<term ∗, float> ret(NULL,0.0);
        term ∗ distr = density→spineTip();
        if (distr→isF(iUniform))
                ret = uniformSampling(density→rc());
        else if (distr→isF(iCategorical))
                ret = categoricalSampling(density→rc());
        else if (distr→isF(iGaussian)) {
                float m = density→lc()→rc()→numf;
                float s = density→rc()→numf;
                while (true) {
                      ret = gaussianSampling(density→lc()→rc(), density→rc());
                      if (fabs(ret.first→numf - m) ≤ 2∗s) break;
                      ret.first→freememory();
                }
        }
        // else if (distr->isF(iDGaussian))
        //       ret = dgaussianSampling(density->lc()->rc(), density->rc());
        else if (distr→isF(iPoint)) {
                ret.first = density→rc()→clone(); ret.second = 1.0; }
        return ret;
}
```

Uses `iCategorical` 145, `iDGaussian` 145, `iGaussian` 145, `iPoint` 145, `iUniform` 145, `isApp` 30a, `isF` 30a, `lc` 30e, `rc` 30e, and `spineTip` 32e.

163b ⟨global:external functions 148b⟩+≡

```
pair<term ∗, float> sample(term ∗ density);
```

Comment 5.0.60. The following are two functions for converting numbers to their string representations.

163c ⟨global:external functions 148b⟩+≡

```
#include <sstream>
inline string numtostr(const int i) { stringstream s; s ≪ i; return s.str(); }
inline string numtostr(const double i) { stringstream s; s ≪ i; return s.str(); }
```

5.1 IO Facilities

Comment 5.1.1. Silent printing is a useful trick I learned from [Knu86].

164 ⟨io.h 164⟩≡

```
#ifndef _IO_H_
#define _IO_H_

#include <string>
#include <iostream>
#include <fstream>
#include <stdio.h>
using namespace std;

#define STDOUT 1
#define STDERR 2
#define SILENT 3
#define EXFILE 4
#define PIPE   5

void initPipe();
void closePipe();
void setSelector(FILE ∗ in);
void setSelector(int x);
int getSelector();

void ioprint(const string & x);
void ioprint(int x);
void ioprint(long long int x);
void ioprint(double x);
void ioprint(char x);
void ioprintln(const string & x);
void ioprintln(int x);
void ioprintln(long long int x);
void ioprintln(double x);
void ioprintln(char x);
void ioprintln();

#endif
```

Defines:

`getSelector`, used in chunks 25–27, 35, 37a, 89b, 91–93, 103d, 110a, and 153b.
`ioprint`, used in chunks 25–27, 35–37, 42a, 63c, 89b, 91–93, 101–103, 107a, 109d, 110a, 141, 142, and 158b.
`ioprintln`, used in chunks 25–27, 37c, 42a, 63c, 85a, 89b, 91–93, 102a, 103d, 107a, and 139a.
`setSelector`, used in chunks 25–27, 42a, 63c, 84b, 85a, 89b, 91–93, 101–103, 109d, 142, 153b, 158b, and 162.

165 ⟨io.cc 165⟩≡

```
  #include "io.h"

  ofstream bcpipe;
  FILE ∗ iofile;
  static int selector;
  ofstream logfile("log.complete");

  void initPipe() { bcpipe.open("pipe1"); bcpipe.setf(ios::fixed); }
  void closePipe() { bcpipe.close(); }
  void setSelector(FILE ∗ in) { iofile = in; selector = EXFILE; }
  void setSelector(int x) { selector = x; }
  int getSelector() { return selector; }

  void ioprint(const string & x) {
          ⟨io::common print command 166a⟩ ⟨io::file 166c⟩ }
  void ioprint(int x) {
          ⟨io::common print command 166a⟩ ⟨io::file 2 166d⟩ }
  void ioprint(long long int x) {
          ⟨io::common print command 166a⟩ ⟨io::file 2a 166e⟩ }
  void ioprint(double x) {
          // cout.setf(ios::fixed, ios::floatfield);
          cout.setf(ios::showpoint);
          ⟨io::common print command 166a⟩ ⟨io::file 3 166f⟩ }
  void ioprint(char x) {
          ⟨io::common print command 166a⟩ ⟨io::file 4 166g⟩ }
  void ioprintln(const string & x) {
          ⟨io::common print command ln 166b⟩ ⟨io::file 166c⟩ }
  void ioprintln(int x) {
          ⟨io::common print command ln 166b⟩ ⟨io::file 2 166d⟩ }
  void ioprintln(long long int x) {
          ⟨io::common print command ln 166b⟩ ⟨io::file 2a 166e⟩ }
  void ioprintln(double x) {
          //cout.setf(ios::fixed, ios::floatfield);
          cout.setf(ios::showpoint);
          ⟨io::common print command ln 166b⟩ ⟨io::file 3 166f⟩ }
  void ioprintln(char x) {
          ⟨io::common print command ln 166b⟩ ⟨io::file 4 166g⟩ }
  void ioprintln() {
  #ifdef DEBUG
          logfile ≪ endl;
  #endif
          if (selector ≡ SILENT) return;
          else if (selector ≡ STDOUT) cout ≪ endl;
          else if (selector ≡ EXFILE) fprintf(iofile, "\n");
          else if (selector ≡ PIPE) bcpipe ≪ endl;
          else cerr ≪ endl;
  }
```

Defines:
`getSelector`, used in chunks 25–27, 35, 37a, 89b, 91–93, 103d, 110a, and 153b.
`ioprint`, used in chunks 25–27, 35–37, 42a, 63c, 89b, 91–93, 101–103, 107a, 109d, 110a, 141, 142, and 158b.
`ioprintln`, used in chunks 25–27, 37c, 42a, 63c, 85a, 89b, 91–93, 102a, 103d, 107a, and 139a.
`setSelector`, used in chunks 25–27, 42a, 63c, 84b, 85a, 89b, 91–93, 101–103, 109d, 142, 153b, 158b, and 162.

166a ⟨io::common print command 166a⟩≡

```
// #define DEBUG
#ifdef DEBUG
logfile ≪ x;
#endif
if (selector ≡ SILENT) return;
if (selector ≡ STDOUT) cout ≪ x;
if (selector ≡ STDERR) cerr ≪ x;
else if (selector ≡ PIPE) bcpipe ≪ x;
```

166b ⟨io::common print command ln 166b⟩≡

```
#ifdef DEBUG
logfile ≪ x ≪ endl;
#endif
if (selector ≡ SILENT) return;
if (selector ≡ STDOUT) cout ≪ x ≪ endl;
if (selector ≡ STDERR) cerr ≪ x ≪ endl;
else if (selector ≡ PIPE) bcpipe ≪ x ≪ endl;
```

166c ⟨io::file 166c⟩≡

```
if (selector ≡ EXFILE) fprintf(iofile, x.c_str());
```

166d ⟨io::file 2 166d⟩≡

```
if (selector ≡ EXFILE) fprintf(iofile, "%d", x);
```

166e ⟨io::file 2a 166e⟩≡

```
if (selector ≡ EXFILE) fprintf(iofile, "%lld", x);
```

166f ⟨io::file 3 166f⟩≡

```
if (selector ≡ EXFILE) fprintf(iofile, "%f", x);
```

166g ⟨io::file 4 166g⟩≡

```
if (selector ≡ EXFILE) fprintf(iofile, "%c", x);
```

Chapter 6

System Modules

6.1 The Booleans Module

167 ⟨booleans.es 167⟩≡

```
- Equality and Disequality

remove : (a -> Bool) -> (a -> b) -> (a -> b) ;
(remove s \x.d_SV/CONST/) = \x.d_SV ;
     - where d_SV is a default term (FIX THIS)

(remove s \x.if u_SV then v else w_SV) =
               \x.if (&& u_SV (not (s x))) then v else ((remove s \x.w_SV) x) ;

- = : a -> a -> Bool ;
import sets.es ;

(= \x.u_SV \y.v_SV) =
        (&& (less \x.u_SV \y.v_SV) (less \y.v_SV \x.u_SV)) ;

less : (a -> b) -> (a -> b) -> Bool ;
(less \x.d_SV/CONST/ z) = True ;
     - where d_SV is a default term (FIX THIS)
(less \x.if u_SV then v else w_SV z) =
             \forall x.(&& (implies u_SV (= v (z x)))
                           (less (remove \x.u_SV \x.w_SV) z)) ;


ite : (Bool * a * a) -> a ;
if True then u else v = u ;
if False then u else v = v ;
if x then x_SV else y_SV/EQUAL,x_SV/ = x_SV ;

if if x then y else w then True else z = if x then y else (glueite w z) ;

glueite : b -> a -> b ;
(glueite False w) = w ;
(glueite 0.0 w) = w ;
(glueite if x then y else z w) = if x then y else (glueite z w) ; Eager ;
  - this Eagerness is necessary to ensure correctness
```

```
- if (if u then True else False) then True else v = if u then True else v ;
- if u_SV then False else v_SV/EQUAL,u_SV/ = False ;
- if u_SV then True else (if v_SV/EQUAL,u_SV/ then True else w_SV =
-                   if v_SV then True else w_SV ;

- this something useful to convert if-then-else's to ||'s and &&'s
- if u then v else w = (|| (&& u v) (&& (not u) w)) ;

- (if x then y else z w) = if x then (y w) else (z w) ; LastResort ;
- \forall w.\forall x.\forall y.\forall z.(= (w if x then y else z) if x then (w y) else (w z)) ;

&& : Bool -> Bool -> Bool ;
(&& True x) = x ;
(&& x True) = x ;
(&& False x) = False ;
(&& x False) = False ;
- Do we really need these? Apparently permute need them.
(&& (|| x y) z) = (|| (&& x z) (&& y z)) ; LastResort ;
(&& x (|| y z)) = (|| (&& x y) (&& x z)) ; LastResort ;
(&& if u then v else w t) = if (&& u t) then v else (&& w t) ; LastResort ;
(&& t if u then v else w) = if (&& t u) then v else (&& t w) ; LastResort ;

- The following are specialized versions of the two rules above.
- In computations, I find that the two rules given above tend to work
- (really) badly when used in conjunction with Escher's leftmost outermost
- reduction order. A more in-depth analysis of this phenomenon is called for.
(&& if (= z u) then v else w t) =
                  if (&& (= z u) t) then v else (&& w t) ; LastResort ;
(&& t if (= x u) then v else w) =
                  if (&& t (= x u)) then v else (&& t w) ; LastResort ;

|| : Bool -> Bool -> Bool ;
(|| True x) = True ;
(|| x True) = True ;
(|| False x) = x ;
(|| x False) = x ;

(|| if u then True else w t) = if u then True else (|| w t) ; LastResort ;
(|| if u then False else w t) = (|| (&& (not u) w) t) ; LastResort ;

(|| t if u then True else w) = if u then True else (|| t w) ; LastResort ;
(|| t if u then False else w) = (|| t (&& (not u) w)) ; LastResort ;

- this is needed when using rmdup2
(|| (= x_SV u_SV) (= y_SV/EQUAL,x_SV/ v_SV/EQUAL,u_SV/)) = (= x_SV u_SV) ;

not : Bool -> Bool ;
(not False) = True ;
(not True) = False ;
(not (not x)) = x ;
(not (&& x y)) = (|| (not x) (not y)) ; LastResort ;
(not (|| x y)) = (&& (not x) (not y)) ; LastResort ;
(not if u then v else w) = if u then (not v) else (not w) ; LastResort ;
```

```
sigma : (a -> Bool) -> Bool ;
⟨existential statements 170⟩

pi : (a -> Bool) -> Bool ;
⟨universal statements 171⟩

implies : Bool -> Bool -> Bool ;
(implies True x) = x ;    - these are needed by queries 8 and 9 in
(implies False x) = True ; - the database example
- (implies p q) = (|| (not p) q) ; LastResort ; - this affects pi, bad.

/= : a -> a -> Bool ;
(/= x y) = (not (= x y)) ;

comp : (a -> b) -> (b -> c) -> a -> c ;
- (comp p1 p2) = \x.(p2 (p1 x)) ;
(comp p1 p2 x) = (p2 (p1 x)) ;

proj1 : (a * b) -> a ;
(proj1 (t1,t2)) = t1 ;

proj2 : (a * b) -> b ;
(proj2 (t1,t2)) = t2 ;

identity : a -> a ;
(identity x) = x ; LastResort ;

- These are used by the theorem prover
TpTag : ProveStatus -> Bool -> Bool ;
DontKnow : ProveStatus ;
```

Comment 6.1.1. The rules for Σ as presented in [Llo03] are as follows:

$$\exists x_1.\cdots\exists x_n.\top = \top \tag{6.1}$$

$$\exists x_1.\cdots\exists x_n.\bot = \bot \tag{6.2}$$

$$\exists x_1.\cdots\exists x_n.(\mathbf{x} \wedge (x_i = \mathbf{u}) \wedge \mathbf{y}) = \exists x_1.\cdots\exists x_{i-1}.\exists x_{i+1}.\cdots\exists x_n.(\mathbf{x}\{x_i/\mathbf{u}\} \wedge \mathbf{y}\{x_i/\mathbf{u}\}) \tag{6.3}$$

$$\exists x_1.\cdots\exists x_n.(\mathbf{u} \vee \mathbf{v}) = (\exists x_1.\cdots\exists x_n.\mathbf{u}) \vee (\exists x_1.\cdots\exists x_n.\mathbf{v}) \tag{6.4}$$

$$\exists x_1.\cdots\exists x_n.(\mathit{if}\ \mathbf{u}\ \mathit{then}\ \top\ \mathit{else}\ \mathbf{v}) = (\mathit{if}\ \exists x_1.\cdots\exists x_n.\mathbf{u}\ \mathit{then}\ \top\ \mathit{else}\ \exists x_1.\cdots\exists x_n.\mathbf{v} \tag{6.5}$$

$$\exists x_1.\cdots\exists x_n.(\mathit{if}\ \mathbf{u}\ \mathit{then}\ \top\ \mathit{else}\ \mathbf{v}) = \exists x_1.\cdots\exists x_n.(\neg\mathbf{u} \vee \mathbf{v}) \tag{6.6}$$

Statements 6.1 to 6.3 are implemented in the internal simplification routines. We now look at how the remaining statements are implemented in the booleans module. The expression $\exists x_1 \cdots \exists x_n$ in the heads is a bit worrying. How can we capture that in a finite number of statements? The answer is very simple: replace $\exists x_1 \cdots \exists x_n$ with $\exists x$ in Statements 6.4 to 6.6. Here are some questions for the reader. Why can we do that? What is the cost of doing that? Why do we bother with the case of $\exists x_1 \cdots \exists x_3$ for Statements 6.4 and 6.5 but not Statement 6.6? Is the number 3 special or can (should?) it be so some other number?

170 ⟨existential statements 170⟩≡

```
(sigma \x.(|| u_SV v_SV)) = (|| (sigma \x.u_SV) (sigma \x.v_SV)) ;
(sigma \x1.(sigma \x2.(sigma \x3.(|| u_SV v_SV)) ) ) =
        (|| (sigma \x1.(sigma \x2.(sigma \x3.u_SV)))
            (sigma \x1.(sigma \x2.(sigma \x3.v_SV)))) ;

(sigma \x.if u_SV then True else v_SV) =
                if \exists x.u_SV then True else \exists x.v_SV ;
(sigma \x1.(sigma \x2.(sigma \x3.if u_SV then True else v_SV))) =
       if (sigma \x1.(sigma \x2.(sigma \x3.u_SV))) then True
       else (sigma \x1.(sigma \x2.(sigma \x3.v_SV))) ;

(sigma \x.if u_SV then False else v_SV) = (sigma \x.(|| (not u_SV) v_SV)) ;
(sigma \x2.(sigma \x1.(sigma \x.if u_SV then False else v_SV))) =
       (sigma \x2.(sigma \x1.(sigma \x.(|| (not u_SV) v_SV)))) ;
(sigma \x.if u_SV then v_SV else w_SV) =
            if (sigma \x.(&& u_SV v_SV)) then True
            else (sigma \x.(&& (not u_SV) w_SV)) ; LastResort ;

- \exists x.if u_SV then v_SV else w_SV =
-  if \exists x.(&& u_SV v_SV) then True else \exists x.(&& (not u_SV) w_SV) ;
```

Comment 6.1.2. The rules for Π as stated in [Llo03] are as follows:

$$\forall x_1.\cdots\forall x_n.(\bot \rightarrow \mathbf{u}) = \top \tag{6.7}$$

$$\begin{aligned}\forall x_1.\cdots\forall x_n.(\mathbf{x} \wedge (x_i = \mathbf{u}) \wedge \mathbf{y} \rightarrow \mathbf{v}) &= \\ &\forall x_1.\cdots\forall x_{i-1}.\forall x_{i+1}.\cdots\forall x_n.((\mathbf{x} \wedge \mathbf{y} \rightarrow \mathbf{v})\{x_i/\mathbf{u}\})\end{aligned} \tag{6.8}$$

$$\forall x_1.\cdots\forall x_n.(\mathbf{u} \vee \mathbf{v} \rightarrow \mathbf{t}) = (\forall x_1.\cdots\forall x_n.(\mathbf{u} \rightarrow \mathbf{t})) \wedge (\forall x_1.\cdots\forall x_n.(\mathbf{v} \rightarrow \mathbf{t})) \tag{6.9}$$

$$\forall x_1.\cdots\forall x_n.((\mathit{ite}\ \mathbf{u}\ \top\ \mathbf{v}) \rightarrow \mathbf{t}) = (\forall x_1.\cdots\forall x_n.(\mathbf{u} \rightarrow \mathbf{t})) \wedge (\forall x_1.\cdots\forall x_n.(\mathbf{v} \rightarrow \mathbf{t})) \tag{6.10}$$

$$\forall x_1.\cdots\forall x_n.((\mathit{ite}\ \mathbf{u}\ \bot\ \mathbf{v}) \rightarrow \mathbf{t}) = \forall x_1.\cdots\forall x_n.(\neg\mathbf{u} \wedge \mathbf{v} \rightarrow \mathbf{t}) \tag{6.11}$$

Statements 6.7 and 6.1.2 are implemented as part of the internal simplification routine. Notice that the body of Statements 6.9 and 6.10 are identical. If we include the statement

$$\mathit{if}\ u\ \mathit{then}\ \top\ \mathit{else}\ v = u \vee v \tag{6.12}$$

as part of the *ite* rules, then we can get Statement 6.10 from Statement 6.9 via Statement 6.12. Proceeding in a similar fashion, we can use

$$\textit{if } u \textit{ then } \bot \textit{ else } v = \neg u \wedge v \tag{6.13}$$

to get rid of Statement 6.11 (and Statement 6.6). One thing with Statements 6.12 and 6.13 is that they force the a in the type of *ite* to be boolean. Can we do this simplification? If we want to retain the flexibility of handling sets represented both as nested *ite*s and disjunctions, the answer is unfortunately no. This is because Statement 6.12 will transform any set represented in *ite* form into its corresponding disjunctive form. This will, for example, affect the operation of *less*, which is only defined for sets represented in *ite* form.

So we cannot do that simplification. (We can probably still safely use Statement 6.13, but we will not try.) That means we have to find a way to represent Statements 6.9–6.11. The following rules together capture them finitely.

$$\forall x.(\mathbf{u} \vee \mathbf{v} \to \mathbf{t}) = \forall x.(\mathbf{u} \to \mathbf{t}) \wedge \forall x.(\mathbf{v} \to \mathbf{t}) \tag{6.14}$$

$$\forall x.((\textit{if } \mathbf{u} \textit{ then } \top \textit{ else } \mathbf{v}) \to \mathbf{t}) = \forall x.(\mathbf{u} \to \mathbf{t}) \wedge \forall x.(\mathbf{v} \to \mathbf{t}) \tag{6.15}$$

$$\forall x.(\forall y.\mathbf{u} \wedge \forall z.\mathbf{v}) = \forall x.\forall y.\mathbf{u} \wedge \forall x.\forall z.\mathbf{v} \tag{6.16}$$

$$\forall x.((\textit{if } \mathbf{u} \textit{ then } \bot \textit{ else } \mathbf{v}) \to \mathbf{t}) = \forall x.(\neg\mathbf{u} \wedge \mathbf{v} \to \mathbf{t}) \tag{6.17}$$

Question 6.1.3. Why do not we need the following counterpart

$$\forall x_1. \cdots \forall x_n.(\top \to \mathbf{u}) = \forall x_1. \cdots \forall x_n.\mathbf{u}$$

to Statement 6.7? I did see the need to put in $\top \to u = u$ in the module for some queries to work.

171 ⟨universal statements 171⟩≡

```
(pi \x.False) = False ;
(pi \x.True) = True ;
(pi \x1.(implies (|| u_SV v_SV) t_SV)) =
      (&& (pi \x1.(implies u_SV t_SV)) (pi \x1.(implies v_SV t_SV))) ;
(pi \x1.(implies if u_SV then True else v_SV t_SV)) =
      (&& (pi \x1.(implies u_SV t_SV)) (pi \x1.(implies v_SV t_SV))) ;

(pi \x.(&& (pi \x1.u_SV) (pi \x2.v_SV))) =
      (&& (pi \x.(pi \x1.u_SV)) (pi \x.(pi \x2.v_SV))) ;

(pi \x.(implies if u_SV then True else v_SV t_SV)) =
      (pi \x.(implies (&& (not u_SV) v_SV) t_SV)) ;

- if u then True else v = (|| u v)
- if u then False else v = (&& (not u) v)
```

6.2 The Numbers Module

172 ⟨numbers.es 172⟩≡

```
Infinity : a ;
NegInfinity : a ;

(add x_SV/CONST/ Infinity) = Infinity ;
(add Infinity x_SV/CONST/) = Infinity ;
(sub x_SV/CONST/ Infinity) = NegInfinity ;
(sub Infinity x_SV/CONST/) = Infinity ;

(min x_SV/CONST/ Infinity) = x_SV ;
(min Infinity x_SV/CONST/) = x_SV ;

(add x 0.0) = x;
(add 0.0 x) = x ;
(mul x 0.0) = 0.0 ;
(mul 0.0 x) = 0.0 ;

(div 0 x) = 0 ;
(div 0.0 x) = 0.0 ;
(mul 1 x) = x ;
(mul 1.0 x) = x ;

Power : (number1 * Int) -> number2 ;
power : (number1 * Int) -> number3 ;
(power (1,n)) = 1;
(power (m_SV/CONST/,0)) = 1 ;
(power (m_SV/CONST/,1)) = m_SV ;
(power (m_SV/CONST/,n_SV/CONST/)) = if (&& (<= m_SV 16) (< n_SV 16)) then
                                        (power2 (m_SV,n_SV))
                                    else (Power (m_SV,n_SV)) ;

power2 : (number1 * Int) -> number2 ;
(power2 (m_SV/CONST/,n_SV/CONST/)) = (mul m_SV (power (m_SV,(sub n_SV 1)))) ;

integer : number1 -> number2 -> number3 ;
(integer x y) = (sub (div x y) (remainder x y)) ;

remainder : number1 -> number2 -> number3 ;
(remainder x y) = (div (mod x y) y) ;

monus : number1 -> number2 -> number3 ;
(monus x_SV/CONST/ y_SV/CONST/) = (max 0 (sub x_SV y_SV)) ;

- this produces a loop
- (> (add u_SV/CONST/ (card v)) v_SV/CONST/) =
-       if (> u_SV v_SV) then True else (> (add u_SV (card v)) v_SV) ;

- > : number -> number -> Bool ;
- (> if u then v_SV/CONST/ else w_SV/CONST/ x_SV/CONST/) =
-                       if u then (> v_SV x_SV) else (> w_SV x_SV) ;

- >= : number -> number -> Bool ;
```

```
- (>= if u then v_SV/CONST/ else w_SV/CONST/ x_SV/CONST/) =
-                        if u then (>= v_SV x_SV) else (>= w_SV x_SV) ;

- < : number -> number -> Bool ;
- (< if u then v_SV/CONST/ else w_SV/CONST/ x_SV/CONST/) =
-                        if u then (< v_SV x_SV) else (< w_SV x_SV) ;

- <= : number -> number -> Bool ;
- (<= if u then v_SV/CONST/ else w_SV/CONST/ x_SV/CONST/) =
-                        if u then (<= v_SV x_SV) else (<= w_SV x_SV) ;

- (< x_SV/CONST/ Infinity) =
-             if (/= x_SV Infinity) then True else (< x_SV Infinity) ;

(< Infinity x_SV/CONST/) = False ;
(< x_SV/CONST/ Infinity) = True ;

abs : number -> number ;
(abs x_SV/CONST/) = if (>= x_SV 0) then x_SV else (add x_SV (mul -2 x_SV)) ;

fabs : number -> number ;
(fabs x_SV/CONST/) = if (>= x_SV 0.0) then x_SV else (add x_SV (mul -2.0 x_SV));

mChooseN : Int -> Int -> Int ;
(mChooseN m_SV/CONST/ 0) = 1 ;
(mChooseN m_SV/CONST/ n_SV/CONST/) =
                      (div (facl m_SV (sub m_SV n_SV)) (fac n_SV)) ;

facl : Int -> Int -> Int ;
(facl m_SV/CONST/ n_SV/CONST/) = if (> m_SV n_SV)
                                 then (mul m_SV (facl (sub m_SV 1) n_SV))
                                 else 1 ;

fac : Int -> Int ;
(fac 0) = 1 ;
(fac 1) = 1 ;
(fac m_SV/CONST/) = (mul m_SV (fac (sub m_SV 1))) ;
```

6.3 The List Module

174 ⟨lists.es 174⟩≡

```
- [] : (List a) ;
- # : a -> (List a) -> (List a) ;

import numbers.es ;

head : (List a) -> a ;
(head (# x y)) = x ;

tail : (List a) -> (List a) ;
(tail (# x y)) = y ;

last : (List a) -> a ;
(last (# x [])) = x;
(last (# x (# y z))) = (last (# y z)) ;

elem : Int -> (List a) -> a ;
(elem 1 (# x y)) = x ;
(elem z_SV/CONST/ (# x y)) = (elem (sub z_SV 1) y) ;

enumList : Int -> (List Int) ;
(enumList x_SV/CONST/) = (enumList2 x_SV x_SV) ;

enumList2 : Int -> Int -> (List Int) ;
(enumList2 0 x) = [] ;
(enumList2 x_SV/CONST/ y_SV/CONST/) =
          (# (add (sub y_SV x_SV) 1) (enumList2 (sub x_SV 1) y_SV)) ;

inList : a -> (List a) -> Bool ;
(inList x []) = False ;
(inList x (# y z)) = if (= x y) then True else (inList x z) ;

length : (List a) -> Int ;
(length []) = 0 ;
(length (# x y)) = (add 1 (length y)) ;

zip : (List a) -> (List b) -> (List (a * b)) ;
(zip [] []) = [] ;
(zip (# x1 y1) (# x2 y2)) = (# (x1,x2) (zip y1 y2)) ;

zipWith : a -> (List b) -> (List (a * b)) ;
(zipWith x []) = [] ;
(zipWith x (# y z)) = (# (x,y) (zipWith x z)) ;

concat : ((List a) * (List a)) -> (List a) ;
(concat ([],x)) = x ;
(concat ((# u x), y)) = (# u (concat (x, y))) ;

concat2 : (List a) -> (List a) -> (List a) ;
(concat2 [] x) = x ;
(concat2 (# u x) y) = (# u (concat2 x y)) ;
```

```
reverse : (List a) -> (List a) ;
(reverse []) = [] ;
(reverse (# x y)) = (concat ((reverse y),[x])) ;

append : ((List a) * (List a) * (List a)) -> Bool ;
(append (u,v,w)) =
     (|| (&& (= u []) (= v w))
         (sigma \r.
          (sigma \x.
           (sigma \y.(&& (&& (= u (# r x)) (= w (# r y)))
                         (append (x,v,y))))))) ;

permute : ((List a) * (List a)) -> Bool ;
(permute ([], x)) = (= x []) ;
(permute ((# x y), w)) =
    (sigma \u.(sigma \v.(sigma \z.
       (&& (= w (# u v)) (&& (delete (u,(# x y),z)) (permute (z,v))))))) ;

delete : (a * (List a) * (List a)) -> Bool ;
(delete (x,[],y)) = False ;
(delete (x,(# y z),w)) =
    (|| (&& (= x y) (= w z))
         (sigma \v.(&& (= w (# y v)) (delete (x,z,v))))) ;

sorted : (List a) -> Bool ;
(sorted []) = True ;
(sorted (# x y)) =
   if (= y []) then True
   else (sigma \u.(sigma \v.(&& (&& (= y (# u v)) (<= x u)) (sorted y)))) ;

isort : (List a) -> (List a) ;
(isort []) = [] ;
(isort (# x y)) = (ins x (isort y)) ;

ins : a -> (List a) -> (List a) ;
(ins x []) = (# x []) ;
(ins x (# y z)) = if (<= x y) then (# x (# y z)) else (# y (ins x z)) ;

isort2 : (a -> a -> Bool) -> (List a) -> (List a) ;
(isort2 p []) = [] ;
(isort2 p (# x y)) = (ins2 p x (isort2 p y)) ;

ins2 : (a -> a -> Bool) -> a -> (List a) -> (List a) ;
(ins2 p x []) = (# x []) ;
(ins2 p x (# y z)) = if (p x y) then (# x (# y z)) else (# y (ins2 p x z)) ;

fold : (a -> b -> b) -> b -> (List a) -> b ;
(fold m v []) = v ;
(fold m v (# x y)) = (m x (fold m v y)) ;

foldr : (a -> b -> b) -> b -> (List a) -> b;
(foldr m s []) = s ;
(foldr m s (# x y)) = (m x (foldr m s y)) ;
```

```
filter : (a -> Bool) -> (List a) -> (List a) ;
(filter p []) = [] ;
(filter p (# x y)) = if (p x) then (# x (filter p y)) else (filter p y) ;

map : (a -> b) -> (List a) -> (List b) ;
(map m []) = [] ;
(map m (# x [])) = (# (m x) []) ;
(map m (# x y)) = (# (m x) (map m y)) ;

rmduplicates : (List a) -> (List a) ;
(rmduplicates []) = [] ;
(rmduplicates (# x y)) = (# x (rmduplicates (removeListEle x y))) ;

removeListEle : a -> (List a) -> (List a) ;
(removeListEle x []) = [] ;
(removeListEle x (# y z)) = if (= x y) then (removeListEle x z)
                            else (# y (removeListEle x z)) ;

neg : (a -> Bool) -> a -> Bool ;
(neg p x) = (not (p x)) ;

qsort : (List a) -> (List a) ;
(qsort []) = [] ;
(qsort (# x y)) =
  (concat ((qsort (filter (neg (< x)) y)),
           (# x (qsort (filter (< x) y))))) ;

listExists : (a -> Bool) -> (List a) -> Bool ;
(listExists p []) = False ;
(listExists p (# x y)) = if (p x) then True else (listExists p y) ;

sublist : Int -> (List a) -> (List a) ;
(sublist n []) = [] ;
(sublist n (# x y)) = if (> n 0) then (# x (sublist (sub n 1) y)) else [] ;

isSublist : (List a) -> (List a) -> Bool;
(isSublist [] x) = True ;
(isSublist (# x y) []) = False ;
(isSublist (# x1 y1) (# x2 y2)) =
              if (= x1 x2) then (isSublist y1 y2) else False ;

ints : Int -> Int -> (List Int) ;
(ints x y) = if (< x y) then (# x (ints (add x 1) y)) else (# x []) ;
```

177 ⟨sets.es 177⟩≡

```
import numbers.es ;

union : (a -> Bool) -> (a -> Bool) -> (a -> Bool) ;
(union s t) = \x.(|| (s x) (t x)) ;

intersect : (a -> Bool) -> (a -> Bool) -> (a -> Bool) ;
(intersect s t) = \x.(&& (s x) (t x)) ;

minus : (a -> Bool) -> (a -> Bool) -> (a -> Bool) ;
(minus s t) = \x.(&& (s x) (= (t x) False)) ;

subset : (a -> Bool) -> (a -> Bool) -> Bool ;
(subset s t) = (pi \x.(implies (s x) (t x))) ;

powerset : (a -> Bool) -> ((a -> Bool) -> Bool) ;
(powerset \x.False)  = \s.(= s \x.False) ;
(powerset \x.if u_SV then True else v_SV) =
           \s.(sigma \t.(sigma \r.(&& ((powerset \x.u_SV) t)
                    (&& ((powerset \x.v_SV) r) ((= s) (union t r)))))) ;
(powerset \x.if u_SV then False else v_SV) = (powerset \x.(&& (not u_SV) v_SV)) ;
(powerset \x.(= x t)) = \s.(|| (= s \y.False) (= s \x.(= x t))) ;
(powerset \x.(|| u_SV v_SV)) =
           \s.(sigma \t.(sigma \r.(&& ((powerset \x.u_SV) t)
                    (&& ((powerset \x.v_SV) r) (= s (union t r)))))) ;

linearise : (a -> Bool) -> (a -> Bool) ;
(linearise \x.False) = \x.False ;
(linearise \x.if u_SV then True else v_SV) =
          (union (linearise \x.u_SV) (linearise \x.v_SV)) ;
(linearise \x.if u_SV then False else v_SV) =
          (linearise \x.(&& (not u_SV) v_SV)) ;
(linearise \x.(= x t)) = \x.if (= x t) then True else False ;
(linearise \x.(|| u_SV v_SV)) = (union (linearise \x.u_SV) (linearise \x.v_SV)) ;

rmdup : (a -> Bool) -> (a -> Bool) ;
(rmdup \x.t_SV) = \x.(rmdup2 t_SV) ;

rmdup2 : Bool -> Bool ;
(rmdup2 False) = False ;
(rmdup2 True) = True ;
(rmdup2 (= x t_SV)) = (= x t_SV) ;
(rmdup2 if (= x t_SV) then True else False) =
                 if (= x t_SV) then True else False ;
(rmdup2 if (= x t_SV) then True else u) =
        if (= x t_SV) then True else (rmdup2 (&& (/= x t_SV) u)) ; Eager ;
(rmdup2 (|| (= x t_SV) u)) = (|| (= x t_SV) (rmdup2 (&& (/= x t_SV) u))) ;
(rmdup2 (|| (|| (= x t_SV) u) v)) = (rmdup2 (|| (= x t_SV) (|| u v))) ;

rmdupCustom : (a -> a -> Bool) -> (a -> Bool) -> (a -> Bool) ;
(rmdupCustom p \x.t_SV) = \x.(rmdupCustom2 p t_SV) ;

rmdupCustom2 : (a -> a -> Bool) -> Bool -> Bool ;
(rmdupCustom2 p False) = False ;
```

```
(rmdupCustom2 p if (= x t_SV) then True else False) =
                        if (= x t_SV) then True else False ;
(rmdupCustom2 p if (= x t_SV) then True else u) =
 if (= x t_SV) then True else (rmdupCustom2 p (&& (not (p x t_SV)) u)) ; Eager;

card : (a -> Bool) -> Int ;
(card s) = (card2 (rmdup s)) ;

card2 : (a -> Bool) -> Int ;
(card2 \x.(= x u)) = 1 ;
(card2 \x.(= u x)) = 1 ;
(card2 \x.(|| (= x u) v_SV)) = (add 1 (card2 \x.v_SV)) ;
(card2 \x.(|| (= u x) v_SV)) = (add 1 (card2 \x.v_SV)) ;
(card2 \x.(|| u_SV v_SV)) = (add (card2 \x.u_SV) (card2 \x.v_SV)) ;
(card2 \x.False) = 0 ;
(card2 \x.if (= x u_SV) then True else v_SV) = (add 1 (card2 \x.v_SV)) ;
(card2 \x.if (&& (<= u_SV/CONST/ x) (<= x v_SV/CONST/)) then True else t_SV) =
       (add (sub v_SV u_SV) (card2 \x.t_SV)) ;
(card2 \x.if (>= x u_SV/CONST/) then True else t_SV) = Infinity ;
(card2 \x.if (<= x u_SV/CONST/) then True else t_SV) = Infinity ;

(card2 \x.if x_SV then True else v_SV) = (add (card2 \x.x_SV) (card2 \x.v_SV)) ;

- typeof(x) - which x? may need to use occurrence.
- (card2 \x.(&& (= (proj1 x) u_SV/CONST/)
-            (= (proj2 x) v_SV/CONST/))) = 1 ; typeof(19) ~ (a * b) -> Bool;

mapFn : (a -> b) -> (a -> Bool) -> (b -> Bool) ;
(mapFn t s) = \x.\exists y.(&& (s y) (= (t y) x)) ;

filterSet : (a -> Bool) -> (a -> Bool) -> (a -> Bool) ;
(filterSet p \x.False) = \x.False ;
(filterSet p \x.(= x v)) = if (p v) then \x.(= x v) else \x.False ;
(filterSet p \x.(|| u_SV v_SV)) = (union (filterSet p \x.u_SV)
                                         (filterSet p \x.v_SV)) ;
(filterSet p \x.if (= x v) then True else v_SV) =
           if (p v) then (union \x.(= x v) (filterSet p \x.v_SV))
           else (filterSet p \x.v_SV) ;

pickAnElement : (a -> Bool) -> a ;
(pickAnElement \x.(= x u)) = u ;
(pickAnElement \x.if (= x u) then True else v_SV) = u ;

switch : Int -> Bool -> Bool -> Bool -> Bool ;
(switch 1 t1 t2 t3) = t1 ;
(switch 2 t1 t2 t3) = t2 ;
(switch 3 t1 t2 t3) = t3 ;

compare : a -> a -> Int ;
(compare x y) = if (= x y) then 1 else if (< x y) then 2 else 3 ;

makeBTree : (a -> Bool) -> (a -> Bool) ;
- we expect the first argument to be in list form
(makeBTree \x.s_SV) = \x.(makeBTree2 (sortIte s_SV)) ;
```

```
makeBTree2 : Bool -> Bool ;
(makeBTree2 False) = False ;
(makeBTree2 if (= x y) then True else False) = (= x y) ;
(makeBTree2 if (= x y) then True else v2) =
         (switch (compare x (midEle if (= x y) then True else v2)) True
                 (makeBTree2 (lessthan (midEle if (= x y) then True else v2)
                                       if (= x y) then True else v2))
                 (makeBTree2 (greaterthan (midEle if (= x y) then True else v2)
                                        if (= x y) then True else v2))) ;

sortIte : Bool -> Bool ;
(sortIte False) = False ;
(sortIte if (= x y) then True else v) = (insIte (= x y) (sortIte v)) ;

insIte : Bool -> Bool -> Bool ;
(insIte (= x y) False) = if (= x y) then True else False ;
(insIte (= x_SV y) if (= z_SV/EQUAL,x_SV/ y2) then True else v) =
         if (< y y2)
         then if (= x_SV y) then True else if (= z_SV y2) then True else v
         else if (= z_SV y2) then True else (insIte (= x_SV y) v) ;

cardBool : Bool -> Int ;
(cardBool False) = 0 ;
(cardBool if (= x y) then True else v) = (add 1 (cardBool v)) ;

get : Float -> Bool -> Bool ;
(get 1.0 if (= x y) then True else v) = y ;
(get n_SV/CONST/ if (= x y) then True else z) = (get (sub n_SV 1.0) z) ;

midEle : Bool -> a ;
(midEle s) = (get (integer (cardBool s) 2) s) ;

lessthan : a -> Bool -> Bool ;
(lessthan z False) = False ;
(lessthan z if (= x y) then True else v2) =
                   if (< y z) then if (= x y) then True else (lessthan z v2)
                   else (lessthan z v2) ;

greaterthan : a -> Bool -> Bool ;
(greaterthan z False) = False ;
(greaterthan z if (= x y) then True else v2) =
                  if (> y z) then if (= x y) then True else (greaterthan z v2)
                  else (greaterthan z v2) ;

removeBound : (a -> Bool) -> Bool ;
(removeBound \x.x_SV) = x_SV ;

simplify2D : ((a * a) -> Bool) -> ((a * a) -> Bool) ;
(simplify2D \x.(&& (= (proj1 x) v1) (= (proj2 x) v2))) = \x.(= x (v1,v2)) ;
(simplify2D \x.(|| u_SV v_SV)) = (union (simplify2D \x.u_SV)
                                        (simplify2D \x.v_SV)) ;
```

⟨multiset functions 180⟩

180 ⟨multiset functions 180⟩≡

```
msetunion : (a -> Int) -> (a -> Int) -> (a -> Int) ;
(msetunion \x.0 m) = m ;
(msetunion \x.if (= x t) then v else w_SV m) =
     \x.if (= x t) then (add v (m t))
        else ((msetunion \x.w_SV (remove \x.(= x t) m)) x) ;

msetdiff : (a -> Int) -> (a -> Int) -> (a -> Int) ;
(msetdiff \x.0 m) = \x.0 ;
(msetdiff \x.if (= x t) then v else w_SV m) =
     \x.if (= x t) then (monus v (m t)) else ((msetdiff \x.w_SV m) x) ;

msetmax : (a -> Int) -> (a -> Int) -> (a -> Int) ;
(msetmax \x.0 m) = m ;
(msetmax \x.if (= x t) then v else w_SV m) =
     \x.if (= x t) then (max v (m t))
        else ((msetmax \x.w_SV (remove \x.(= x t) m)) x) ;

msetmin : (a -> Int) -> (a -> Int) -> (a -> Int) ;
(msetmin \x.0 m) = \x.0 ;
(msetmin \x.if (= x t) then v else w_SV m) =
     \x.if (= x t) then (min v (m t)) else ((msetmin \x.w_SV m) x) ;

msetinc : (a -> Int) -> (a -> Int) -> Bool ;
(msetinc \x.0 m) = True ;
(msetinc \x.if u_SV then v else w_SV m) =
     (&& (pi \x.(implies u_SV (<= v (m x))))
          (msetinc (remove \x.u_SV \x.w_SV) m)) ;

msetmember : a -> (a -> Int) -> Bool ;
(msetmember x m) = (< 0 (m x)) ;
```

Chapter 7

Programming in Escher

7.1 Programming Examples

181 ⟨data.es 181⟩≡

```
list1 : (List Int) ;
list1 = [] ;

list2 : (List Int) ;
list2 = (# 1 []) ;

list3 : (List Int) ;
list3 = (# 1 (# 2 [])) ;

list4 : (List Int) ;
list4 = (# 1 list3) ;

list5 : (List Int) ;
list5 = (# 2 list4) ;

list6 : (List Int) ;
list6 = (# 1 (# 1 (# 2 (# 3 (# 4 []))))) ;

list7 : (List Int) ;
list7 = (# 1 (# 1 (# 2 (# 3 (# 3 (# 4 (# 4 (# 4 [])))))))) ;

list8 : (List Int) ;
list8 = (# 1 (# 1 (# 2 (# 3 (# 3 (# 4 (# 4 (# 3 [])))))))) ;

list88 : (List Int) ;
list88 = (# 1 (# 1 (# 2 (# 3 (# 3 (# 4 (# 4 (# 3 [])))))))) ;

list9 : (List Int) ;
list9 = (# 1 (# 1 (# 2 (# 3 (# 3 []))))) ;

us1 : (List Int) ;
us1 = (# 199 (# 3 (# 2 (# 1 (# 99 (# 12 (# 20 (# 21 (# 51 (# 42 [])))))))))) ;

us2 : (List Int) ;
us2 = (# 7 (# 33 (# 120 (# 1 (# 199 (# 1012 (# 1120 (# 821 (# 851 (# 542 us1)))))))))) ;
```

```
us3 : (List Int) ;
us3 = (# 0 (# 44 (# 12 (# 15 (# 990 (# 125 (# 2220 (# 921 (# 511 (# 442 us2)))))))))) ;

us4 : (List Int) ;
us4 = (# 20 (# 98 (# 290 (# 10 (# 90 (# 123 (# 2300 (# 210 (# 513 (# 342 us3)))))))))) ;

us5 : (List Int) ;
us5 = (# 13 (# 32 (# 29 (# 9 (# 299 (# 122 (# 200 (# 219 (# 5134 (# 242 us4)))))))))) ;

us6 : (List Int) ;
us6 = (# 180 (# 39 (# 27 (# 13 (# 91 (# 112 (# 25 (# 211 (# 151 (# 142 us5)))))))))) ;

- This is an example of a function with arity 1 but effective arity 0
set1 : Int -> Bool ;
set1 = \x.if (= x 1) then True else if (= x 2) then True else False ;

set2 : Int -> Bool ;
set2 = \x.if (= x 2) then True else if (= x 1) then True else False ;

set3 : Int -> Bool ;
set3 = \x.if (= x 1) then True else False ;

set4 : Int -> Bool ;
set4 = \y.if (= y 1) then True else False ;

set5 : Int -> Bool ;
set5 = \x.if (= x 2) then True else if (= x 3) then True else False ;

{-
Annie, Bill, Mary, Joe, Harry, Ginny : People ;
prod0 : (People * People * People * People) ;
prod0 = (Annie, Bill, Mary, Joe) ;
prod1 : (People * People * People * People) ;
prod1 = (Annie, Harry, Ginny, Joe) ;
prod2 : (People * People * People * People) ;
prod2 = (Annie, Harry, Ginny, Joe) ;
-}
```

183 ⟨queries.es 183⟩≡

```
import booleans.es ;
import lists.es ;
import sets.es ;
import data.es ;

- : (isort us6) ; - (2855, 0.467)

- : (qsort us6) ; - (18249, 2.007)

- : (= () ()) ; - (1)
- : (= prod0 prod1) ; - (8)
- : (= prod2 prod1) ; - (10)
- : (= list3 list2) ; - (6)
- : (= list88 list8) ; - (27)    - 0.010

- : ((less \pve2.((= pve2) 1)) \pve3.((= pve3) 1)) ;

- : (= set3 set4) ; - (22)
- : (= set3 set5) ; - (15)
- : (= set1 set2) ; - (66) - 0.021

- There is a difference because of simplifyConjunction2
- : (append (x, y, list2 )) ; - (21)
- : (append (u, v, list2 )) ; - (23)

- : (append (x, y, list8)) ; - (150)
- : (append (u, v, list8)) ; - (128)
- : (append (list9, z, (concat (list8, list6)))) ; - (91)
- : (append (list6, v, (concat (list8, list6)))) ; - (78)
- : (append (list9, (concat (list8, list6)), x)) ; - (149) - var capture
- : (append (list9, (concat (list8, list6)), w)) ; - (155) - 0.164

- : (delete ( 1, ((# 2) ((# 1) [])), ((# 2) []))) ; - (14)
- : (delete ( 12, ((# 2) ((# 1) [])), ((# 2) []))) ; - (16)
- : (delete ( 2, ((# 2) ((# 1) ((# 2) [])  )), x )) ; - (28) - 0.012

- : (permute (((# 1) []) , x)) ; - (19)
- : (permute ( ((# 2) ((# 1) [])), x)) ; - (85) - 0.022

- : (permute ( ((# 3) ((# 2) ((# 1) []))), x)) ; - (292, 0.064)
- : (permute ( (# 10 (# 3 (# 2 (# 1 [])))), x)) ; - (1328, 0.440)
- : (permute ( (# 12 (# 10 (# 3 (# 2 (# 1 []))))), x)) ; - (6305, 7.061)

- : (sorted list7) ; - (74)
- : (sorted list8) ; - (71)
- : (sorted (isort us6)) ; - (8668) - 1.809


- crickettennis = \x.if (= x Cricket) then True
-                    else if (= x Tennis) then True else False ;
- : \x.(pi \y.(implies (crickettennis y) (likes (x,y)))) - (159)
- : \x.(pi \y.(implies (favourite y) (likes (x,y)))) - (129) - 0.038
-
```

```
-
-
- : (powerset \x.((|| ((= x) 1)) ((= x) 2))) - (41)
- : (linearise \x.((|| ((= x) 1)) ((= x) 2))) - (8)
- : ((union \x.((|| ((= x) 1)) ((= x) 2))) \x.((|| ((= x) 1)) ((= x) 3))) - (3)
-
- : (intersect set12 set13) - (31)
- - 0.021
-
- : (msetunion mset1 mset2) - (52)
- : (msetdiff mset1 mset2) - (22)
- : (msetmax mset1 mset2) - (52)
- : (msetmin mset1 mset2) - (19) - 0.015
-
- : (msetinc mset0 mset2) - (34)
- : (msetmember F mset1) - (8)
- : (msetmember A mset1) - (6) - 0.0012
-
- bunch = \x.if (= x (Abloy,3,Short,Normal)) then True
-            else if (= x (Abloy,4,Medium,Broad)) then True else False ;
-
- (projmake (x1,x2,x3,x4)) = x1
- (projlength (x1,x2,x3,x4)) = x3
-
- - cond : Key -> Bool
- cond = \x.(&& (= Abloy (projmake x)) (= Medium (projlength x)))
-
- (setexists1 p t) = (sigma \x.(&& (t x) (p x)))
-
- : (setexists1 cond bunch) - (27, 0.0010)
-

Avon , Bedfordshire , Berkshire ,
Buckinghamshire , Cambridgeshire , Cornwall ,
Devon , Dorset , Essex , Gloucestershire ,
Hampshire , Herefordshire , Hertfordshire ,
Kent , London , Northamptonshire , Oxfordshire ,
Somerset , Surrey , Sussex , Warwickshire ,
Wiltshire , Worcestershire : County ;

Bath , Bournemouth , Bristol , Cheltenham ,
Cirencester , Dorchester , Exeter , Gloucester ,
Penzance , Plymouth , Salisbury , Shaftesbury ,
Sherbourne , Taunton , Torquay , Truro ,
Winchester : City ;

neighbours : (County * County) -> Bool ;
neighbours =
  \x.if (= x (Devon, Cornwall)) then True
     else if (= x (Devon, Dorset)) then True
     else if (= x (Devon, Somerset)) then True
     else if (= x (Avon, Somerset)) then True
     else if (= x (Avon, Wiltshire)) then True
     else if (= x (Avon, Gloucestershire)) then True
```

```
     else if (= x (Dorset, Wiltshire)) then True
     else if (= x (Somerset, Wiltshire)) then True
     else if (= x (Gloucestershire, Wiltshire)) then True
     else if (= x (Dorset, Somerset)) then True
     else if (= x (Dorset, Hampshire)) then True
     else if (= x (Hampshire, Wiltshire)) then True
     else if (= x (Hampshire, Berkshire)) then True
     else if (= x (Hampshire, Sussex)) then True
     else if (= x (Hampshire, Surrey)) then True
     else if (= x (Sussex, Surrey)) then True
     else if (= x (Sussex, Kent)) then True
     else if (= x (London, Surrey)) then True
     else if (= x (London, Kent)) then True
     else if (= x (London, Essex)) then True
     else if (= x (London, Hertfordshire)) then True
     else if (= x (London, Buckinghamshire)) then True
     else if (= x (Surrey, Buckinghamshire)) then True
     else if (= x (Surrey, Kent)) then True
     else if (= x (Surrey, Berkshire)) then True
     else if (= x (Oxfordshire, Berkshire)) then True
     else if (= x (Oxfordshire, Wiltshire)) then True
     else if (= x (Oxfordshire, Gloucestershire)) then True
     else if (= x (Oxfordshire, Warwickshire)) then True
     else if (= x (Oxfordshire, Northamptonshire)) then True
     else if (= x (Oxfordshire, Buckinghamshire)) then True
     else if (= x (Berkshire, Wiltshire)) then True
     else if (= x (Berkshire, Buckinghamshire)) then True
     else if (= x (Gloucestershire, Worcestershire)) then True
     else if (= x (Worcestershire, Herefordshire)) then True
     else if (= x (Worcestershire, Warwickshire)) then True
     else if (= x (Bedfordshire, Buckinghamshire)) then True
     else if (= x (Bedfordshire, Northamptonshire)) then True
     else if (= x (Bedfordshire, Cambridgeshire)) then True
     else if (= x (Bedfordshire, Hertfordshire)) then True
     else if (= x (Hertfordshire, Essex)) then True
     else if (= x (Hertfordshire, Cambridgeshire)) then True
     else if (= x (Hertfordshire, Buckinghamshire)) then True
     else if (= x (Buckinghamshire, Northamptonshire)) then True else False ;

 distance : (City * City * Int) -> Bool ;
 distance =
   \x.if (= x (Plymouth, Exeter, 42)) then True
     else if (= x (Exeter, Bournemouth, 82)) then True
     else if (= x (Bristol, Taunton, 43)) then True
     else if (= x (Bristol, Gloucester, 35)) then True
     else if (= x (Torquay, Exeter, 23)) then True
     else if (= x (Plymouth, Torquay, 24)) then True
     else if (= x (Bristol, Bath, 13)) then True
     else if (= x (Exeter, Taunton, 34)) then True
     else if (= x (Penzance, Plymouth, 78)) then True
     else if (= x (Taunton, Bournemouth, 70)) then True
     else if (= x (Bournemouth, Salisbury, 28)) then True
     else if (= x (Taunton, Salisbury, 64)) then True
     else if (= x (Salisbury, Bath, 40)) then True
```

```
       else if (= x (Bath, Gloucester, 39)) then True
       else if (= x (Bournemouth, Bath, 65)) then True
       else if (= x (Truro, Penzance, 26)) then True
       else if (= x (Plymouth, Truro, 52)) then True
       else if (= x (Shaftesbury, Salisbury, 20)) then True
       else if (= x (Sherbourne, Shaftesbury, 16)) then True
       else if (= x (Dorchester, Bournemouth, 28)) then True
       else if (= x (Salisbury, Winchester, 24)) then True
       else if (= x (Exeter, Sherbourne, 53)) then True
       else if (= x (Sherbourne, Taunton, 29)) then True
       else if (= x (Bath, Cirencester, 32)) then True
       else if (= x (Cirencester, Cheltenham, 16)) then True
       else if (= x (Cheltenham, Gloucester, 9)) then True
       else if (= x (Dorchester, Sherbourne, 19)) then True
       else if (= x (Bath, Shaftesbury, 33)) then True
       else if (= x (Winchester, Bournemouth, 41)) then True
       else if (= x (Exeter, Dorchester, 53)) then True else False ;

isin : (City * County) -> Bool ;
isin =
  \x.if (= x (Bristol, Avon)) then True
       else if (= x (Taunton, Somerset)) then True
       else if (= x (Salisbury, Wiltshire)) then True
       else if (= x (Bath, Avon)) then True
       else if (= x (Bournemouth, Dorset)) then True
       else if (= x (Gloucester, Gloucestershire)) then True
       else if (= x (Torquay, Devon)) then True
       else if (= x (Penzance, Cornwall)) then True
       else if (= x (Plymouth, Devon)) then True
       else if (= x (Exeter, Devon)) then True
       else if (= x (Winchester, Hampshire)) then True
       else if (= x (Dorchester, Dorset)) then True
       else if (= x (Cirencester, Gloucestershire)) then True
       else if (= x (Truro, Cornwall)) then True
       else if (= x (Cheltenham, Gloucestershire)) then True
       else if (= x (Shaftesbury, Dorset)) then True
       else if (= x (Sherbourne, Dorset)) then True else False ;

- : \x.(sigma \y.(&& (|| (distance (Bristol, x, y))
-                        (distance (x, Bristol, y)))
-                    (< y 40)) ) ; - (582, 0.070)

- : \x.\y.(sigma \z.(&& (distance (x,y,z)) (< z 20))) ; - (239, 0.043)

- : \x.(|| (neighbours (Oxfordshire,x)) (neighbours (x, Oxfordshire))) ;
- - (395, 0.059)

- : \x.(sigma \y.(&& (isin (x,y)) (/= y Wiltshire))) ; - (158, 0.037)

- : \x.(sigma \y.(&& (|| (neighbours (Oxfordshire,y))
-                       (neighbours (y, Oxfordshire))) (isin (x,y)))) ;
- - (1174, 0.150)

- : \x.(sigma \y.(&& (isin (x,y)) (|| (neighbours (Oxfordshire,y))
```

```
-             (neighbours (y, Oxfordshire)))  )) ; - (9446, 1.369)

- westcountry : County -> Bool ;
- westcountry = \x.if (= x Devon) then True else if (= x Cornwall) then True
-                  else if (= x Somerset) then True
-                  else if (= x Avon) then True else False ;
- : \x.(sigma \y.(&& (westcountry y) (isin (x,y)))) ; - (293, 0.054)

- : \x.(sigma \y.(sigma \z.(&& (|| (distance (Bristol, y, z))
-     (distance (y, Bristol, z)) )  (&& (< z 50) (isin (y,x))) ))) ; - (915, 0.130)

- : (pi \x.(implies (|| (neighbours (Avon,x)) (neighbours (x,Avon)))
-       (sigma \y.(isin (y,x))) )) ; - (740, 0.364)
-
- : (sigma \x.(&& (isin (Bristol, x))
-                 (pi \z.(implies (sigma \y.(&& (|| (distance (Bristol, z, y))
-                                        (distance (z, Bristol, y))) (< y 40)))
-                                  (isin (z, x)))))) ; - (335, 0.073)
```

187 ⟨tricks.es 187⟩≡

```
- these rules are supposed to bring out conjunctively embedded terms of the
- form (x = t).
- a loop can occur if the NOTAPP condition in the third statement is not
- in place
- (&& u_SV{NOTAPP,=} (= x_SV{VAR} t_SV)) = (&& (= x_SV t_SV) u_SV)
- (&& (&& (= x_SV{VAR} t_SV) u_SV) v_SV) =
-                            (&& (= x_SV t_SV) (&& u_SV v_SV))
- (&& u_SV{NOTAPP,=} (&& (= x_SV{VAR} t_SV) v_SV)) =
-                            (&& (= x_SV t_SV) (&& u_SV v_SV))

- swap equality order
- (= t_SV{NOTVAR} x_SV{VAR}) = (= x_SV t_SV)
```

Comment 7.1.1. The following is a program that calculates the day a particular date falls on.

188 ⟨tvrec.es 188⟩≡

```
(weekday x) = if (<= 2 (whatday x)) then True else False ;

(whatday (x,y,z)) =
   (mod
    (sub
     (add (add (add z
                  (julian_day (x,y,z)))
                  (integer (sub z 1) 4))
                  (integer (sub z 1) 400))
     (integer (sub z 1) 100))
    7)

(decode_day 0) = Saturday
(decode_day 1) = Sunday
(decode_day 2) = Monday
(decode_day 3) = Tuesday
(decode_day 4) = Wednesday
(decode_day 5) = Thursday
(decode_day 6) = Friday

(julian_day (x,1,z)) = x
(julian_day (x,y,z)) = (add x (sumDays ((sub y 1),z)))

(sumDays (y,z)) =
   if (= y 1) then (numberOfDays (1,z))
   else (add (numberOfDays (y,z)) (sumDays ((sub y 1),z))) ;

(numberOfDays (1,x)) = 31
(numberOfDays (2,x)) = if (leap_year x) then 29 else 28 ;
(numberOfDays (3,x)) = 31
(numberOfDays (4,x)) = 30
(numberOfDays (5,x)) = 31
(numberOfDays (6,x)) = 30
(numberOfDays (7,x)) = 31
(numberOfDays (8,x)) = 31
(numberOfDays (9,x)) = 30
(numberOfDays (10,x)) = 31
(numberOfDays (11,x)) = 30
(numberOfDays (12,x)) = 31

(leap_year x) = if (= (mod x 4) 0)
                then if (= (mod x 100) 0) then (= (mod x 400) 0) else True
                else False ;

- : (decode_day (whatday (1,11,2005)))
```

7.2 Programming Tips

Comment 7.2.1. To minimize the impact of the logic programming rules on efficiency, try to put code that can instantiate variables at the leftmost possible position. For example, one should write

$$setExists_1\ p\ t = \exists x.((t\ x) \wedge (p\ x))$$

instead of

$$setExists_1\ p\ t = \exists x.((p\ x) \wedge (t\ x)).$$

Comment 7.2.2.

$$\lambda x.(\exists y.(((neighbours(Oxfordshire, y)) \vee (neighbours(y, Oxfordshire))) \wedge (isin(x, y))))$$

```
-- Steps = 1176
-- Final Answer:
--  \x.if (= x Salisbury) then True else if (= x Gloucester) then True
--     else if (= x Cirencester) then True else (= x Cheltenham)
```

$$\lambda x.(\exists y.((isin(x, y)) \wedge ((neighbours(Oxfordshire, y)) \vee (neighbours(y, Oxfordshire)))))$$

```
-- Total candidate redexes tried = 91659
-- Steps = 9447
-- Final Answer:
--  \x.if (= x Salisbury) then True else if (= x Gloucester) then True
--     else if (= x Cirencester) then True else (= x Cheltenham)
```

Comment 7.2.3. The following definition would not work. Why?

```
    (fac 0) = 1
    (fac n) = (mul (fac (sub n 1)) n)
```

Comment 7.2.4. This also wouldn't work properly (sometimes) for the same reason.

```
(smallest (# x [])) = x
(smallest (# x y)) = if (smaller2 x (smallest y)) then x else (smallest y)
```

Chapter 8

A Listing of the Code Chunks

⟨symbols and their integer representations 145⟩
⟨type name to type objects mapping 157a⟩
⟨io.cc 165⟩
⟨io::common print command 166a⟩
⟨io::common print command ln 166b⟩
⟨io::file 166c⟩
⟨io::file 2 166d⟩
⟨io::file 3 166f⟩
⟨io::file 4 166g⟩
⟨io::file 2a 166e⟩
⟨io.h 164⟩
⟨booleans.es 167⟩
⟨existential statements 170⟩
⟨lists.es 174⟩
⟨multiset functions 180⟩
⟨numbers.es 172⟩
⟨sets.es 177⟩
⟨universal statements 171⟩
⟨data.es 181⟩
⟨queries.es 183⟩
⟨tricks.es 187⟩
⟨tvrec.es 188⟩

Bibliography

[Cla78] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Han94] Michael Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Knu86] Donald E. Knuth. *TEX: The Program*. Addison-Wesley, 1986.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.

[Llo95] John W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, Bristol University, 1995.

[Llo99] John W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

[Llo03] John W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Cognitive Technologies. Springer, 2003.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1992.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, 1996.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*. Oxford, 1998.

[Pax95] Vern Paxson. *Flex: A fast scanner generator*, 2.5 edition, March 1995.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pey02] Simon Peyton Jones (editor). Haskell 98 language and libraries: The revised report. Technical report, 2002.

Index