Mark R. Shinwell
mrs30@cam.ac.uk
Queens' College, Cambridge

The Implementation of FreshML

a meta language for programming with bound names modulo renaming

Computer Science Tripos, Part II, 2000

Mark R. Shinwell – Queens' College

The Implementation of FreshML

Computer Science Tripos, Part II, 2000

Word Count:	11,400 words
Project Originator:	Dr Andrew Pitts
Project Supervisor:	Dr Andrew Pitts

Aims of the Project

The main aim of the project was to produce the first implementation of a new programming language, FreshML, recently designed in the Computer Laboratory. The language features novel type construction facilities and other features to enable programming with bound identifiers modulo renaming.

The project was primarily based on a preliminary version of a research paper by Pitts and Gabbay.

Work Completed

Much theory was developed during the project, including inductively-defined big-step transition relations for translating FreshML source code into FreshML core language code, a principal typing algorithm for the FreshML type system and an environment-based evaluation relation for evaluating FreshML expressions.

An implementation of FreshML, written in ML using the Standard ML of New Jersey suite, was almost entirely completed, providing a means of inputting FreshML source text and having the system type-check and evaluate it.

Special Difficulties

The novelty of the language meant that there was much underlying theory to be understood. After a considerable amount of effort much of this was grasped, which led to some replacement theory for parts of the aforementioned research paper (mainly relating to implementation issues).

Declaration of Originality

I Mark Shinwell of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed ______________________________ **Date** 4th May 2000

Certain small sections of research papers have been quoted in this dissertation. These are marked as specified in Section 1.4.

Contents

1 Introduction **9**

1.1 The project 9

1.2 Language features 9

1.3 Assumptions 11

1.4 Important! 11

1.5 Note 11

2 Preparation **13**

2.1 Project aims 13

2.2 Requirements 13

2.3 Possible extensions 14

2.4 Analysis 14

2.5 Interpreter structure 15

2.6 Timescales and milestones 15

2.7 Modules 16

2.7.1 ML module system 19

2.7.2 Encapsulation 19

2.8 Language 20

2.9 Theory 20

2.9.1 Existing theory to be understood 20

2.9.2 Theories to be invented 20

2.10 Development environment 21

2.11 Development model 21

2.12 Version control and backup 23

2.13 The dissertation 23

2.14 A small functional language 24

3 Implementation **25**

3.1 General details 25

3.2 Lexing and parsing 26

3.3 Datatype declaration handling 27

3.4 Translation to the core 28

3.4.1 Introduction 28

3.4.2 Expressions 28

3.4.3 Case expressions 32

3.5 Core expression optimisation 40

3.5.1 Motivation 40

3.5.2 Substitution algorithm for optimisation 40

3.6 Type inference 41

3.6.1 Introduction 41

3.6.2 Types 41

3.6.3 Restricted types 42

3.6.4 Extra typing rules 42

3.6.5 FreshML typing judgement 43

3.6.6 Substitution axiom 43

3.6.7 Unification 43

3.6.8 Pseudo-code for the typing algorithm 44

3.6.9 Typing of matches 51

3.7 Expression evaluation 55

3.7.1 Dynamic operational semantics 55

3.7.2 Sample source code 62

3.7.3 Top level environment and typing context issues 63

4 Evaluation **65**

4.1 Testing 65

4.2 Sample tests 66

4.2.1 Non-negative integers 66

4.2.2 Datatype testing 67

4.2.3 Capture-avoiding substitution . . . 69
4.3 Validation against requirements . . . 71
4.4 Other improvements . . . 72
4.4.1 Renaming of type variables . . . 72
4.4.2 Infix operators . . . 72
4.4.3 Comments . . . 72
4.5 Improved optimisation . . . 73
4.6 Alternative semantics . . . 73

5 Conclusions **75**

Bibliography **77**

A Grammar **79**

B Core grammar **81**

C Optimisation **83**

Index **89**

Project proposal **91**

1 Introduction

1.1 The project

A meta language for programming with bound names modulo renaming. Such is the description of FreshML given in the research paper on which this project is based [1]. Variable binding operations occur frequently in programming languages, but a natural way of representing bound variables and operations involving them—essential for meta-programming (writing code which manipulates syntactic structures)—has not yet been implemented in a mainstream programming language.

Various research around the world is currently in progress in this area (see references [5], [6] and [8] from [2]). The new language FreshML is the work in progress at Cambridge.

FreshML, although currently small and experimental, provides features aimed to enable straightforward representation and manipulation of bound names. The aim of this project was to design and produce the first implementation of FreshML.

Some more accessible material than the working notes [1] is given in the published paper [2], although it should be noted that some notation in [2] differs from that used in [1]. This project uses the notation of [1].

It should be noted that after writing the *Project Proposal* (included at the end of this document), the name of the language was changed from FML to FreshML.

1.2 Language features

This section provides an introduction to the novel features of FreshML. It should be borne in mind that FreshML is designed to provide features for *representing* structures of other languages.

Consider as an example the implementation of functions in ML manipulating terms of the λ-calculus. Typically a declaration such as the following is used to represent λ-terms, which can be *variables*, *functions*, or *applications*.

```
datatype Lambda = Var   of string
                | Apply of Lambda * Lambda
                | Fn    of string * Lambda
```

We would represent the identity function, $\lambda x \,.\, x$, as `Fn("x", Var "x")`.

The problem is that such definitions are too concrete and *ignore the fact that* `Fn` *is representing a binding operation*. **The name of the bound variable does not matter—unlike what this representation implies!** What is needed is a way of expressing that `Fn` is representing a binding operation to the meta language (here ML), such that it might restrict the use of the bound variables so that problems such as name clashes[1], for example, do not occur. FreshML possesses *abstraction types* to represent binding constructs and allows datatype constructors to explicitly express binding constructs. Object-language[2] variables are expressed as *atoms*, which are given type atm:

```
datatype Lambda = Var   of atm
                | Apply of Lambda * Lambda
                | Fn    of atm . Lambda
```

Given an abstraction, which has type atm . τ, it is not possible to access the bound variable name directly. Instead, one must *concrete* the abstraction at a *fresh name*—a name which has not been used before. This concretion operation, written as, for example, *abst* @ v, gives the body of an abstraction with the new fresh name in place of the bound variable name. The FreshML type system is such that it includes a judgement about *freshness* of names, to ensure that values containing bound variable names *can only be used in ways which are insensitive to renaming*.

For example, a function attempting to list the bound variables of a λ-term will not type-check.

FreshML provides features to create fresh atoms, to abstract over terms and to concrete terms, together with "standard" functional language features such as polymorphism[3], function declaration and application constructs, etc.

[1] Say during substitution of one λ-term for a variable.
[2] The object language is the language we are representing, here the λ-calculus.
[3] Although not `let`-bound polymorphism as is present in ML.

1.3 Assumptions

This dissertation describes not only the implementation of FreshML but also the theory needed for the implementation which was developed throughout the project. Every effort has been made to make the theory here accessible to Computer Scientists who are not familiar with the field, but a basic knowledge of Standard ML, logical connectives, discrete mathematics and rule induction are assumed.

For motivation about the novel features of FreshML the reader is advised to consult [2].

1.4 Important!

> Paragraphs like this one which have a double vertical bar to the left of them are not the author's own work. All other definitions, axioms, rules, and so forth are the work of the author.

1.5 Note

The following convention regarding logical formulae is used in this dissertation.

In formulae such as, for example,

$$\forall x \in S \;.\; f(x) = g(x) \Rightarrow \exists y \;.\; h(y) \text{ holds},$$

the scope of the "." characters extend from their occurrence **to the end of the bracketed block in which they belong, or, if no such block exists, to the end of the expression**. Thus the above expression is equivalent to

$$\forall x \in S \left[f(x) = g(x) \Rightarrow \exists y \left(h(y) \text{ holds} \right) \right],$$

and **not**, for example,

$$\left[\forall x \in S \;.\; f(x) = g(x) \right] \Rightarrow \exists y \;.\; \left(h(y) \text{ holds} \right).$$

2 Preparation

This chapter describes the preparatory work undertaken before the implementation of the FreshML interpreter was started.

2.1 Project aims

The basic aim of the project was to design and implement an interpreter for the FreshML programming language. This included the development of any theory which would be required for various stages of the interpreter.

The fundamental requirements which the interpreter should fulfill were analysed and are stated in Section 2.2 below.

2.2 Requirements

- The interpreter shall take FreshML source text and return the type and value of the expressions therein.
- The interpreter shall be written in ML.
- The interpreter shall be modular, so parts can be easily replaced without disruption to the whole interpreter.

The three requirements do a good job of hiding the complexity of the underlying theory and implementation details!

2.3 Possible extensions

The following were identified as possible areas in which the project could be extended, if time permitted:

- The addition of ML-style `let`-bound polymorphism to the language.
- An interactive front end "à la ML".
- Work on producing helpful, accurate error messages upon erroneous source text input by the user.

2.4 Analysis

This and the next sections give brief commentaries on some of the design decisions made early in the project lifecycle. Refinement of the requirements was undertaken, which led to issues such as the following:

- overall high-level interpreter structure;
- interpreter structure viewed from the perspective of implementing it using a module system such as that of ML;
- the exact language to be implemented;
- necessary theories to be understood;
- theories to be invented;
- development environment;
- development model;
- backup strategies;
- timescales and milestones;
- testability;
- methods of testing and evaluation;
- this dissertation.

These issues are dealt with over the next few sections, with the exception of the majority of the testing issues, which are described in the *Evaluation* chapter.

2.5 Interpreter structure

It was decided that the interpreter should have the structure illustrated in Figure 2.1, which is similar to interpreters for other functional languages such as Haskell, for example.

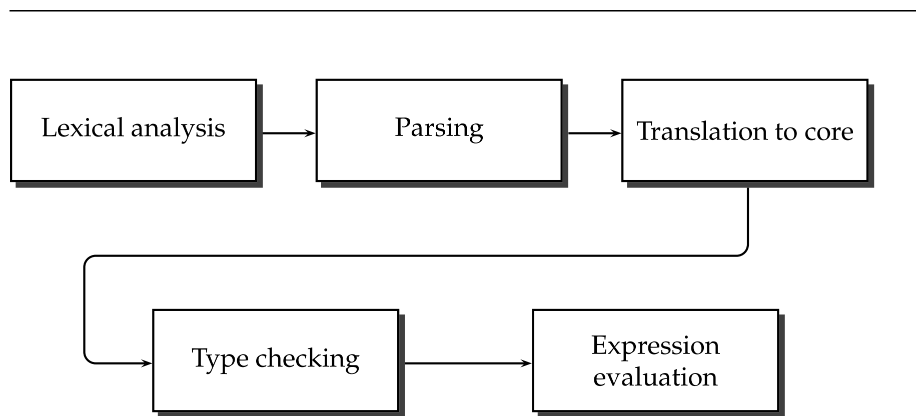

Figure 2.1: FreshML interpreter structure.

FreshML source text is passed to the interpreter. The lexical analysis phase tokenises the source text to produce a token stream. The parser then takes this and produces an abstract syntax tree. This is fed to the core translator to produce core expressions—expressions in a simple, core subset of the full language. The type inference algorithm then attempts to assign a type to each expression. Provided this is successful, the expressions are then evaluated to produce values, which are returned to the user.

2.6 Timescales and milestones

These had already been set out in the *Project Proposal*, a copy of which is included at the end of this dissertation.

2.7 Modules

From the interpreter structure and further deliberations, the following modules were identified and named:

FML	Top-level module (program entry point).
Parse	Lexing and parsing.
Syntax	Abstract syntax tree representation for the full language.
CoreSyntax	Abstract syntax tree representation for the core language.
CoreTrans	Translation to core.
TypeElab	Type inference ("elaboration").
Evaluator	Expression evaluation.
PP	Pretty-printing of expressions, values, etc. for debugging and user feedback.
Test	Testing functions for various modules.

It was decided to implement each in a fashion such that one could easily replace one implementation of a particular module's interface with another (see Section 2.7.1 below).

The specification and dependency information for each module is given below. Dependencies are not "followed through": if module X depends on module A and A in turn depends on B, then only module A is listed as a dependency directly for module X.

FML

This module will depend on:

- the lexer and parser;
- the core translator;
- the type inference algorithm;
- the evaluator;
- the pretty-printer.

The module shall consist of a top-level function which:

- has the type required by the SML functions for heap saving (to create a standalone executable);
- print a startup banner;
- read the FreshML source file specified as argument to the program;
- invoke the lexer and parser;
- invoke the core translator on the parser result (the abstract syntax tree);

- invoke the type inference algorithm on the core expression(s) produced;
- invoke the evaluator on the core expression(s);
- print the result(s) of evaluation together with the type of each expression.

The raising of an exception by any of these stages shall cause execution to halt after that stage.

Parse

This module will depend on:

- ML-Lex and ML-Yacc—standard tools for generating lexers and parsers;
- the generated ML code from ML-Lex and ML-Yacc;
- the abstract syntax tree module.

The module shall:

- read lines from a stream;
- invoke the lexical analyser on each line;
- invoke the parser on the ensuing token stream to produce an abstract syntax tree.

Syntax errors in the input FreshML code shall cause an exception to be raised, incorporating the information about the error gleaned from ML-Lex or ML-Yacc, as appropriate.

Syntax

This module shall not depend on any others and shall define an abstract syntax tree datatype for representing FreshML syntax trees.

CoreSyntax

This module shall define an abstract syntax tree datatype for representing FreshML core syntax trees. This module shall depend on the Syntax module, as certain language constructs such as guards are identical in both the core and full languages (see the Appendices for details).

CoreTrans

This module will depend on:

- both abstract syntax tree modules (Syntax and CoreSyntax).

The module shall:

- read in an abstract syntax tree;
- translate each expression therein to the core language, raising exceptions upon any failure of translation;
- return a set of abstract syntax trees representing the core expressions.

TypeElab

This module will depend on:

- the core abstract syntax tree module.

The module shall:

- read in a set of abstract syntax trees representing core expressions;
- attempt to infer the type of each expression, raising exceptions upon a type elaboration error;
- return the type of each expression, in a human-readable textual form.

Evaluator

This module will depend on:

- the core abstract syntax tree module.

The module shall:

- read in a set of abstract syntax trees representing core expressions;
- attempt to determine the result of evaluating each expression, raising exceptions upon an evaluation error;
- return the result of evaluating each expression, in a human-readable textual form.

PP

This module will depend on:

- the core abstract syntax tree module.

The module shall provide facilities for pretty-printing:

- core expressions (and hence core values, matches, etc.);
- full expressions (and hence values, matches, etc.);
- lists of both of the above.

Test

This module shall provide test harnesses for the various modules, to be written as required for each individual module.

2.7.1 ML module system

It was decided that the ML module system would be used to provide interfaces between the various modules of the interpreter. The scheme was designed as follows: for each module M defining more than just datatypes whose implementation must be exposed (e.g. the abstract syntax trees), the following ML source files were produced:

- `sig`M`.sml` – the *interface* file for M, defining a signature `sig`M;
- M`.sml` – the *implementation* file for M, defining a structure or functor M conforming to the signature `sig`M.

2.7.2 Encapsulation

The specification that a certain implementation structure or functor conforms to a signature would be specified using `:>`, which ensures that only values, types and so forth specified in the appropriate signature are externally visible. This provides *encapsulation*, which is important for maintaining the separation between interface and implementation and for ensuring that modules' private data is not directly accessed by other modules.

2.8 Language

It was decided to slightly extend the language given in [1] to include tuple types as part of the language. Strings and other primitives were considered for addition to the language, but were eventually decided against due to the lack of time for an already complicated project. (There is no *theoretical* difficulty in adding such primitives to the language).

There are also a few other minor deviations from the language described in [1]: for example, constructor expressions are treated as $con\ exp$ rather than $con\ \overrightarrow{v}$ (see the Appendices).

2.9 Theory

A considerable amount of preparation was required in reading and digesting the partially-completed paper [1]. The following lists identify particular areas of theory which were identified during the planning stages to be particularly pertinent to the project.

2.9.1 Existing theory to be understood

- Various parts of [1]:
  - language grammars (full and core languages);
  - translation-to-core rules;
  - typing rules;
  - evaluation rules;
- general theory relating to types and type inference algorithms beyond the scope of the Part II *Types* course;

In addition the author's knowledge of ML had to be extended considerably to encompass the module system and various other language features, neither of which had been covered in the Part IA *Foundations of Computer Science* course.

2.9.2 Theories to be invented

From the outset it was understood to be necessary to develop theory in the following areas.

- a concrete grammar suitable for LALR parsing (to be used by ML-Yacc);
- a type inference algorithm for FreshML;
- a concrete operational semantics.

In the end, far more theory had to be produced than was anticipated—see the *Implementation* chapter for full details. This was all documented as it was invented for future reference and inclusion in this dissertation. This body of material constituted the prime body of documentation produced during the project. Such documentation is important as it is much easier to look through a theoretical description of an algorithm relating to a set of axioms and rules than to look through the code itself.

2.10 Development environment

It was decided to develop the project under Linux as specified in the *Project Proposal* document (see the end of this dissertation). The Standard ML of New Jersey system was selected for the project, owing to its familiar syntax[1] and support for various standard algorithms through its Basis Library.

2.11 Development model

The very nature of the project and the level of uncertainty about certain parts (particularly any difficulties which designing the type inference algorithm might expose) meant that a traditional waterfall model was not appropriate. A more iterative approach was called for after the initial time spent analysing the problem, digesting the relevant literature (particularly [1]) and reworking specifications and module interfaces as necessary.

The model which was envisaged for the project is illustrated in Figure 2.2. Each of the modules would likely be dependent on other modules and so during the construction of each module a stub would have to be written for any modules required but not yet written.

[1] As opposed to CAML, for example.

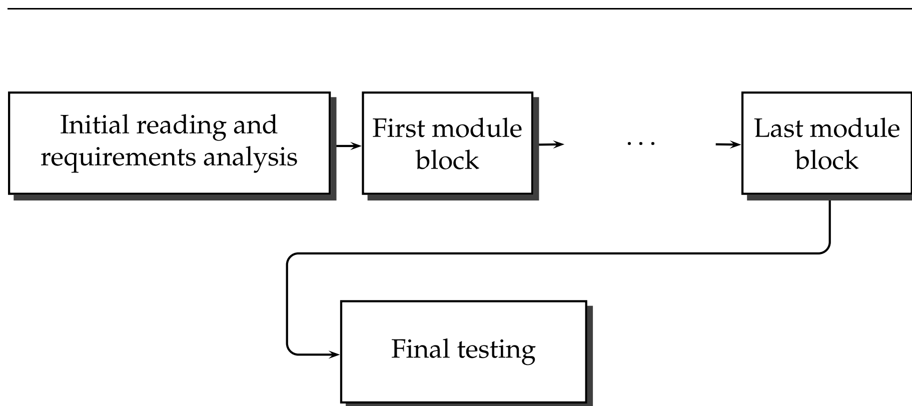

Figure 2.2: FreshML interpreter development model.

The schedule of work for each module block would be as illustrated in Figure 2.3. Each module would be designed (which might involve understanding and/or developing considerable theory), coded, tested and evaluated against the specification. Refinements to the code, design (and possibly even the specification in an extreme case) would then be made until the module is satisfactory.

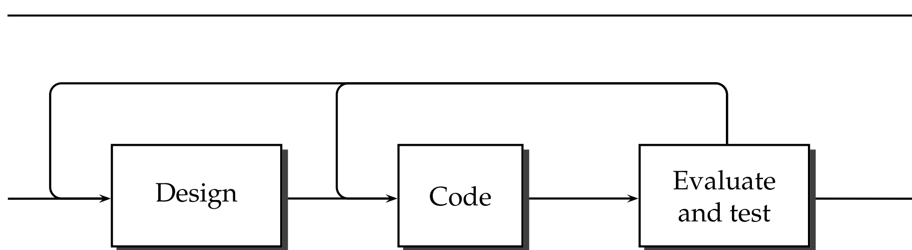

Figure 2.3: The development of one module.

2.12 Version control and backup

It was originally planned to use CVS as the version control system throughout the project, coupled with a scheme of backups thus:

- Regular backups to tape.
- Thrice-daily backups to Pelican.
- Additional backups to Thor, as disk quota permitted.

It was found, as work progressed (particularly with long periods of time spent tweaking implementations of axioms and rules, as described later), that it was more efficient to rely on the thrice-daily labelled backups for fine-grained version control. This enabled easy rollback to work of just a few hours' old, should problems have arisen.

Documentation was also kept under this system throughout the project.

2.13 The dissertation

This dissertation was formatted entirely using the LaTeX text processing system. All source files were kept under same the backup régime as described previously.

The LaTeX source files were arranged one per chapter, with one central "binding together" file, for ease of authoring and editing.

2.14 A small functional language

It was decided that, as a sort of simplified prototype, an interpreter for a very small functional language ought to be written as part of the project preparation, to increase the author's knowledge of ML and in particular the implementation of a simple type inference algorithm.

This system comprised a simple lexer and parser, a type inference stage and an evaluation stage and used the standard ML-Lex and ML-Yacc tools for generation of the lexer and parser. These were implemented as separate ML modules using the interface/implementation paradigm discussed previously in Section 2.7.1.

The following source code fragment shows the sort of expressions allowable in the prototype:

```
(* Abstract syntax tree structure. *)
datatype SynTree = NULL
                 | If of SynTree * SynTree * SynTree
                 | Bool of bool
                 | Fn of string * SynTree
                 | Var of string
                 | Apply of SynTree * SynTree
```

The system was completed successfully in a short time and the experience gained was valuable. An example dialogue with the interpreter is shown below.

```
fltest: (fn a => fn b => (a)(b))

   fn : (('b -> 'c) -> ('b -> 'c))

fltest: fn a => if a then (fn x => true)
                     else (fn x => false)

   fn : (bool -> ('c -> bool))

fltest: ((fn a => if a then (fn x => true)
                       else (fn x => false))
         (true)) (true)

   true : bool

fltest: fn a => ( (a) (a) )

   Type elaboration failed:
       unification failed: 'a with ('a -> 'b)
```

3 Implementation

In this chapter we present both the implementation work in ML and the theory which had to be developed to support it. A solid theoretical foundation was ensured for the implementation by both creating various axioms and rules and by utilising existing theory from the FreshML papers [1] and [2].

3.1 General details

Just over three thousand lines of ML code were produced (including comments). The majority of the code consisted of the implementation of various algorithms whose workings are defined by sets of axioms and rules—a task for which ML is ideal. Thus this chapter presents large sections of the project from a theoretical point of view.

The abstract syntax trees were represented using standard ML datatypes. Red-black trees were used for efficient data storage and retrieval—for example when recording the defined constructors and their arguments for a particular type constructor in a piece of FreshML source text.

As much re-use of existing code was made as possible; for example, standard algorithms were used from the SML/NJ *Basis Library* as required.

3.2 Lexing and parsing

The first stage in the interpreter pipeline is the module to take FreshML source text and convert it into tokens: the action of *lexical analysis*. A *parser* then takes the stream of tokens and produces an abstract syntax tree.

As described in Section 2.7, ML-Lex and ML-Yacc were utilised as standard tools to produce ML source for the lexer and parser. The grammar for FreshML given in [1] had to be enhanced slightly—for example to include tuples (see Section 2.8) and explicit `datatype` declarations—and turned into a concrete unambiguous grammar suitable for feeding to ML-Lex.

The grammar developed is given in Appendix A.

This proved to be a troublesome process; resolving the ambiguities and obtaining the correct precedences is difficult with a relatively large grammar. By the end of the project a few shift/reduce conflicts still remained, but these appeared to manifest themselves only during testing by the insistence of the interpreter on more parentheses than should have been necessary around certain expressions.

Some implementation difficulties were had with integrating the lexer and parser into the main interpreter (mainly due to outdated examples and a lack of documentation in the SML/NJ system), but these were eventually resolved. The small prototype interpreter helped with sorting out these problems.

3.3 Datatype declaration handling

After work had started on the core translation stage (described in Section 3.4), it was noted that it was necessary to have a database of constructors and their types for processing the input parse tree. For example, the arbitrary name "`lam`" could appear in the source text and could either be a value identifier or a constructor name. It was found to be necessary for the purposes of the core translation stage which of these it is.

Thus the additional *datatype declaration handling* stage was inserted between the parser and the core translator. The specification for the module was as follows:

- to receive a parse tree from the parser;
- to identify `datatype` declarations and parse them, producing a database of constructor information;
- resolve all names to either value identifiers or constructors (by searching the constructor database; if the name in question appears in the database as a valid constructor then it is deemed to be a constructor, otherwise it is deemed to be an identifier);
- to report any syntax errors which are identifiable after knowing which names have been defined as constructors;
- to pass a new abstract syntax tree incorporating these identifications to the core translator.

Two levels of nested red-black trees (implemented using the SML/NJ Basis Library) were used to keep the constructor database, which has interfaces permitting, amongst other things, the insertion and extraction of constructor data (to support queries such as "is the name "`lam`" a defined constructor name?").

A new abstract syntax tree datatype was defined which contained only expressions and function declarations (since the `datatype` declarations are processed by this new module) for passing data from this new module to the core translator.

The new module also checks that the user is not trying to define functions with the same name as an existing constructor, or bind a constructor name (for example in a `let` expression). Patterns in matches are checked and identified as being either value identifier or constructor patterns. Various other syntactic checks are made to trap a variety of syntax errors at this stage in the interpreter pipeline.

Note that names which do not match any defined constructor name are deemed to be value identifiers; if the name is in fact *undefined* at the point in the code where it occurs, this will be trapped by the type inference algorithm.

3.4 Translation to the core

3.4.1 Introduction

The FreshML type inference rules (see Section 3.6) operate on a subset of the full FreshML language called the *core language* (often abbreviated to just "the core"). Thus a stage of translation from the full language to the core language is necessary after parsing before type checking can begin. This is similar to the method used in Haskell [4].

The grammar for the core language is reproduced from [1] in Appendix B.

The FreshML paper [1] gives a small-step transition relation to translate from the full language to the core. Implementation started using this relation, but it was soon discovered that such a relation was inconvenient for the production of an implementation.

To work around this problem a big-step relation was produced, which is based on the ideas behind the small-step relation but is expressed very differently. Such a relation is more akin to the way in which an implementation works: informally "give me an expression and I will return the core expression" rather than "give me an expression and I will return an expression which might have been partially translated into the core, but not completely" which is an informal statement of how the small-step relation worked.

This formulation turned out to be difficult and time-consuming, but was eventually completed successfully.

In this section we introduce and justify the big-step relation[1] and describe its implementation in ML.

Notation 3.4.1 Here we take $exp, exp', \ldots \in$ Exp[2]; $e, e', \ldots \in$ CExp; $x, x', \ldots \in$ VId; $pat \in$ Pat and $match \in$ Match. The symbol $\ldots$ denotes a hypothesis for a rule which has been split over more than one line. $\overrightarrow{x}$ denotes a sequence of one or more xs. $\overline{x}$ denotes a sequence of zero or more xs. *con* denotes a constructor name. *gd* denotes a single guard and $\overrightarrow{gd}$ a general guard (which can consist of one or more single guards).

3.4.2 Expressions

Here we present the theory developed for this project for the translation to the core.

A full syntax to core syntax translation is denoted as $exp \Downarrow_E e$ ($exp \in$ Exp, $e \in$ CExp) and is mutually inductively defined (with $\Downarrow_C$) by the axioms and rules given on subsequent pages.

[1]In fact, two mutually inductively defined relations are used.
[2]Consult the Appendices for the definitions of Exp, CExp, etc.

$$x \Downarrow_E x \qquad (\Downarrow_E \text{vid})$$

$$\frac{exp \Downarrow_E e}{\texttt{let}\ x\ \texttt{be fresh in}\ exp \Downarrow_E \texttt{new}\ x\ \texttt{in}\ e} \qquad (\Downarrow_E \text{new})$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e'}{\texttt{let}\ x = exp\ \texttt{in}\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ e'} \qquad (\Downarrow_E \text{let})$$

Figure 3.1: Axiom and rules for simple constructs.

As can be seen from the axiom ($\Downarrow_E$ vid) in Figure 3.1, value identifiers are syntactically unaffected by translation to the core.

As an illustration of how the rules are implemented, consider the ($\Downarrow_E$ let) rule in Figure 3.1. This is read as follows:

> "To translate an expression of the form `let` $x = exp$ `in` exp' to the core language, first translate exp to get e (e being by induction in the core language), then translate exp' to get e' and finally return `let` $x = e$ `in` e' ".

Hence the operation of the two rules above should be clear; they simply translate their individual components (exp, etc.) to the core and then return a core expression.

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \in \text{VId} \quad e' \in \text{CVal}}{exp\ \texttt{.}\ exp' \Downarrow_E e\ \texttt{.}\ e'} \qquad (\Downarrow_E \text{abst}_1)$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \in \text{VId} \quad e' \notin \text{CVal}}{exp\ \texttt{.}\ exp' \Downarrow_E \texttt{let}\ x' = e'\ \texttt{in}\ e\ \texttt{.}\ x'} \qquad \begin{array}{l}(\Downarrow_E \text{abst}_2)\\ x' \text{ fresh}\end{array}$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \notin \text{VId} \quad e' \in \text{CVal}}{exp\ \texttt{.}\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ x\ \texttt{.}\ e'} \qquad \begin{array}{l}(\Downarrow_E \text{abst}_3)\\ x \text{ fresh}\end{array}$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \notin \text{VId} \quad e' \notin \text{CVal}}{exp\ \texttt{.}\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in (let}\ x' = e'\ \texttt{in}\ x\ \texttt{.}\ x'\ \texttt{)}} \qquad \begin{array}{l}(\Downarrow_E \text{abst}_4)\\ x, x' \text{ fresh}\end{array}$$

Figure 3.2: Rules for expressions involving abstractions.

The three rules in Figure 3.2 give translations for expressions involving abstraction values into the core. Abstraction values in the core are only permitted to be of the form $x\ .\ v$, where x is a value identifier and v is a syntactic value (as defined in Appendix B).

Thus expressions in the full language which are not of this form have to be expanded out as described by the rules. In fact, one rule of the form

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e'}{exp\ .\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ (\texttt{let}\ x' = e'\ \texttt{in}\ x\ .\ x'\ )} \quad \begin{array}{l}(\Downarrow_E \text{abst})\\ x, x' \text{ fresh}\end{array}$$

would suffice, but this introduces unnecessary `let` expressions, for example when translating an abstraction $x\ .\ v$ where the first component x is already a value identifier. Thus the four rules in Figure 3.2 have hypotheses to restrict the syntactic categories of the input to each rule.

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \in \text{CVal} \quad e' \in \text{VId}}{exp\ @\ exp' \Downarrow_E e\ @\ e'} \quad (\Downarrow_E \text{con}_1)$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \in \text{CVal} \quad e' \notin \text{VId}}{exp\ @\ exp' \Downarrow_E \texttt{let}\ x' = e'\ \texttt{in}\ e\ @\ x'} \quad \begin{array}{l}(\Downarrow_E \text{con}_2)\\ x' \text{ fresh}\end{array}$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \notin \text{CVal} \quad e' \in \text{VId}}{exp\ @\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ x\ @\ e'} \quad \begin{array}{l}(\Downarrow_E \text{con}_3)\\ x \text{ fresh}\end{array}$$

$$\frac{exp \Downarrow_E e \quad exp' \Downarrow_E e' \quad e \notin \text{CVal} \quad e' \notin \text{VId}}{exp\ @\ exp' \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ (\texttt{let}\ x' = e'\ \texttt{in}\ x\ @\ x')} \quad \begin{array}{l}(\Downarrow_E \text{con}_4)\\ x, x' \text{ fresh}\end{array}$$

Figure 3.3: Rules for expressions involving concretions.

Figure 3.3 gives rules for expressions involving concretions. The core language dictates that concretions must be of the form $v\ @\ x$, where v is a syntactic value and x is a value identifier. Thus the rules expand out more complex forms (e.g. $exp\ @\ exp'$, where $exp,\ exp' \in$ Exp) using `let` expressions, much like the abstraction rules.

The justification for the existence of four concretion rules is identical to that given above for the abstraction rules.

Figure 3.4 gives rules[3] for translation of function declarations and function applications.

$$\frac{\texttt{case}\ x\ \texttt{of}\ \{\ \mathit{match}\ \}\Downarrow_C e}{\texttt{fun}\ \{\ \mathit{match}\ \}\Downarrow_E \texttt{fun}\ f = \{\ x \texttt{=>} e\ \}} \quad (\Downarrow_E \text{fun}_1)\ \ f, x \text{ fresh}$$

$$\frac{\texttt{case}\ x\ \texttt{of}\ \{\ \mathit{match}\ \}\Downarrow_C e}{\texttt{fun}\ f = \{\ \mathit{match}\ \}\Downarrow_E \texttt{fun}\ f = \{\ x \texttt{=>} e\ \}} \quad (\Downarrow_E \text{fun}_2)\ \ x \text{ fresh}$$

$$\frac{\mathit{exp}\Downarrow_E v \quad \mathit{exp}'\Downarrow_E e}{\mathit{exp}\ \mathit{exp}'\Downarrow_E v\ e} \quad (\Downarrow_E \text{app})$$

Figure 3.4: Rules for expressions involving functions and applications.

The part of a function between curly braces is termed a *match*. An example function (in the full FreshML syntax) is illustrated below with the match enclosed in a box. Note how the match includes multiple *clauses*, separated by the | character as in ML.

```
fun f = { Zero => Zero | Suc x => Suc Suc x }
```

The translation process specified by the ($\Downarrow_E$ fun$_i$) rules in Figure 3.4 translates the function body into a `case` statement—function values in the core must be of the form `fun` $f = \{x \texttt{=>} e\}$. The function `f` above would be translated into the core expression:

```
fun f = { a => case a of { Zero => Zero
                          | Suc x => Suc Suc x } }
```

where `a` is a fresh name allocated by the translation rule. Anonymous functions, for example

```
fun { x => x },
```

are allocated a fresh name by the rule ($\Downarrow_E$ fun$_1$) and translated into named functions.

Translation of function applications is trivial: the function and argument are both translated to the core separately and then returned together as a core application expression.

[3]The $\Downarrow_C$ relation is described in Section 3.4.3.

Figure 3.5 gives two rules for the translation of `case` statements. The core language only supports case expressions of the form $\texttt{case}\ x\ \texttt{of}\ \{m\}$ (where m is a *core match* as defined in Appendix B) and thus any expressions in the full language of the form $\texttt{case}\ exp\ \texttt{of}\ \{match\}$ must be expanded.

Translations for the bodies of case expressions are specified by the $\Downarrow_C$ relation, described in section 3.4.3.

$$\frac{exp \Downarrow_E e \quad e \in \text{VId} \quad \texttt{case}\ e\ \texttt{of}\ \{\ match\ \} \Downarrow_C e'}{\texttt{case}\ exp\ \texttt{of}\ \{\ match\ \} \Downarrow_E e'} \quad (\Downarrow_E \text{case}_1)$$

$$\frac{exp \Downarrow_E e \quad e \notin \text{VId} \quad \texttt{case}\ x\ \texttt{of}\ \{\ match\ \} \Downarrow_C e'}{\texttt{case}\ exp\ \texttt{of}\ \{\ match\ \} \Downarrow_E \texttt{let}\ x = e\ \texttt{in}\ e'} \quad \begin{array}{l}(\Downarrow_E \text{case}_2)\\ x \text{ fresh}\end{array}$$

Figure 3.5: Rules for expressions involving `case` statements.

3.4.3 Case expressions

The matches inside case expressions must have translation performed upon them to expand out patterns which are not supported in the core language.

In order to appreciate why the translation is specified in the way given below, it is necessary to consider how a typical `case` expression must be translated. Translation has to occur on both the "high-level" structure to massage the expression into a form acceptable in the core ($\texttt{case}\ x\ \texttt{of}\ \{\dots\}$ – see Figure 3.5), and also on the match inside the body of the statement.

It is thus convenient to express the translation of matches separately from the translation of expressions (the latter being specified by $\Downarrow_E$). Note that in the Figures which follow, the matches on which the rules are working are given inside the body of the enclosing `case` statement, as sometimes expressions have to be generated during the translation process which must lie outside this statement (for example rule ($\Downarrow_C$ case$_1$)).

Axioms and rules

The additional translation of `case` expressions is defined as a relation $\Downarrow_C$, which is mutually inductively defined (with $\Downarrow_E$) by the axiom and rules given in Figures 3.6 to 3.11.

Notation 3.4.2 (In addition to notation described previously). The special token *FAIL* is used to indicate a failure of translation. ⟨ Angle brackets ⟩ indicate optional parts of rules. A rule containing such optional parts must be applied with either all or none of the optional parts present.

An axiom which specifies how the $\Downarrow_C$ relation is to be used is exhibited in Figure 3.6 below. This just constrains the "input" expressions to the translation process specified by the relation to be `case` statements acting on value identifiers. This condition is satisfied by rules ($\Downarrow_E$ case$_1$), ($\Downarrow_E$ case$_2$), ($\Downarrow_E$ fun$_1$) and ($\Downarrow_E$ fun$_2$) described previously.

$$\texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match}\ \Downarrow_C \mathit{FAIL} \qquad (\Downarrow_C \text{case}_0) \qquad (\mathit{exp} \notin \text{VId})$$

Figure 3.6: Use of the $\Downarrow_C$ relation.

Abstraction patterns

Figure 3.7 gives rules for translating matches involving abstraction patterns.

$$\frac{\texttt{case}\ \mathit{exp} \mathbin{@} x\ \texttt{of}\ \{\ \mathit{pat}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \ \texttt{=>}\ \mathit{exp}'\ \}\ \Downarrow_E e'}{\texttt{case}\ \mathit{exp}\ \texttt{of}\ \{\ x\ \texttt{.}\ \mathit{pat}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \ \texttt{=>}\ \mathit{exp}'\ \}\ \Downarrow_C \texttt{new}\ x\ \texttt{in}\ e'} \qquad (\Downarrow_C \text{case}_1)$$

$$\frac{\left[\begin{array}{l}\texttt{case}\ \mathit{exp} \mathbin{@} x\ \texttt{of}\ \{ \\ \quad \mathit{pat}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \ \texttt{=>}\ \mathit{exp}' \\ \quad \mid c \qquad\qquad\qquad \texttt{=>}\ \texttt{case}\ \mathit{exp}\ \texttt{of}\ \{\ \mathit{match}\ \} \\ \} \end{array}\right] \Downarrow_E e'}{\left[\begin{array}{l}\texttt{case}\ \mathit{exp}\ \texttt{of}\ \{ \\ \quad x\ \texttt{.}\ \mathit{pat}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \ \texttt{=>}\ \mathit{exp}' \\ \quad \mid \mathit{match} \\ \} \end{array}\right] \Downarrow_C \texttt{new}\ x\ \texttt{in}\ e'} \qquad \begin{array}{l}(\Downarrow_C \text{case}_2) \\ \\ c \text{ fresh}\end{array}$$

Figure 3.7: Rules involving abstraction patterns.

In FreshML, abstraction patterns such as x . *pat* matching on an expression exp are expanded out by creating a fresh atom x and concreting exp at that atom. Then a pattern match of *pat* against $\mathit{exp} \mathbin{@} x$ (exp concreted at x) proceeds. (This is described more fully in [1]).

To implement this, we simply provide rules to match abstraction patterns and expand them out as specified.

We provide an example for clarity. The expression

```
case exp of { x . y => SomeConstructor }
```

translates to

```
new x in
    case exp @ x of { y => SomeConstructor }
end
```

under the $\Downarrow_C$ relation.

When there are further matches to deal with, for example when we are attempting to translate an expression like

```
case exp of {
    x . y    => Abstraction
  | anything => SomethingElse
}
```

then we have to provide a clause not to match `anything` against "`exp @ x`" but against "`exp`", to preserve the correct meaning of the expression.

For example, the following translation would be wrong:

```
new x in
    case exp @ x of {
        y        => Abstraction
      | anything => SomethingElse
    }
end
```

It is straightforward to see that a correct translation would be (where `z` is a fresh name):

```
new x in
    case exp @ x of {
        y => Abstraction
      | z => case exp of {
                 anything => SomethingElse
             }
    }
end
```

which is the behaviour of the rules specified.

FreshML also includes *guarded patterns* providing atom-equality and atom-inequality tests. (There is no "`if`" statement in the language). For full details the reader is referred to [1]. We illustrate guards in the following example:

```
fun eq = {  (x, y) where x = y  => Equal
          | (x, y) where x <> y => NotEqual }
```

The intended meaning should be obvious.

Thus when $\Downarrow_C$ was being defined it was necessary to take account of these "guarded matches". In the case of the abstraction patterns treated in Figure 3.7 this simply means providing two more rules to preserve the guards during the translation process.

Value identifier patterns

In Figure 3.8 below we present rules for translating value identifier patterns. Note that $\overrightarrow{gd}$ has been replaced by just gd—multiple guards are not permitted for value identifier patterns (since only one atom is present to be tested).

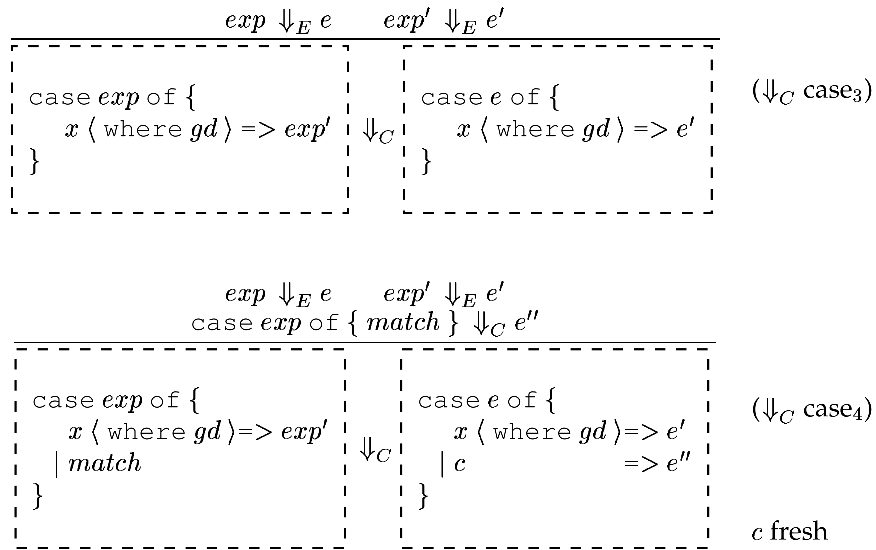

Figure 3.8: Rules involving value identifier patterns.

The treatment of matches involving value identifier patterns is straightforward, because such patterns are permitted in the core language. Thus all that is required is to translate the body of the match into a core expression and any subsequent matches into core matches (the latter by generating another `case` expression).

The rules above provide a formalisation of this intuition.

$$\frac{\boxed{\begin{array}{l}\texttt{case}\ exp'\ \texttt{of}\ \{\\ \quad con\ x\ \texttt{=>}\\ \qquad \texttt{case}\ x\ \texttt{of}\ \{\\ \qquad\quad pat\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp\\ \qquad \}\\ \}\end{array}} \Downarrow_E e}{\texttt{case}\ exp'\ \texttt{of}\ \{\ con\ pat\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp\ \}\ \Downarrow_C e}\quad \begin{array}{l}(\Downarrow_C \text{case}_5)\\ x \text{ fresh}\end{array}$$

$$\frac{\boxed{\begin{array}{l}\texttt{case}\ exp'\ \texttt{of}\ \{\\ \quad con\ x\ \texttt{=>}\\ \qquad \texttt{case}\ x\ \texttt{of}\ \{\\ \qquad\quad pat\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp\\ \qquad\ \mid c \qquad\qquad\qquad \texttt{=>}\ \texttt{case}\ exp'\ \texttt{of}\ \{\ match\ \}\\ \qquad \}\\ \quad \mid match\\ \}\end{array}} \Downarrow_E e}{\texttt{case}\ exp'\ \texttt{of}\ \{\ con\ pat\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp \mid match\ \}\ \Downarrow_C e}\quad \begin{array}{l}(\Downarrow_C \text{case}_6)\\ c, x \text{ fresh}\end{array}$$

Figure 3.9: Rules involving constructor patterns.

Constructor patterns

We now deal with the translation of constructor patterns[4]. These may themselves contain complex patterns which are not acceptable in the core, for example:

```
   MyCon(a.b, c) => Abstraction
 | x             => SomethingElse
```

Non-nullary constructors are treated in the interpreter as a constructor name applied to an expression (which may be a tuple[5]). Thus to translate a constructor pattern to the core, we construct a `case` expression which first matches on the appropriate constructor and *then* matches on the specific constructor argument (if any). This new expression (which is not yet a core expression) is

[4] Note that guarded constructor patterns and guarded tuple patterns are deemed to be acceptable in the core—this is a slight deviation from [1]. Section 3.6.4 explains why this is done.

[5] Rather like in ML, where all non-curried functions take just one argument: function expressions such as `fn (a, b) => a` are of course functions taking a tuple as argument.

recursively translated until all non-core patterns have been dealt with. Thus tuples and other complex patterns are translated by the other translation rules just as if they had been in a match by themselves without the constructor.

Figure 3.9 gives rules for translating constructor patterns.

An example should make this clear. Translation of the expression

```
case exp of { MyCon(a.b, c) => Abstraction }
```

would cause the construction of the following (non-core) expression:

```
case exp of {
    MyCon x =>
        case x of { (a.b, c) => Abstraction }
}
```

where `x` is a fresh name.

This new expression is then recursively translated into the core.

Note how the constructor match has been turned into a match acceptable in the core:

```
MyCon x => ...
```

and a match which can be expanded using other rules:

```
(a.b, c) => ...
```

Tuple patterns

Considerable difficulty was had with formulating rules to express the translation of matches involving tuple patterns. The problem here is that we may have a complex pattern of the form

$$(\,\overrightarrow{pat}\,)$$

where the individual *pat*s are not acceptable core patterns. For example, we might wish to have a match

```
  (x.body, y) => FirstElementIsAbstraction
| (a, b)      => FirstElementIsSomethingElse
```

In order to translate this to the core, following a similar method to that given in [1] (but used there to express the translation of constructor patterns), we walk over the tuple and construct a suitable expression having the same effect as the input `case` expression but with all patterns acceptable in the core.

There are two distinct stages in this process for translating a single match involving a tuple pattern to the core:

- translation of the patterns into acceptable core forms;
- translation of the body of the match into a core expression.

This is quite hard to formalize due to the way in which the patterns need to be walked over in the first step, and the body of the match translated only once when the patterns have been dealt with.

Notation 3.4.3 We take $pat' \notin$ VId, $\overline{gd_1} \subseteq_+ \overrightarrow{pat}$ and $\overline{gd_2} \subseteq_+ pat'$ (where $\overline{gd}$ $\subseteq_+$ *pat* means that each value identifier occurring in the guard $\overline{gd}$ must occur *positively* in *pat*. x occurs positively in itself, and in x' . *pat*, *con pat* or ($\overrightarrow{pat}$) if it occurs positively in *pat* (respectively $\overrightarrow{pat}$)).

$$\dfrac{\boxed{\begin{array}{l}\texttt{case}\ exp'\ \texttt{of}\ \{\\ \quad \texttt{(}\ \overrightarrow{x}\ x'\ \overrightarrow{pat}\ \texttt{)}\ \langle\ \texttt{where}\ \overline{gd_1}\ \rangle\ \texttt{=>}\\ \qquad \texttt{case}\ x'\ \texttt{of}\ \{\\ \qquad\quad pat'\ \langle\ \texttt{where}\ \overline{gd_2}\ \rangle\ \texttt{=>}\ exp\\ \qquad \}\\ \}\end{array}} \Downarrow_E e}{\texttt{case}\ exp'\ \texttt{of}\ \{\ \texttt{(}\ \overrightarrow{x}\ pat'\ \overrightarrow{pat}\ \texttt{)}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp\ \}\ \Downarrow_C e} \quad \begin{array}{l}(\Downarrow_C \text{case}_7)\\ x' \text{ fresh}\end{array}$$

$$\dfrac{\boxed{\begin{array}{l}\texttt{case}\ exp'\ \texttt{of}\ \{\\ \quad \texttt{(}\ \overrightarrow{x}\ x'\ \overrightarrow{pat}\ \texttt{)}\ \langle\ \texttt{where}\ \overline{gd_1}\ \rangle\ \texttt{=>}\\ \qquad \texttt{case}\ x'\ \texttt{of}\ \{\\ \qquad\quad pat'\ \langle\ \texttt{where}\ \overline{gd_2}\ \rangle\ \texttt{=>}\ exp\\ \qquad\ \mid c \qquad\qquad\qquad\quad \texttt{=>}\ \texttt{case}\ exp'\ \texttt{of}\ \{\ match\ \}\\ \qquad \}\\ \quad \mid match\\ \}\end{array}} \Downarrow_E e}{\texttt{case}\ exp'\ \texttt{of}\ \{\ \texttt{(}\ \overrightarrow{x}\ pat'\ \overrightarrow{pat}\ \texttt{)}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle\ \texttt{=>}\ exp \mid match\ \}\ \Downarrow_C e} \quad \begin{array}{l}(\Downarrow_C \text{case}_8)\\ c, x' \text{ fresh}\end{array}$$

Figure 3.10: Rules involving tuple patterns not acceptable in the core.

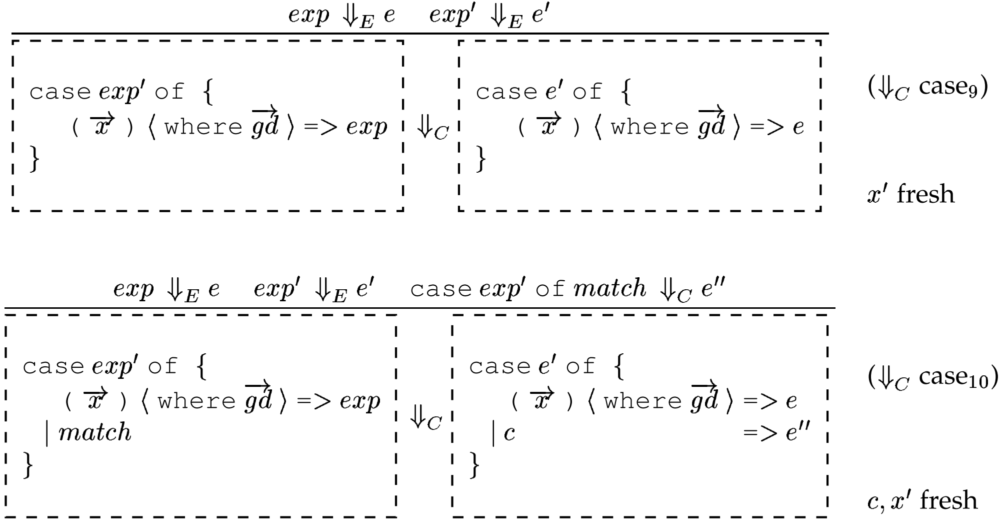

Figure 3.11: Rules involving tuple patterns acceptable in the core.

We distinguish between two structural forms of matches: firstly those whose enclosing `case` statements take the form[6]

$$\texttt{case}\ \mathit{exp'}\ \texttt{of}\ \ \{\ \texttt{(}\ \overrightarrow{x}\ \mathit{pat'}\ \overrightarrow{\mathit{pat}}\ \texttt{)}\ \langle\ \texttt{where}\ \overrightarrow{\mathit{gd}}\ \rangle\ \texttt{=>}\ \mathit{exp}\ \}$$

where $\mathit{pat'} \notin$ VId, and secondly those taking the form

$$\begin{array}{l}\texttt{case}\ e'\ \texttt{of}\ \ \{\\ \quad \texttt{(}\ \overrightarrow{x}\ \texttt{)}\ \langle\ \texttt{where}\ \overrightarrow{\mathit{gd}}\ \rangle\ \texttt{=>}\ e\\ \}\end{array}$$

Observe that the second form matches an input expression whose match pattern is acceptable in the core, whilst the first form matches an input expression whose match pattern is not acceptable in the core (due to complex patterns still being present). The side condition on the first form ensures that only patterns not acceptable in the core match against it.

Figure 3.10 gives rules for manipulating matches whose first pattern is not acceptable in the core. The operation of rule ($\Downarrow_C$ case$_7$) is as follows (($\Downarrow_C$ case$_8$) is similar). This matches against expressions of the first of the two forms given above. We determine the first pattern in the tuple pattern which is not acceptable in the core and construct a (non-core) expression to reduce the pattern in

[6]Here we are just dealing with single matches for the moment.

question to a value identifier pattern. The process is then repeated by induction to walk over all the tuple patterns which need transformation.

At the end of this process we will have an expression which will have a tuple pattern which now only contains patterns which are acceptable in the core (which would match the second of the two forms given above). Next, the rules in Figure 3.11 are used to translate the whole generated expression (which, although its constituent patterns are acceptable in the core, is not yet a core expression) and the match body to the core. This yields a fully translated core expression.

3.5 Core expression optimisation

3.5.1 Motivation

After work on the core translation algorithm was complete, it was observed that certain expressions were exhibiting exponential increases in size during the translation process.

Although it had not been previously planned, it was decided to introduce a core optimization stage to reduce the size of these expressions. This would lead to smaller core expressions, yielding various benefits:

- faster performance of the type inference algorithm and evaluator;
- increased testability and easier debugging;
- shorter expressions to report to the user when a type inference or evaluation error occurs. This is examined further in Section 4.3.

3.5.2 Substitution algorithm for optimisation

In order to implement the optimisation algorithm described below, it is necessary to define the operation of capture-avoiding substitution of one value identifier for all free occurrences of another on FreshML core expressions. This is defined in Appendix C.

We mutually inductively define the relations $\Downarrow_{\mathrm{O}}$, $\Downarrow_{\mathrm{O}_V}$ and $\Downarrow_{\mathrm{O}_M}$ to simplify certain patterns of core expressions as given in Figures C.5 to C.7. The optimisations given are not by any means the best one could do: it was all that could be produced given the constraints of the project. Some discussion about possible improvements is deferred until the *Evaluation* section.

The optimisation relation is given in Appendix C for the interested reader.

3.6 Type inference

3.6.1 Introduction

The core of the project was the design and implementation of a *principal typing* algorithm for FreshML. Such an algorithm takes an expression (in the case of FreshML, a core expression) and determines the type (if any) of that expression. A typing algorithm is said to be *principal* if the type it returns is the *most general* type for that expression, as defined in Definition 3.6.1.

Definition 3.6.1 (Most general types) For an expression E having a type τ, τ is said to be the *most general* type for E if for each other possible type of E there exists a substitution from type variables to types which, when applied to τ, yields this other possible type.

The axioms and rules governing the FreshML type inference algorithm are given later in this section and also in [1]. Note that [2] uses a different typing judgement.

To determine the type of an expression an algorithm is used to recursively walk over the structure of the input expression. This attempts to create instances of the axioms and rules governing the typing judgement along the way, thus constructing a proof tree from the bottom up.

The position of the type checker in the FreshML interpreter pipeline is illustrated in Figure 2.1. Input to the type checker is an optimised core expression.

3.6.2 Types

We implement not only the types specified in the FreshML paper [1] but also tuple types. The set of FreshML types, Ty (ranged over by τ), used in the FreshML implementation is inductively defined by the following production:

τ	$\rightarrow$	α	type variable
	$\mid$	atm	atom
	$\mid$	$\tau \rightarrow \tau$	function type
	$\mid$	$\tau \times \tau \ldots$	tuple type
	$\mid$	atm . τ	atom abstraction type
	$\mid$	$\overline{\tau}\ tcn$	type construction

3.6.3 Restricted types

The novel feature of the FreshML type system is that an additional judgement is used: that relating to *freshness*. An atom a is deemed to be *fresh* for an expression e if e is insensitive to the name of a. After [1] we write

$$\tau \# \overline{a}$$

to denote a *restricted type*, which not only gives the "type" of an expression as usual (τ) but also the atoms $\overline{a}$ known to be fresh for the expression.

3.6.4 Extra typing rules

Extra typing rules had to be invented for tuple patterns. Additionally rules were introduced to deal with guarded constructor and guarded tuple patterns directly, rather than expanding them into the core language as is done in [1]. This increases the complexity of the type inference algorithm but was found to make a very significant reduction in the size of the core expressions.

Notation 3.6.1 We take $\tau, \tau', \ldots \in$ Ty, $\rho, \rho', \ldots \in$ RTy, $\overline{x}, \overline{x}', \ldots \in$ Fin(VId).

- For sets X and Y, the set Subs(X, Y) denotes the set of all partial functions from X to Y, thought of as *substitutions* for elements of Y by elements of X. One such substitution is written $[X/Y]$. $S(\tau)$ indicates the application of the substitution S to τ.
- The set Ty is the set of FreshML types.
- The set RTy is the set of FreshML restricted types.
- The set VId is the set of all value identifiers.
- The set TyCtxt is the set of all valid typing contexts. ($\Gamma \in$ TyCtxt *valid* $\Leftrightarrow \Gamma$ ok, where Γ ok is as defined in Section 3.6.8).
- Fin(X) denotes the set of all finite subsets of the set X.
- $\Gamma, x : \rho$ denotes Γ extended to map x to ρ. In the implementation this corresponds to an overwriting operation if $x \in$ dom(Γ).
- supp(ρ) denotes the *support* of a restricted type ρ, which is the freshness component of ρ. (Thus supp$(\tau \# \overline{a}) = \overline{a}$).
- $\overline{a}a$ denotes $\overline{a} \cup \{a\}$ (where $\overline{a} \in$ Fin(VId) and $a \in$ VId);
- ρa denotes ρ with its support extended by a;
- $\Gamma \# a$ denotes Γ with each atom's support therein extended by a;
- "_" denotes a "don't care" argument or result as in ML.

3.6.5 FreshML typing judgement

As described in [1], the FreshML typing judgements take the forms

$$\begin{array}{ll} \Gamma \vdash e : \rho & \text{for expressions;} \\ \Gamma \vdash m : \rho \Rightarrow \rho & \text{for matches.} \end{array}$$

The typing contexts $\Gamma \in$ TyCtxt are pairs $(\Gamma_{\mathrm{ty}}, \Gamma_{\#})$, where Γ_{ty} maps value identifiers to restricted types and $\Gamma_{\#}$ maps each value identifier x in its domain to a set of value identifiers which are fresh for x.

3.6.6 Substitution axiom

The return value for each clause of the type inference algorithm, such clauses taking syntax trees exp, is of the form $(\tau \# \overline{x}, S)$, where $\tau \# \overline{x} \in$ RTy and S is a partial function representing a substitution [Ty/TyVar] **which does not need to be applied to τ by the caller of the clause**.

3.6.7 Unification

The action of *unification* of two types τ and τ' in type systems such as that of ML is typically as given in Definition 3.6.2.

Definition 3.6.2 (Unification) The process by which a substitution of types for type variables, $S \in$ Sub(Ty, TyVar) can be found for two types τ and τ' such that

$$S(\tau) = S(\tau')$$

Unification algorithms are typically inductively defined on the structure of types. In FreshML however we need a more powerful definition of unification as we have freshness information as well as "regular" types to deal with. This means that we need an algorithm which takes two FreshML restricted types $\tau \# \overline{x}$ and $\tau' \# \overline{x}'$ and returns a unifying substitution for τ and τ' *and also* a freshness component which contains the atoms which are common to $\overline{x}$ and $\overline{x}'$. To see why the latter is so, consider this analogy: when we unify two types τ and τ' we are aiming to find a type to which both τ and τ' can be specialised—finding in essence the "commonality" between the types. Hence when we unify two freshness components $\overline{x}$ and $\overline{x}'$ we are aiming to find information common to both components, which will be the intersection of $\overline{x}$ and $\overline{x}'$.

A suitable definition was constructed and is given below in Definition 3.6.3. It should be noted that this assumes that the unification on the "type" part τ of

restricted types of the form $\tau \# \overline{x}$ is a *most general* unification algorithm, i.e. that the substitution S returned is such that

$$\forall S' \in \mathrm{Sub(Ty, TyVar)} \,.\, \exists S'' \in \mathrm{Sub(Ty, TyVar)} \,.\, S' = S''S,$$

where $S''S$ represents the composition of the substitutions S'' and S.

Definition 3.6.3 (Unification of restricted types) We define the unification of restricted types $\tau \# \overline{x}$ and $\tau' \# \overline{x}'$ as follows:

$$\begin{aligned}&\mathrm{unifyr}(\tau \# \overline{x}, \tau' \# \overline{x}') \equiv \\ &\qquad (\mathrm{unify}(\tau, \tau'), \overline{x} \cap \overline{x}') \in (\mathrm{Subs(Ty, TyVar)} \times \mathrm{Fin(VId)})\end{aligned}$$

where "unify" is the standard most general unification operation of type $\tau \times \tau \to \mathrm{Subs(Ty, TyVar)}$.

3.6.8 Pseudo-code for the typing algorithm

The following sections present pseudo-code and detailed descriptions of the implementation produced. Readers who do not wish to inspect the details are advised to skip to Section 3.7.

Note that the algorithm produces principal *restricted* types: principality for the type component is as usual but principality for the freshness component is ensured by returning the largest possible set of atoms[7] which are fresh for a given expression. For example, when typing a tuple type we determine the freshness components for all the individual tuple elements and then return the *intersection* of these—the largest set of atoms fresh for *every* tuple element.

We define the following:

- the function typ : $\mathrm{TyCtxt} \times \mathrm{CExp} \to \mathrm{RTy} \times \mathrm{Subs(Ty, TyVar)}$ which types a core expression;
- the function typm : $\mathrm{TyCtxt} \times \mathrm{CMatch} \to (\mathrm{RTy} \Rightarrow \mathrm{RTy}) \times \mathrm{Subs(Ty, TyVar)}$ which types a core match;

and make the following claims about firstly the equivalence between typ and the inductively defined relations and secondly the principality of the restricted type returned:

- $\mathrm{typ}(\Gamma, e) = (\tau \# \overline{a}, S) \Rightarrow S(\Gamma) \vdash e : \tau \# \overline{a}$;
- $\Gamma' \vdash e : \tau' \# \overline{a}' \Rightarrow \exists S' \,.\, \Gamma' = (S'S)(\Gamma) \;\wedge\; \tau' = S'(\tau) \;\wedge\; \overline{a}' \subseteq \overline{a}$.

[7] The atoms, of cause, all being declared in the appropriate typing context.

Typing of value identifiers

The rule for typing value identifiers given in the FreshML paper [1] is as follows[8]:

$$\frac{\Gamma\ \mathrm{ok} \quad \Gamma_{\mathrm{ty}}(x) = \tau \quad \{x\} \times \overline{a} \subseteq \Gamma_{\#}}{\Gamma \vdash x : \tau \ \# \ \overline{a}}$$

where Γ ok is as given in Definition 3.6.4 below (slightly rephrased from [1]).

Definition 3.6.4 (Well-formed typing contexts) A typing context is said to be *well-formed* (written Γ ok) when both of the following conditions hold:

- $\forall x \in \mathrm{dom}(\Gamma_{\mathrm{ty}})$. each type $\Gamma_{\mathrm{ty}}(x) \in \mathrm{Ty}$ only involves type constructors declared previously to the expression being typed;
- $\forall (x, a) \in \Gamma_{\#} \, . \, x \in \mathrm{dom}(\Gamma_{\mathrm{ty}}) \ \wedge \ a \in \{x \in \mathrm{dom}(\Gamma_{\mathrm{ty}}) \mid \Gamma_{\mathrm{ty}}(x) = \mathrm{atm}\}$.

We implement this as follows.

- let $\rho = \Gamma(x)$; if x is not found an error is raised;
- return $\rho \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

We claim that Γ ok is ensured by each clause of the typing algorithm.

Typing of the new statement

The typing rule is

$$\frac{\Gamma \ \# \ a \vdash e : \rho a \quad a \notin \mathrm{dom}(\Gamma_{\mathrm{ty}}) \cup \mathrm{supp}(\rho)}{\Gamma \vdash \mathtt{new}\ a\ \mathtt{in}\ e : \rho}$$

We implement this as follows.

- let $\Gamma' \in \mathrm{TyCtxt} = \Gamma, a : (\mathrm{atm} \ \# \ \phi)$ and with a made fresh for all $x \in \mathrm{dom}(\Gamma_{\#})$;
- let $(\tau \ \# \ \overline{a}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (\Gamma', e)$;
- if $a \notin \overline{a}$ then raise an error, "Cannot deduce that a is fresh in e";
- return $((\tau \ \# \ \overline{a}) \setminus \{a\}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

[8] Note that $\{x\} \times \overline{a} \subseteq \Gamma_{\#}$ means effectively "$\Gamma_{\#}(x) = \overline{a}'$ for some $\overline{a}' \supseteq \overline{a}$".

Typing of the let statement

The typing rule is

$$\frac{\Gamma \vdash e : \rho \quad \Gamma, x : \rho \vdash e' : \rho'}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \rho'}$$

We implement this as follows.

- let $(\rho, S) \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar})) = \text{typ}\ (\Gamma, e)$;
- let $(\rho', S') \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar})) = \text{typ}\ (\Gamma, x : \rho, e')$;
- return $(S(\rho'), SS') \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$.

Typing of tuples

Empty tuples are failed immediately with an error.

For non-empty tuples, a typing rule had to be invented (since the version of FreshML in [1] did not support tuple types). The following rule was produced:

$$\frac{\forall i\ .\ 1 \leq i \leq k\ .\ \Gamma \vdash e_i : \rho_i \ \#\ \overline{a}}{\Gamma \vdash (e_1, \ldots, e_k) : \rho_1 \times \ldots \times \rho_k \ \#\ \overline{a}}$$

We implement this as follows.

- determine the types $\{(\tau_i \ \#\ \overline{a}_i)\} \in \text{Fin(RTy)}$ of the tuple components (thus obtaining substitutions from types to type variables $\{S_i\}$);
- determine the intersection of the freshness components $\{\overline{a}_i\} \in \text{Fin(VId)}$ to give a set of value identifiers $\overline{a} \in \text{Fin(VId)}$;
- return $([\tau_1 \times \ldots \times \tau_k]\ \#\ \overline{a}, S) \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$, where S is the composition of the S_k.

Typing of an abstraction value

The typing rule is

$$\frac{\Gamma_{\mathrm{ty}}(a) = \mathrm{atm} \quad \Gamma \vdash v : \tau \;\#\; (\overline{a} \setminus \{a\})}{\Gamma \vdash a \,.\, v : \mathrm{atm} \,.\, \tau \;\#\; \overline{a}}$$

We implement this as follows.

- let $\tau' \in \mathrm{Ty} = \Gamma_{\mathrm{ty}}(a)$;
- let $S' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau', \mathrm{atm})$;
- let $(\tau \;\#\; \overline{a}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (S'(\Gamma), v)$;
- return $([\mathrm{atm} \,.\, \tau] \;\#\; (\overline{a} \cup \{a\}), SS') \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Notice that this returns the *largest* set of fresh atoms possible.

Typing of a concretion

The typing rule is

$$\frac{\Gamma \vdash v : \mathrm{atm} \,.\, \tau \;\#\; \overline{a}a \quad \Gamma_{\mathrm{ty}}(a) = \mathrm{atm} \quad \{a\} \times \overline{a} \subseteq \Gamma_{\#}}{\Gamma \vdash v \ @\ a : \tau \;\#\; \overline{a}}$$

We implement this as follows.

- let $(\tau' \;\#\; \overline{a}) \in \mathrm{RTy} = \Gamma(a)$;
- let $S' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau', \mathrm{atm})$;
- let $(\tau \;\#\; \overline{b}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (S'(\Gamma), v)$;
- let $\alpha \in \mathrm{TyVar}$ be a fresh type variable;
- let $S'' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau, \mathrm{atm} \,.\, \alpha)$;
- let $S_F \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = S''S'S$;
- return $(S_F(\alpha) \;\#\; ((\overline{a} \cap \overline{b}) \setminus \{a\}), S_F) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Typing of a function value

The typing rule is

$$\frac{\begin{array}{c}\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau' \\ f, x \notin \mathrm{dom}(\Gamma_{\mathrm{ty}}) \qquad \Gamma_{\mathrm{ty}}(\overline{a}) = \mathrm{atm} \\ \{x'\} \times \overline{a} \subseteq \Gamma_{\#} \text{ for all } x' \in \mathrm{fv}(e) \setminus \{f, x\}\end{array}}{\Gamma \vdash (\ \texttt{fun}\ f = \{\ x \texttt{=>}\ e\ \}\ ) : (\tau \to \tau') \ \#\ \overline{a}}$$

We implement this as follows.

- let $\alpha \in \mathrm{TyVar}$ be a fresh type variable;
- let $\beta \in \mathrm{TyVar}$ be a fresh type variable;
- let $\Gamma' \in \mathrm{TyCtxt} = \Gamma, x : \alpha \ \#\ \phi, f : \alpha \to \beta \ \#\ \phi$;
- let $(\tau \ \#\ \overline{a}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (\Gamma', e)$;
- let $S' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau, \beta)$;
- let $\overline{v} \in \mathrm{Fin}(\mathrm{VId})$ be the free variables of the function body e;
- let $\overline{b} \in \mathrm{Fin}(\mathrm{VId})$ be the intersection of $\Gamma_{\#}(v)$ for all $v \in \overline{v}$;
- let $S'' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})$ be the composition of the substitutions arising from unifying all $b \in \overline{b}$ with atm;
- let $S_F \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = S''S'S$;
- return $(S_F(\alpha \to \tau) \ \#\ \overline{b}, (S_F$ with all substitutions for α and β removed$)) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Typing of a function application

The typing rule is

$$\frac{\Gamma \vdash v : (\tau' \to \tau) \mathbin{\#} \overline{a} \quad \Gamma \vdash v' : \tau' \mathbin{\#} \overline{a}}{\Gamma \vdash v\, v' : \tau \mathbin{\#} \overline{a}}$$

We implement this as follows.

- let $(\tau \mathbin{\#} \overline{a}, S') \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (\Gamma, v)$;
- let $S \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau, \alpha \to \beta)$, where α and β are fresh type variables;
- let $(\tau' \mathbin{\#} \overline{b}, S'') \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (\Gamma, v')$;
- let $S''' \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = \mathrm{unify}\ (\tau_1, \tau')$;
- let $S_F \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = S'''S''S'S$;
- return $(S_F(\tau_2) \mathbin{\#} (\overline{a} \cup \overline{b}), S_F) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Typing of the case statement

The typing rule is

$$\frac{\Gamma \vdash v : \rho \quad \Gamma \vdash m : \rho \Rightarrow \rho'}{\Gamma \vdash \texttt{case}\ v\ \texttt{of}\ \{\ m\ \} : \rho'}$$

We implement this as follows.

- let $(\tau \mathbin{\#} \overline{a}, S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typ}\ (\Gamma, v)$;
- let $(\tau' \mathbin{\#} \overline{b} \Rightarrow \tau'' \mathbin{\#} \overline{c}, S') \in (\mathrm{RTy} \times \mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typm}\ (S(\Gamma), m)$;
- let $(S'', \overline{d}) \in (\mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) \times \mathrm{Fin}(\mathrm{VId})) = \mathrm{unifyr}\ (S'(\tau) \mathbin{\#} \overline{a}, \tau' \mathbin{\#} \overline{b})$;
- let $S_F \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = S''S'S$;
- return $(S''(\tau'') \mathbin{\#} \overline{c}, S_F) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Typing of a constructor value

The typing rule is

$$\frac{\Gamma \vdash \overrightarrow{v} : ([\overrightarrow{\tau}' \;/\; \overrightarrow{\alpha}]\; \overrightarrow{\tau}) \;\#\; \overline{a} \qquad (\overrightarrow{\tau}'\; tcn) \text{ is defined by } ddec \quad ddec \text{ exists of the form: } \texttt{datatype}\; \overrightarrow{\alpha}\; tcn = \ldots \mid con\; \overrightarrow{\tau} \mid \ldots}{\Gamma \vdash con\; \overrightarrow{v} : (\overrightarrow{\tau}'\; tcn) \;\#\; \overline{a}}$$

We implement this as two clauses: one for nullary constructor values (those taking no argument) and one for non-nullary constructor values. The nullary constructor clause uses the insight that a nullary constructor value is fresh for *all* value identifiers in the typing context.

For nullary constructor values we do the following:

- extract the information about the constructor *con* from the constructor manager—the type constructor *tcn* associated with it and any type variables $\overrightarrow{\alpha}$ parameterising it (e.g. the α in "α `list`");
- allocate fresh type variables $\overrightarrow{\alpha}'$ for each of the type variables $\overrightarrow{\alpha}$ (to ensure they are fresh in the current context, irrespective of what the user named them in the datatype declaration);
- extract all value identifiers $\overline{a}$ in the typing context Γ;
- return ($\overrightarrow{\alpha}'\; tcn \;\#\; \overline{a}$, empty substitution).

For non-nullary constructor values ("*con e*") we do the following:

- let $(\tau_v \;\#\; \overline{a}, S) \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar})) = \text{typ}\;(\Gamma, e)$;
- extract the information about the constructor *con* from the constructor manager—the type constructor *tcn* associated with it, the type of the argument τ and any type variables $\overrightarrow{\alpha}$ parameterising it (e.g. the α in "α `list`");
- allocate fresh type variables $\overrightarrow{\alpha}'$ for each of the type variables $\overrightarrow{\alpha}$ (to ensure they are fresh in the current context, irrespective of what the user named them in the datatype declaration). In parallel with this create a map[9] from TyVar $\rightarrow$ TyVar which details which type variable has been mapped to which fresh one;
- rewrite the constructor argument type τ to use the fresh type variables (by using the map from the previous step), to get a new type τ';
- let $S' \in \text{Subs}(\text{Ty}, \text{TyVar}) = \text{unify}(\tau_v, \tau')$;
- let $S_F \in \text{Subs}(\text{Ty}, \text{TyVar}) = S'S$;

[9]Implemented using a red-black tree using standard SML/NJ library functions.

- obtain the concrete type constructor argument type $\vec{\alpha}'$ by applying the substitution S_F to the fresh type variables of the type constructor in order to specialise (e.g. this might produce "`num list`" from "α `list`");
- return ($\vec{\alpha}'$ *tcn* # $\overline{a}$, S_F).

3.6.9 Typing of matches

The following algorithm typm, described by clause, takes an FreshML match and returns its type.

Single value identifier pattern

The typing rule is

$$\frac{\Gamma, x : \rho \vdash e : \rho'}{\Gamma \vdash x \Rightarrow e : \rho \Rightarrow \rho'}$$

We implement this as follows:

- let $\rho \in \mathrm{RTy} = (\alpha \ \# \ \phi)$ (where α is a fresh type variable);
- let $\Gamma' \in \mathrm{TyCtxt} = \Gamma, x : \rho$;
- let $(\rho', S) \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$ = typ (Γ', e);
- return $(S(\rho) \Rightarrow \rho', S) \in (\mathrm{RTy} \times \mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Value identifier pattern followed by further matches

The typing rule is

$$\frac{\Gamma, x : \rho \vdash e : \rho' \qquad \Gamma \vdash m : \rho \Rightarrow \rho'}{\Gamma \vdash x \Rightarrow e \mid m : \rho \Rightarrow \rho'}$$

We implement this as follows:

- let $(\rho \Rightarrow \rho', S) \in (\mathrm{RTy} \times \mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar})) = \mathrm{typm}\ (\Gamma, m)$;
- let $\Gamma' \in \mathrm{TyCtxt} = \Gamma, x : \rho$;
- let $(\rho'', S') \in (\mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$ = typ (Γ', e);
- let $(S'', \overline{a}) \in (\mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) \times \mathrm{Fin}(\mathrm{VId}))$ = unifyr (ρ', ρ'');
- let $S_F \in \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}) = S''\ S'\ S$ and $(\tau \ \# \ _) \in \mathrm{RTy} = \rho'$
- return $(S_F(\rho) \Rightarrow S_F(\tau \ \# \ \overline{a}), S_F) \in (\mathrm{RTy} \times \mathrm{RTy} \times \mathrm{Subs}(\mathrm{Ty}, \mathrm{TyVar}))$.

Single equality-guarded value identifier pattern

The typing rule is

$$\frac{\Gamma_{\text{ty}}(a) = \text{atm} \quad \{a\} \times \overline{a} \subseteq \Gamma_{\#} \quad \Gamma, x : (\text{atm} \# \overline{a}) \vdash e : \rho}{\Gamma \vdash x \; \texttt{where} \; x = a \; \texttt{=>} \; e : (\text{atm} \# \overline{a}) \Rightarrow \rho}$$

We implement this as follows:

- let $S' \in \text{Subs(Ty, TyVar)} = \text{unify} \; (\Gamma_{\text{ty}}(a), \text{atm})$;
- let $\overline{a} \in \text{Fin(VId)} = \Gamma_{\#}(a)$;
- let $\Gamma' \in \text{TyCtxt} = \Gamma, x : (\text{atm} \# \overline{a})$;
- let $(\rho, S) \in (\text{RTy} \times \text{Subs(Ty, TyVar)}) = \text{typ} \; (\Gamma', e)$;
- return $((\text{atm} \# \overline{a}) \Rightarrow S'(\rho), S'S) \in (\text{RTy} \times \text{RTy} \times \text{Subs(Ty, TyVar)})$.

Equality-guarded value identifier pattern followed by further matches

The typing rule is

$$\frac{\begin{array}{cc}\Gamma_{\text{ty}}(a) = \text{atm} & \{a\} \times \overline{a} \subseteq \Gamma_{\#} \\ \Gamma, x : (\text{atm} \# \overline{a}) \vdash e : \rho & \Gamma \vdash m : (\text{atm} \# \overline{a}) \Rightarrow \rho\end{array}}{\Gamma \vdash x \; \texttt{where} \; x = a \; \texttt{=>} \; e \mid m : (\text{atm} \# \overline{a}) \Rightarrow \rho}$$

We implement this as follows:

- let $S'''' \in \text{Subs(Ty, TyVar)} = \text{unify}(\Gamma_{\text{ty}}(a), \text{atm})$;
- let $(\rho \Rightarrow \rho', S) \in (\text{RTy} \times \text{RTy} \times \text{Subs(Ty, TyVar)}) = \text{typm} \; (\Gamma, m)$;
- let $\overline{a} \in \text{Fin(VId)} = \Gamma_{\#}(a)$;
- let $(S', \overline{b}) \in (\text{Subs(Ty, TyVar)} \times \text{Fin(VId)}) = \text{unifyr} \; ((\text{atm} \# \overline{a}), \rho)$
- let $\Gamma' \in \text{TyCtxt} = \Gamma, x : (\text{atm} \# \overline{b})$;
- let $(\rho'', S'') \in (\text{RTy} \times \text{Subs(Ty, TyVar)}) = \text{typ} \; (\Gamma', e)$;
- let $(S''', \overline{c}) \in (\text{Subs(Ty, TyVar)} \times \text{Fin(VId)}) = \text{unifyr} \; (\rho', \rho'')$
- let $S_F \in \text{Subs(Ty, TyVar)} = S'''' \; S''' \; S'' \; S' \; S$ and $(\tau \# _) \in \text{RTy} = \rho'$
- return $((\text{atm} \# \overline{b}) \Rightarrow S_F(\tau \# \overline{c}), S_F) \in (\text{RTy} \times \text{RTy} \times \text{Subs(Ty, TyVar)})$.

Single inequality-guarded value identifier pattern

The typing rule is

$$\frac{\forall a \in \overrightarrow{a}\ .\ \Gamma_{\text{ty}}(a) = \text{atm} \quad \Gamma, x : (\text{atm} \ \#\ \overrightarrow{a}) \vdash e : \rho}{\Gamma \vdash x \ \texttt{where}\ x \neq \overrightarrow{a} \ \texttt{=>}\ e : (\text{atm} \ \#\ \phi) \Rightarrow \rho}$$

We implement this as follows:

- let $S' \in \text{Subs}(\text{Ty}, \text{TyVar})$ = the composition of the substitutions resulting from "unify ($\Gamma_{\text{ty}}(a)$, atm)" for each $a \in \overrightarrow{a}$;
- let $\Gamma' \in \text{TyCtxt} = \Gamma, x : (\text{atm} \ \#\ \overrightarrow{a})$;
- let $(\rho, S) \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$ = typ (Γ', e);
- return $((\text{atm} \ \#\ \phi) \Rightarrow S'(\rho), S'S) \in (\text{RTy} \times \text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$.

Inequality-guarded value identifier pattern followed by further matches

The typing rule is

$$\frac{\begin{array}{c}\Gamma \vdash m : (\text{atm} \ \#\ \overline{a'}) \Rightarrow \rho \\ \forall a \in \overrightarrow{a}\ .\ \Gamma_{\text{ty}}(a) = \text{atm} \\ \Gamma, x : (\text{atm} \ \#\ \overrightarrow{a}\, \overrightarrow{a}') \vdash e : \rho\end{array}}{\Gamma \vdash x \ \texttt{where}\ x \neq \overrightarrow{a} \ \texttt{=>}\ e \mid m : (\text{atm} \ \#\ \overrightarrow{a}') \Rightarrow \rho}$$

We implement this as follows:

- let $S'''' \in \text{Subs}(\text{Ty}, \text{TyVar})$ = the composition of the substitutions resulting from "unify ($\Gamma_{\text{ty}}(a)$, atm)" for each $a \in \overrightarrow{a}$;
- let $((\tau \ \#\ \overline{b}) \Rightarrow \rho, S) \in (\text{RTy} \times \text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$ = typm (Γ, m);
- let $S' \in \text{Subs}(\text{Ty}, \text{TyVar})$ = unify (τ, atm);
- let $\overline{c} \in \text{Fin}(\text{VId}) = \overrightarrow{a} \cup \overline{b}$;
- let $\Gamma' \in \text{TyCtxt} = \Gamma, x : (\text{atm} \ \#\ \overline{c})$;
- let $(\rho', S'') \in (\text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$ = typ (Γ', e);
- let $(S''', \overline{d}) \in (\text{Subs}(\text{Ty}, \text{TyVar}) \times \text{Fin}(\text{VId}))$ = unifyr (ρ, ρ')
- let $S_F \in \text{Subs}(\text{Ty}, \text{TyVar}) = S''''\ S'''\ S''\ S'\ S$ and $(\tau \ \#\ _) \in \text{RTy} = \rho$
- return $((\text{atm} \ \#\ \overline{b}) \Rightarrow S_F(\tau' \ \#\ \overline{d}), S_F) \in (\text{RTy} \times \text{RTy} \times \text{Subs}(\text{Ty}, \text{TyVar}))$.

Tuple and constructor patterns

The algorithms for implementing guarded and unguarded tuple and constructor patterns were extremely complicated and it was not certain whether they were correct. The algorithms are thus not reproduced here.

The procedure followed is similar to that for constructor values, except there is the additional complexity of the matches to be dealt with. Much common code is shared between the algorithms for the various combinations of unguarded/guarded and tuple/constructor patterns.

The main difficulty encountered is the mixing of equality and inequality guards. It is possible, perfectly reasonably, to write FreshML code such as:

```
fun f = { (a, b, c) where a = b and a <> c => a
        | x => b }
```

The difficulty lies in combining the rules for equality and inequality guards, which have different structures due to the differing amount of freshness information which one can infer from the two types of guards.

The implementation produced is believed to produce correct results but it has not been fully investigated as to whether the types returned are principal. There is some doubt as to whether they are in all cases.

As is discussed in the *Evaluation* chapter, this is an area which needs further research work.

3.7 Expression evaluation

The evaluation of expressions is the final stage in the FreshML interpreter pipeline, as illustrated in Figure 2.1. Before coding could begin, it was necessary to define how the evaluator should work. This is detailed in the next section.

3.7.1 Dynamic operational semantics

A concrete dynamic operational semantics using environments (rather than the very inefficient substitution methods as used for convenience in [1]) was formulated before coding began.

It should be noted that the final version of [2], which did not appear until the project and this dissertation were nearly complete, also includes a semantics similar to that described here. The version presented here was designed independently and is the author's own work.

We define *results* of evaluation thus[10][11]:

Res	$\to$	**Atm**	of	SemAtm	semantic atom
	$\mid$	**Abst**	of	SemAtm $\times$ Res	abstraction
	$\mid$	**Con**	of	Con $\times$ (Res option)	type construction
	$\mid$	**Tuple**	of	$\overrightarrow{\text{Res}}$	tuple
	$\mid$	**Fn**	of	VId $\times$ VId $\times$ CExp $\times$ Env	function closure

We define the *environment* Env to be a partial function mapping value identifiers to results of evaluation.

Notation 3.7.1 Here we take $r, r', \ldots \in$ Res and other notation as previously. The special token *FAIL* is used to indicate a failure of evaluation.

We give the semantic rules in a style similar to that used for Standard ML [3].

The form of the semantic judgement for expressions is

$$E \vdash e \Rightarrow r$$

where E is an environment, e is an expression and r is the result of evaluating the expression e in environment E. The form of the judgement for values is analogous to this. For matches the judgement is of the form

$$E, r \vdash_m m \Rightarrow r'$$

where r is the result of evaluating the value to be matched against and m is a match.

[10] Note that semantic atoms are the type of the entities referred to by values of type atm.
[11] α option is either NONE or SOME(x), where x has type α.

For guards we define

$$E \vdash_g r, gd$$

to mean "in environment E, result r matches guard g" (where r is the result of evaluating the subject of the guard g). We write $\nvdash_g$ to indicate the converse.

If no rule is found to apply to a particular expression, then the result of evaluating that expression is deemed to be *FAIL*.

We assume that the input expression is syntactically correct when giving the semantics.

The dynamic semantics is inductively defined by the axioms and rules in Figures 3.12 to 3.16.

$$E \vdash_v x \Rightarrow E(x) \qquad (\vdash_v \text{vid})$$

$$\frac{E \vdash_v x \Rightarrow r \quad E \vdash_v v \Rightarrow r'}{E \vdash_v x \;.\; v \Rightarrow \textbf{Abst}(r, r')} \qquad (\vdash_v \text{abst})$$

$$E \vdash_v \mathit{con} \Rightarrow \textbf{Con}(\mathit{con}, \text{NONE}) \qquad (\vdash_v \text{con}_1)$$

$$\frac{E \vdash_v e \Rightarrow r}{E \vdash_v \mathit{con}\ e \Rightarrow \textbf{Con}(\mathit{con}, \text{SOME}\ r)} \qquad (\vdash_v \text{con}_2)$$

$$E \vdash_v \texttt{fun}\ f = \{x \texttt{=>}\ e\} \Rightarrow \textbf{Fn}(f, x, e, E) \qquad (\vdash_v \text{fun})$$

Figure 3.12: Dynamic semantics for values.

We provide a brief commentary on the workings of the axioms and rules for evaluation below.

Values

Value identifiers are simply looked up in the appropriate environment. Failure of this process leads to failure of evaluation.

Abstraction values are evaluated by looking the abstracted atom up in the environment to get the corresponding semantic atom, after which the abstraction body is evaluated and the two results returned as a pair.

Constructor values simply have the constructor argument evaluated, if one exists, and then a result is returned pairing this result with the constructor name.

Function values are returned as *function closures*—a binding of the function name, argument name, body and current environment—such that when a function is evaluated it is evaluated *in the environment of definition*. (This is known as static scoping).

$$\frac{E \vdash_v v \Rightarrow r}{E \vdash v \Rightarrow r} \qquad (\vdash \text{val})$$

$$\frac{E \vdash e_1 \Rightarrow r_1 \ \ldots \quad E \vdash e_k \Rightarrow r_k}{E \vdash (e_1, \ldots, e_k) \Rightarrow \mathbf{Tuple}(r_1, \ldots, r_k)} \qquad (\vdash \text{tup})$$

$$\frac{E[x \mapsto \mathbf{Atm}(a)] \vdash e \Rightarrow r}{E \vdash \texttt{new}\ x\ \texttt{in}\ e \Rightarrow r} \qquad (\vdash \text{new})$$

(where a is a new semantic atom).

$$\frac{E \vdash e \Rightarrow r \quad E[x \mapsto r] \vdash e' \Rightarrow r'}{E \vdash \texttt{let}\ x = e\ \texttt{in}\ e' \Rightarrow r'} \qquad (\vdash \text{let})$$

$$\frac{E \vdash_v x \Rightarrow \mathbf{Atm}(a) \quad E \vdash e \Rightarrow \mathbf{Abst}(a', r)}{E \vdash e \mathbin{@} x \Rightarrow (a, a')(r)} \qquad (\vdash \text{conc})$$

where $(a, a')(r)$ denotes the *swapping* of all occurrences of a and a' in r.

$$\frac{E \vdash_v v \Rightarrow r \quad E, r \vdash_m m \Rightarrow r'}{E \vdash \texttt{case}\ v\ \texttt{of}\ \{\ m\ \} \Rightarrow r'} \qquad (\vdash \text{case})$$

$$\frac{\begin{array}{c} E \vdash_v v \Rightarrow \mathbf{Fn}(f, x, e', E') \qquad E \vdash e \Rightarrow r \\ E'[f \mapsto \mathbf{Fn}(f, x, e', E'), x \mapsto r] \vdash e' \Rightarrow r' \end{array}}{E \vdash v\ e \Rightarrow r'} \qquad (\vdash \text{app}_1)$$

$$\frac{\begin{array}{c} E \vdash_v v \Rightarrow r \\ r \text{ is not of the form } \mathbf{Fn}(f, x, e', E') \end{array}}{E \vdash v\ e \Rightarrow \mathit{FAIL}} \qquad (\vdash \text{app}_2)$$

Figure 3.13: Dynamic semantics for expressions.

Expressions

Pure values are simply passed to the value evaluation rules described above.

Tuples are evaluated by evaluating each component one by one and then returning the results packaged up as a tuple result.

Atom creation constructs (`new` x `in` e) are evaluated as follows. First a new semantic atom is created with a unique name. Then this is bound to x by extending the current environment and the body e is then evaluated in this new environment, giving the final result.

Value identifier binding constructs (`let` $x = e$ `in` e') are evaluated as follows. First e is evaluated; the result is then bound to x by extending the current environment and the body e' is then evaluated in this new environment, giving the final result.

Concretions are evaluated by evaluating the expression to concrete and ensuring that it is an abstraction ("a . r"), looking up the semantic atom a' at which to concrete the expression and then returning r with all occurrences of a and a' transposed.

Case expressions (`case` v `of` $\{ m \}$) are evaluated as follows. First the value v is evaluated. Then the match rules are invoked to match the result of evaluating v against the match m (see below).

Function applications are evaluated by first evaluating the first component of the application itself. For evaluation to succeed the result must obviously be a function closure. Assuming this is so, evaluation proceeds. The *environment of definition* of the function (held in the closure) is extended with:

- a binding of the function's bound variable to the result of evaluating the second component of the application itself;
- a binding of the function name to the closure.

The function body is then evaluated in this new environment to give the final result.

Guards

The semantic rules for guards simply relate to matching a single guard, which is either of the form $x = a$ or $x \neq \overrightarrow{a}$, against a result r which has been produced by evaluating the subject of the guard (x in these examples).

The rules simply specify:

- For equality guards ($x = a$), evaluate a and succeed if the result equals r;
- For inequality guards ($x \neq \overrightarrow{a}$), evaluate each $a \in \overrightarrow{a}$ and succeed if the results are all not equal to r.

Matches

The match rules take a result r and a match m (possibly of the form $m_1 \mid m_2 \ldots$) and determine:

- which match, if any, successfully matches against r;
- the result of evaluating the body of the successful match (if any such match exists).

We first deal with unguarded matches.

Empty matches indicate that iterating over all matches has failed to find a successful match and thus evaluation fails.

Value identifier patterns match any value. Thus the environment is extended to bind the pattern value identifier to the result r and the match body is evaluated in this new environment.

Tuple patterns match any tuple of the same length (as the tuple components are just value identifiers—see Appendix B.). Assuming the lengths of the pattern tuple and the tuple r are equal, each tuple component in the pattern tuple is bound in the environment to the corresponding component in r. The match body is then evaluated in this new environment.

Constructor patterns are treated identically to tuple patterns, except for the additional condition that the constructor names in r and the pattern must match.

Now we note the additional conditions required when dealing with guarded matches. These are in fact more simple than might be apparent from the formal definition.

Guarded value identifier patterns match if the guard condition holds.

Guarded tuple patterns match if the guard conditions hold; these conditions may apply to some or all of the components of the tuple. If there exists some guard condition which does not hold then the match fails.

Guarded constructor patterns follow the same logic as guarded tuple patterns.

$$\frac{E \vdash_v v \Rightarrow r' \quad r = r'}{E \vdash_g r, (x = v)} \qquad (\vdash_g \text{eq})$$

$$\frac{\forall v \in \overrightarrow{v}\ .\ (E \vdash_v v \Rightarrow r') \wedge (r \neq r')}{E \vdash_g r, (x \neq \overrightarrow{v})} \qquad (\vdash_g \text{neq})$$

Figure 3.14: Dynamic semantics for guards.

$$E, r \vdash_m \textit{empty match} \Rightarrow \textit{FAIL} \qquad (\vdash_m \text{base})$$

$$\frac{E[x \mapsto r] \vdash e \Rightarrow r'}{E, r \vdash_m x \texttt{=>} e \Rightarrow r'} \qquad (\vdash_m \text{vid}_1)$$

$$\frac{i = j \quad E[x_1 \mapsto r_1, \ldots, x_i \mapsto r_i] \vdash e \Rightarrow r}{E, \textbf{Tuple}(r_1, \ldots, r_i) \vdash_m (x_1, \ldots, x_j) \texttt{=>} e \langle \mid m \rangle \Rightarrow r} \qquad (\vdash_m \text{tup}_1)$$

$$\frac{i \neq j \quad E, \textbf{Tuple}(r_1, \ldots, r_i) \vdash_m m \Rightarrow r}{E, \textbf{Tuple}(r_1, \ldots, r_i) \vdash_m (x_1, \ldots, x_j) \texttt{=>} e \mid m \Rightarrow r} \qquad (\vdash_m \text{tup}_2)$$

$$\frac{i = j \quad con_1 = con_2 \quad E[x_1 \mapsto r_1, \ldots, x_i \mapsto r_i] \vdash e \Rightarrow r}{\begin{array}{l} E, \textbf{Con}(con_1, \text{SOME } (r_1, \ldots, r_i)) \vdash_m \\ con_2(x_1, \ldots, x_j) \texttt{=>} e \langle \mid m \rangle \Rightarrow r \end{array}} \qquad (\vdash_m \text{con}_1)$$

$$\frac{\begin{array}{c} (i \neq j) \vee (con_1 \neq con_2) \\ E, \textbf{Con}(con_1, \text{SOME } (r_1, \ldots, r_i)) \vdash_m m \Rightarrow r \end{array}}{\begin{array}{l} E, \textbf{Con}(con_1, \text{SOME } (r_1, \ldots, r_i)) \vdash_m \\ con_2(x_1, \ldots, x_j) \texttt{=>} e \mid m \Rightarrow r \end{array}} \qquad (\vdash_m \text{con}_2)$$

$$\frac{con_1 = con_2 \quad E, \textbf{Con}(con_1, \text{NONE}) \vdash e \Rightarrow r}{E, \textbf{Con}(con_1, \text{NONE}) \vdash_m con_2 \texttt{=>} e \langle \mid m \rangle \Rightarrow r} \qquad (\vdash_m \text{con}_3)$$

$$\frac{con_1 \neq con_2 \quad E, \textbf{Con}(con_1, \text{NONE}) \vdash_m m \Rightarrow r}{E, \textbf{Con}(con_1, \text{NONE}) \vdash_m con_2 \texttt{=>} e \mid m \Rightarrow r} \qquad (\vdash_m \text{con}_4)$$

Figure 3.15: Dynamic semantics for unguarded matches.

$$\frac{\overrightarrow{gd} = gd \quad E \vdash_g r, gd \quad E[x \mapsto r] \vdash e \Rightarrow r'}{E, r \vdash_m x \texttt{ => } e \texttt{ where } \overrightarrow{gd} \langle \mid m \rangle \Rightarrow r'} \quad (\vdash_m \text{vid}_2)$$

(where *gd* is a *single* guard).

$$\frac{\overrightarrow{gd} = gd \quad E \nvdash_g r, gd \quad E, r \vdash_m m \Rightarrow r'}{E, r \vdash_m x \texttt{ => } e \texttt{ where } \overrightarrow{gd} \mid m \Rightarrow r'} \quad (\vdash_m \text{vid}_3)$$

$$\frac{\begin{array}{c}\forall k \,.\, [gd = (x_k,\ op,\ \dots)] \in \overrightarrow{gd} \,.\, E \vdash_g r_k,\ gd \\ i = j \quad E[x_1 \mapsto r_1, \dots, x_i \mapsto r_i] \vdash e \Rightarrow r\end{array}}{E, \textbf{Tuple}(r_1, \dots, r_i) \vdash_m (x_1, \dots, x_j) \texttt{ => } e \texttt{ where } \overrightarrow{gd}\ \langle \mid m \rangle \Rightarrow r} \quad (\vdash_m \text{tup}_3)$$

(where *op* is $=$ or $\neq$)

$$\frac{\begin{array}{c}[\exists k \,.\, [gd = (x_k,\ op,\ \dots)] \in \overrightarrow{gd} \,.\, E \nvdash_g r_k,\ gd] \ \vee i \neq j \\ E, \textbf{Tuple}(r_1, \dots, r_i) \vdash_m m \Rightarrow r\end{array}}{E, \textbf{Tuple}(r_1, \dots, r_i) \vdash_m (x_1, \dots, x_j) \texttt{ => } e \texttt{ where } \overrightarrow{gd} \mid m \Rightarrow r} \quad (\vdash_m \text{tup}_4)$$

$$\frac{\begin{array}{c}\forall k \,.\, [gd = (x_k,\ op,\ \dots)] \in \overrightarrow{gd} \,.\, E \vdash_g r_k,\ gd \\ i = j \quad con_1 = con_2 \quad E[x_1 \mapsto r_1, \dots, x_i \mapsto r_i] \vdash e \Rightarrow r\end{array}}{\begin{array}{c}E, \textbf{Con}(con_1, \text{SOME } (r_1, \dots, r_i)) \vdash_m \\ con_2(x_1, \dots, x_j) \texttt{ => } e \texttt{ where } \overrightarrow{gd} \langle \mid m \rangle \Rightarrow r\end{array}} \quad (\vdash_m \text{con}_3)$$

$$\frac{\begin{array}{c}[\exists k \,.\, [gd = (x_k,\ op,\ \dots)] \in \overrightarrow{gd} \,.\, E \nvdash_g r_k,\ gd] \vee \\ (i \neq j) \vee (con_1 \neq con_2) \\ E, \textbf{Con}(con_1, \text{SOME } (r_1, \dots, r_i)) \vdash_m m \Rightarrow r\end{array}}{\begin{array}{c}E, \textbf{Con}(con_1, \text{SOME } (r_1, \dots, r_i)) \vdash_m \\ con_2(x_1, \dots, x_j) \texttt{ => } e \texttt{ where } \overrightarrow{gd} \mid m \Rightarrow r\end{array}} \quad (\vdash_m \text{con}_4)$$

Figure 3.16: Dynamic semantics for guarded matches.

3.7.2 Sample source code

The ML implementation of the evaluator is a verbatim translation of the formal rules. An example piece of code, used for evaluating expressions, is included below for perusal. The function `evalExpReal` is a wrapper around `evalExp` such that debugging statements can easily be inserted for execution upon evaluation of an expression.

```
    (* values *)
and evalExp (CSyx.ExpValue(v), env) = evalVal (v, env)

    (* tuples *)
  | evalExp (CSyx.ExpTuple(exps), env) =
      Tuple(evalExpList (exps, env))

    (* new expressions *)
  | evalExp (CSyx.ExpNew(x, exp), env) =
      let val newAtm = freshAtom ()
      in
          evalExpReal (exp, overwriteEnv (x,
                                          newAtm,
                                          env))
      end

    (* let expressions *)
  | evalExp (CSyx.ExpLet(x, exp, exp'), env) =
      let val v = evalExpReal (exp, env)
      in
          evalExpReal (exp', overwriteEnv (x, v, env))
      end

    (* concretions *)
  | evalExp (CSyx.ExpConc(value, a'), env) =
      (case evalVal (value, env) of
          Abst(sa, v) =>
              swapSemanticAtoms (sa, lookupEnvAtm (a',
                                                   env), v)
        | _ => raise EvalFailedInt "Only abstraction \
                                   values can be concreted!"
      )

    (* case expressions *)
  | evalExp (CSyx.ExpCase(v, m), env) =
      let val v' = evalVal (v, env)
      in
          evalMatch (v', m, env)
      end
```

```
  (* function applications *)
| evalExp (CSyx.ExpApp(v, exp), env) =
    let val v1 = evalVal (v, env)
        val v2 = evalExpReal (exp, env)
    in
        (* ensure operator (v1)
           is a function: *)
        case v1 of
            Fn (f, x, body, envOfDefn) =>
                (* ... yes, so evaluate body---in
                       the correct environment! *)
                evalExpReal (body,
                    overwriteEnv (f, v1,
                        overwriteEnv (x, v2,
                            envOfDefn)))
          | _ => raise EvalFailedInt "Attempt to \
                                 apply non-function!"
    end
```

3.7.3 Top level environment and typing context issues

In the interpreter a top level environment and a top level typing context is kept. Every time a function declaration at the top level is parsed, its type is stored in the top level typing context and a function closure is stored in the top level environment.

This is required to enable future references to the function to be resolved.

4 Evaluation

This chapter presents descriptions of the testing procedures for the FreshML interpreter, together with critical analysis and assessment of the project's success. Additionally, information on some areas which caused difficulty is presented.

The project ran fairly closely to the schedule detailed in the *Project Proposal* (included at the end of this dissertation), with the exception that the lexing and parsing phases were pulled forward (in order that other modules requiring abstract syntax trees as input could be tested by entering source text rather than unwieldly expressions to construct the trees themselves) and that the type elaboration phase took considerably longer than anticipated. It turned out that the evaluator was designed and coded in a correspondingly shorter time.

This dissertation was completed several days ahead of schedule.

4.1 Testing

A major difficulty due to the nature of the project is that of test cases. Each has to be individually formulated—there is no automated testing procedure which can be applied.

Each module was tested as it was written. Tests were chosen to exercise all the clauses of the various implementations of the theoretical axioms and rules (for example in the testing of the core translation stage, the ML clauses translating each syntactic construct were exercised).

The testing procedures threw up numerous small errors and caused iteration of the design process: some axioms and rules had to be modified as a result of unforeseen behaviour or just to improve their method of operation.

The ultimate test would be to produce a proof that the typing algorithm is indeed correct with respect to the inductively defined typing relations, but such a proof would be very much beyond the scope of this project.

4.2 Sample tests

Here we provide a few example FreshML expressions which were used during testing, covering "standard" and novel FreshML language features, together with the corresponding output from the interpreter.

4.2.1 Non-negative integers

Let us define a unary encoding of the non-negative integers $\{0, 1, \ldots\}$ as `Zero`, `Succ(Zero)`, `Succ(Succ(Zero))`, etc. We can write this as an FreshML datatype declaration:

```
datatype nat = Zero | Succ of nat;
```

Now we can define a function `plus`, which tests various basic constructs together with unguarded constructor matches and type constructions. The following code defines this function and illustrates the addition $1 + 1$:

```
let plus = fun { x =>
                     fun p = { Zero => x
                             | Succ y => Succ (p y)
                             }
               }
in
    let one = (Succ Zero)
    in
        (plus one) one
    end
end
```

We feed this to the FreshML interpreter to receive the response:

```
Succ Succ Zero : nat
```

as would be expected.

During testing a test harness was included to display the core expressions produced as a result of translating the input code. For this example this was as follows (exactly as formatted by the pretty-printing module):

```
Core expression:
let plus = fun _b = { _c =>
    case _c of
        x => fun p = { _d =>
            case _d of
                Zero () => x
              | _e => case _d of
                    Succ y => Succ ((p y))
                end
            end
        }
    end
} in
    let one = Succ (Zero) in
        let _f = (plus one) in
            (_f one)
        end
    end
end
```

4.2.2 Datatype testing

Some tests were devised explicitly for the purpose of testing the FreshML interpreter's treatment of type constructions and datatypes—one of the most complex areas as is evident from the various core translation, typing and evaluation rules.

The FreshML code below demonstrates an encoding of lists using a polymorphic datatype α `list` and various list manipulation functions:

```
datatype 'a list = Nil
                 | Cons of 'a * ('a list);

fun cons = { (x, xs) => Cons (x, xs) };

fun head = { Cons (x, xs) => x };
fun tail = { Cons (x, xs) => xs };

fun append = { (Nil, l) => l
             | (Cons(x, xs), l) => Cons (x,
                                         append (xs, l))
             };

fun mkList = { x => Cons (x, Nil) };
```

```
fun reverse = { Nil => Nil
              | Cons(x, xs) => append(reverse xs,
                                      mkList x)
              };
```

This and the other datatype tests passed successfully. The output from the finished interpreter in this case was:

```
cons = fn : ('d * 'd list) -> 'd list
head = fn : 'h list -> 'h
tail = fn : 'n list -> 'n list
append = fn : ('db list * 'db list) -> 'db list
mkList = fn : 'rb -> 'rb list
reverse = fn : 'vb list -> 'vb list
```

which is correct (but note Section 4.4.1). We test the evaluator by defining a datatype `word` and run a test thus:

```
datatype word = humpty
              | dumpty
              | sat
              | on
              | the
              | wall;

let humptyDumpty = append (Cons (humpty, Nil),
                           Cons (dumpty, Nil))
in
    let str = append (humptyDumpty,
                      Cons (sat,
                      Cons (on,
                      Cons (the,
                      Cons (wall, Nil)))))
    in
        reverse str
    end
end
```

which gives the expected output:

```
Cons (wall, Cons (the, Cons (on,
    Cons (sat, Cons (dumpty,
                 Cons (humpty, Nil)))))) : word list
```

Note how the type constructor α `list` has been correctly specialised.

4.2.3 Capture-avoiding substitution

Two of the most complicated test expressions were two expressions to perform capture-avoiding substitution on λ-terms. These illustrate the power of FreshML: there is no need to include checks for name clashes as there would be in an implementation using a concrete representation of λ-terms (such as a "traditional" implementation in ML).

The FreshML datatype `Lambda` encodes λ-terms thus:

```
datatype Lambda = Var of atm
                | App of Lambda * Lambda
                | Lam of atm . Lambda;
```

and the function `sub` encodes the substitution of `t` for `a` in the λ-term `s`:

```
fun sub = { (t, a) =>
    fun s = {
        Var x where x = a  => t
      | Var x where x <> a => Var x
      | App (x, x')        => App (s x, s x')
      | Lam x.x'           => Lam (x.(s x'))
    }
};
```

This expression was used during testing of the core translation stage and was observed to exhibit a very large increase in code size from the full language to the core. This observation initiated the creation of the core optimisation stage described in Section 3.5.

Notice the use of:

- guarded constructor patterns;
- tuple patterns.

Feeding the expression to the FreshML interpreter elicits the response:

```
sub = fn : (Lambda * atm) -> Lambda -> Lambda
```

as would be expected.

Note how this datatype can only represent *closed* λ-terms, due to the fact that to create atoms (using `let ... be fresh in`) requires that the atom be fresh for the enclosed expression—which means that it must be abstracted over. For example, `x` would not be fresh for `Var x` (a representation of a non-closed λ-term).

We can illustrate the novel abstraction features by using the FreshML code:

```
let y be fresh in
    let x be fresh in
        x . (y . ( (sub (Var x, y))
                    (Lam(x . (App(Var x, Var y)))) ) )
    end
end
```

This is an encoding in FreshML of the substitution $[x/y]$ (x and y being free) on the open λ-term

$$\lambda x \,.\, x\; y$$

which can be turned into a closed expression thus:

$$\lambda x \,.\, \lambda y \,.\, [\lambda x \,.\, x\; y]\,[x/y]$$

Using a concrete representation of λ-terms we have to ensure name-clashes do not occur, to avoid the erroneous result:

$$\lambda x \,.\, \lambda y \,.\, [\lambda x \,.\, x\; x]$$

Instead we expect a correct result, for example:

$$\lambda x \,.\, \lambda y \,.\, [\lambda z \,.\, z\; x]$$

Indeed, the FreshML system's response is:

```
    d.(c.(Lam e.(App (Var e, Var d)))) : atm.(atm.(Lambda))
```

as expected. However, note how the `sub` function did not have to do any checking for name clashes—this is all sorted out by the language[1].

An alternative implementation of `sub` can also be written thus:

```
    fun sub = { t =>
        fun s = { x.(Var y) where y = x => t
                | x.(Var y) where y <> x => Var y
                | x.(App(y,z)) => App(s (x.y), s (x.z))
                | x.(Lam y.z) => Lam(y.(s (x.z)))
        }
    };
```

which is correctly typed by the interpreter:

```
    sub = fn : Lambda -> atm.(Lambda) -> Lambda
```

[1]Basically because in the fourth clause of `s` (defined inside `sub`), the input term is always concreted at a fresh name.

4.3 Validation against requirements

By the end of the design and coding phases the interpreter was substantially complete, save for some small outstanding issues given below. Overall, the interpreter met all the specification points listed in the *Preparation* section. FreshML source text could be input and interpreted as required.

The final system did fail to give a certain FreshML test expression from [1] the correct type. It was thought that this was due to a minor error in the theory, but such an error was not found.

The most significant problem was the uncertainty surrounding the implementation of the typing algorithm for tuple and constructor matches; as cited previously this is an area where more research needs to be undertaken on producing suitable typing rules.

Due to a lack of time it was not possible to implement any of the suggested additional extensions. This was mainly due to the theoretical parts of the project turning out to be considerably more complicated than was expected at the start of the project.

It is believed that the code correctly implements the axioms and rules in this dissertation. It is, however, possible that there are unseen errors in the theory presented here. These could be caught by producing a proof of correctness as described in section 4.1.

However, there were a number of issues outstanding. During the implementation periods various "flag" comments on minor issues were made in the source code for later attention. Some of these were unfortunately not dealt with—mostly ones relating to efficiency (for example optimising a function to be tail-recursive) or other minor points.

Some issues remain in the current system. We highlight a few of these below:

- there remain a very few conflicts in the grammar, causing it to be necessary to insert more parentheses than should be strictly necessary;
- error reporting could be improved—in particular, ML-Lex and ML-Yacc are guilty of having particularly poor error texts. Errors thrown in the type checking or evaluation stages have to be reported with respect to a core expression, as at the moment there exists no way of relating a particular part of a core expression to the full expression from which it came. This is a strong argument for inventing type inference and evaluation rules that work directly on the full language.
- Looking from a perspective after having developed all the theory and the associated body of knowledge, some code could probably be improved.

4.4 Other improvements

The following sections describe a few features which were not referred to in the *Preparation* section, but which could usefully be added in future.

4.4.1 Renaming of type variables

A feature which was not implemented owing to lack of time was the renaming of type variables given as interpreter output. These should strictly start at α and work up. However, currently the type inference algorithm may return results such as

```
append = fn : 'd list -> 'd list -> 'd list
```

which is correct but not as æsthetically pleasing as

```
append = fn : 'a list -> 'a list -> 'a list
```

4.4.2 Infix operators

It would be desirable to define infix operators, for example infix + instead of prefix `plus`. This would mainly involve changes to the parsing stage and could be implemented without a great deal of difficulty.

4.4.3 Comments

Comments were not implemented in this project mainly due to the emphasis on the novel aspects of the interpreter. As with infix operators, this would only involve changes to the parsing stage and could be implemented without difficulty.

4.5 Improved optimisation

It would be possible to devise more involved optimisation rules to further reduce the size of core expressions. This would, however, require much care in order not to change the meaning of a piece of FreshML code during the translation phase.

One lesson worthy of note which has been learnt from this project is that it might be better to type-check and evaluate full FreshML expressions directly, dispensing with the translation phase, to avoid the additional complexity and to help error reporting as described above. Indeed, those involved in FreshML research are intending to follow this up.

4.6 Alternative semantics

There are some issues concerning efficiency in the current implementation of the FreshML evaluator. This is mainly due to the concretion operation requiring the swapping of atoms in an expression, which is an $O(n)$ process (where n is the "size" of the expression in terms of syntactic complexity).

One possible solution to this would be to design a different operational semantics which modified the environment and results of evaluation in such a way that it would be possible to concrete an expression simply by swapping pointers. This would involve keeping some kind of mapping between each atom and semantic atom in the expression being dealt with.

This is a possible future extension which it would be important to considercarefully when thinking about how FreshML's novel features could be incorporated into a more mainstream language such as ML.

5 Conclusions

Overall the project was deemed to be a success, despite various outstanding issues as detailed in the *Evaluation* section. The very nature of the project—a new language with typing and evaluation rules of a nature not previously developed—meant that it was ambitious to design and implement a whole interpreter system. It was felt that the major achievements of the project were:

- the development of a FreshML principal typing algorithm;
- the development of axioms and rules for expressing the core translation and evaluation phases;
- the production of a substantially complete FreshML interpreter.

Additionally, useful insights into the workings of the FreshML system (for example that it would be better to type-check full expressions directly) have been gained and should be useful in future research.

The project has also vastly increased the author's knowledge of ML, type inference algorithms and semantics.

So what of the future? The theory behind FreshML is being continually developed at the Laboratory. The work produced during this project will enable people to experiment with programming using the novel language features. Work is aimed at eventually allowing languages such as ML to incorporate FreshML's novel features—this would be an exciting development, particularly for those in the automated reasoning and meta-programming sectors. Some difficult issues remain, such as exception handling—and this project has shown that typing and evaluation rules working on the full syntax will save considerable time in designing translation rules and effective ways of reporting errors.

Perhaps one day the language features implemented in this project may become part of an everyday programming language. Only time will tell . . .

Bibliography

[1] Andrew M. Pitts and Murdoch J. Gabbay, *A Meta Language for Programming with Bound Names Modulo Renaming* (working notes; unpublished), Cambridge, September 1999.

[2] Andrew M. Pitts and Murdoch J. Gabbay, *A Metalanguage for Programming with Bound Names Modulo Renaming*. In: R. Backhouse and J. Oliveira (Eds.) *Mathematics of Program Construction*, MPC 2000, Proceedings, Ponte de Lima, Portugal, July 2000. Lecture Notes in Computer Science (Springer-Verlag, 2000), to appear.

[3] Robin Milner, Mads Tofte, Robert Harper and David MacQueen, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[4] Simon Peyton-Jones and J. Hughes, *Report on the programming language Haskell 98*, February 1999.

A Grammar

We give below the FreshML grammar used in the interpreter. This is not the grammar fed to ML-Yacc but rather a version following the style of [1] (from where much of this is taken). For further details the reader is referred to [1]. x is taken to be a value identifier $\in$ VId.

Types	τ	$\rightarrow$	α	type variable
Ty		$\mid$	atm	atom
		$\mid$	atm . τ	abstraction
		$\mid$	$\overrightarrow{\tau}$	type construction
		$\mid$	$\tau \times \tau \ldots$	tuple
		$\mid$	$\tau \rightarrow \tau$	function
Expressions	exp	$\rightarrow$	x	value identifier
Exp		$\mid$	`let` x `be fresh in` exp	fresh atom creation
		$\mid$	`let` x = exp `in` exp	value binding
		$\mid$	exp `.` exp	abstraction
		$\mid$	exp @ exp	concretion
		$\mid$	($\overrightarrow{exp}$)	tuple
		$\mid$	`case` exp `of` $\{$ $match$ $\}$	case expression
		$\mid$	`fun` $\{$ $match$ $\}$	anonymous function
		$\mid$	`fun` x = $\{$ $match$ $\}$	named (recursive) function
		$\mid$	$exp\ exp$	application
		$\mid$	con	constructor
Matches	$match$	$\rightarrow$	pat => exp $\langle \mid match \rangle$	unguarded match
Match		$\mid$	pat `where` $\overrightarrow{gd}$ => exp $\langle \mid match \rangle$	guarded match
Guards	gd	$\rightarrow$	$x = x$	equality guard
Gd		$\mid$	$x \neq \overrightarrow{x}$	inequality guard
Patterns	pat	$\rightarrow$	x	value identifier pattern
Pat		$\mid$	x . pat	abstraction pattern
		$\mid$	($\overrightarrow{pat}$)	tuple pattern
		$\mid$	con $\langle$ pat $\rangle$	constructor pattern

B Core grammar

We give below the FreshML grammar for the core language, reproduced from [1] and modified slightly for this project as previously described. The core languages has the same sets of types, guards and datatypes as the full language. x is taken to be a value identifier $\in$ VId.

Core expressions	e	$\rightarrow$	v	semantic value
CExp		$\mid$	`new` x `in` e	fresh atom creation
		$\mid$	`let` $x = e$ `in` e	value binding
		$\mid$	v @ x	concretion
		$\mid$	($\overrightarrow{e}$)	tuple
		$\mid$	`case` v `of` $\{ m \}$	case expression
		$\mid$	$v\ v$	application
Core matches	m	$\rightarrow$	x => $e\ \langle \mid m \rangle$	unguarded VId match
CMatch		$\mid$	$con\ \langle\ e\ \rangle$ => $e\ \langle \mid m \rangle$	unguarded constructor match
		$\mid$	($\overrightarrow{x}$) => $e\ \langle \mid m \rangle$	unguarded tuple match
		$\mid$	x `where` gd => e $\langle \mid m \rangle$	guarded VId match
		$\mid$	$con\ \langle\ e\ \rangle$ `where` $\overrightarrow{gd}$ => e $\langle \mid m \rangle$	guarded constructor match
		$\mid$	($\overrightarrow{x}$) `where` $\overrightarrow{gd}$ => e $\langle \mid m \rangle$	guarded tuple match
Syntactic values	v	$\rightarrow$	x	value identifier
CVal		$\mid$	x . v	abstraction value
		$\mid$	$con\ \langle\ e\ \rangle$	constructor value
		$\mid$	`fun` $x = \{\ x$ => $e\ \}$	function value

C Optimisation

This Appendix contains material relating to core expression optimisation.

Capture-avoiding substitution

Notation C.0.1 Here we take $e \in$ CExp; $v \in$ CVal; $a, x \in$ VId; $pat \in$ Pat; and $match \in$ Match. con is taken to be a constructor name. We define the special token *REDUNDANT* to indicate a redundant match exception (as in ML— a match which can never be executed is deemed to be erroneous).

$\overrightarrow{e}$ means "one or more of e". $\overrightarrow{e}_1$, $\overrightarrow{e}_2$, ... , refer to the components of $\overrightarrow{e}$ (numbered starting from unity).

σ is a partial function $\subset$ VId $\times$ VId representing the substitution of value identifiers for other value identifiers. fv(e) is taken to be the free variables of the expression e. fv(σ) is taken to be the free variables of the substitution σ. fv($\overrightarrow{gd}$) is taken to be the free variables of the guard $\overrightarrow{gd}$. $I(\sigma)$ is taken to be the image of σ. Read "$S[\sigma](e)$" as "the substitution algorithm applying the substitution σ to e".

We present the algorithm S in Figures C.1 to C.4.

The optimisation relations

We present the mutually inductively defined optimisation relations in Figure C.5.

$S_g[\sigma](a = b) \equiv \sigma[a] = \sigma[b]$

$S_g[\sigma](a \neq bs) \equiv \sigma[a] \neq (bs[\sigma])$ (where $l[\sigma]$ denotes the list l with the substitution σ applied to it).

Figure C.1: Substitution on guards.

$S[\sigma](v) \equiv S_v[\sigma](v)$ (where S_v is defined in Figure C.3).

$S[\sigma]((\overrightarrow{e})) \equiv (S[\sigma](\overrightarrow{e}_1), S[\sigma](\overrightarrow{e}_2), \dots)$

$$S[\sigma](\texttt{ new } v \texttt{ in } e) \equiv \begin{cases} \texttt{new } a \texttt{ in } S[\sigma[v \mapsto a]](e) & (v \in I(\sigma)) \\ \texttt{new } v \texttt{ in } S[\sigma](e) & (v \notin I(\sigma)) \end{cases}$$

(where $a \neq v$ is a name not in fv(σ) $\cup$ fv(e)).

$$S[\sigma](\texttt{ let } v = e \texttt{ in } e') \equiv \begin{cases} \texttt{let } a = S[\sigma](e) & (v \in I(\sigma)) \\ \qquad \texttt{in } S[\sigma[v \mapsto a]](e') & \\ \texttt{let } v = S[\sigma](e) \texttt{ in } S[\sigma](e') & (v \notin I(\sigma)) \end{cases}$$

(where $a \neq v$ is a name not in fv(σ) $\cup$ fv(e) $\cup$ fv(e')).

$S[\sigma](v \;@\; a) \equiv S_v[\sigma](v) \;@\; \sigma(a)$

$S[\sigma](\texttt{ case } v \texttt{ of } \{ \, match \, \}) \equiv \texttt{case } S_v[\sigma](v) \texttt{ of } \{ \, S_m[\sigma](match) \, \}$

(where S_m is defined in Figure C.4).

$S[\sigma](v \; v') \equiv S_v[\sigma](v) \; S_v[\sigma](v')$

Figure C.2: Substitution on core expressions.

$S_v[\sigma](v) \equiv \sigma(v)$

$S_v[\sigma](v \,.\, w) \equiv (\sigma(v)) \,.\, S_v[\sigma](w)$

$S_v[\sigma](con) \equiv con$

$S_v[\sigma](con \; e) \equiv con \; (S[\sigma](e))$

$$S_v[\sigma](\texttt{fn } f \; arg \texttt{ => } e) \equiv \begin{cases} \texttt{fn } a \; arg \texttt{ => } S[\sigma[f \mapsto a]](e) & (arg \in I(\sigma)) \\ \texttt{fn } f \; arg \texttt{ => } S[\sigma](e) & (arg \notin I(\sigma)) \end{cases}$$

(where $a \neq f$ is a name not in fv(σ) $\cup$ fv(arg) $\cup$ fv(e)).

Figure C.3: Substitution on core values.

$$S_m[\sigma](v \texttt{=>} e) \equiv \begin{cases} a \texttt{=>} S[\sigma[v \mapsto a]](e) & (v \in I(\sigma)) \\ v \texttt{=>} (S[\sigma](e)) & (v \notin I(\sigma)) \end{cases}$$

(where $a \neq v$ is a name not in $\mathrm{fv}(\sigma) \cup \mathrm{fv}(e)$).

$$S_m[\sigma](v \texttt{=>} e \;\texttt{where}\; gd\,) \equiv \begin{cases} a \texttt{=>} S[\sigma[v \mapsto a]](e) & (v \in I(\sigma)) \\ \qquad \texttt{where}\; S_g[\sigma](\{gd\}) & \\ v \texttt{=>} (S[\sigma](e)) & (v \notin I(\sigma)) \\ \qquad \texttt{where}\; S_g[\sigma](\{gd\}) & \end{cases}$$

(where S_g is defined in Figure C.1 and $a \neq v$ is a name not in $\mathrm{fv}(\sigma) \cup \mathrm{fv}(e) \cup \mathrm{fv}(\{gd\})$).

$$S_m[\sigma](con \texttt{=>} e) \equiv con \texttt{=>} (S[\sigma](e))$$

$$S_m[\sigma](con\ \overrightarrow{x} \texttt{=>} e) \equiv \begin{cases} con\ \overrightarrow{x}' \texttt{=>}\ (S[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](e)) & (\overrightarrow{x} \cap I(\sigma) \neq \phi) \\ con\ \overrightarrow{x} \texttt{=>}\ (S[\sigma](e)) & (\overrightarrow{x} \cap I(\sigma) = \phi) \end{cases}$$

(where the $\overrightarrow{x}'$ are names not in $\overrightarrow{x} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(e)$ and $S[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']]$ is defined to map each $x \in \overrightarrow{x}$ to the corresponding member of $\overrightarrow{x}'$).

$$S_m[\sigma](con \texttt{=>} e \;\texttt{where}\; \overrightarrow{gd}\,) \equiv con \texttt{=>} (S[\sigma](e)) \;\texttt{where}\; S_g[\sigma](\overrightarrow{gd})$$

$$S_m[\sigma](con\ \overrightarrow{x} \texttt{=>} e \;\texttt{where}\; \overrightarrow{gd}\,) \equiv \begin{cases} con\ \overrightarrow{x}' \texttt{=>} & (\overrightarrow{x} \cap I(\sigma) \neq \phi) \\ \quad (S[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](e)) & \\ \qquad \texttt{where} & \\ \qquad S_g[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](\overrightarrow{gd}) & \\ con\ \overrightarrow{x} \texttt{=>} & (\overrightarrow{x} \cap I(\sigma) = \phi) \\ \quad (S[\sigma](e)) & \\ \qquad \texttt{where}\; S_g[\sigma](\overrightarrow{gd}) & \end{cases}$$

(where the $\overrightarrow{x}'$ are names not in $\overrightarrow{x} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(e) \cup \mathrm{fv}(\overrightarrow{gd})$).

$$S_m[\sigma]((\ \overrightarrow{x}\ ) \texttt{=>} e) \equiv \begin{cases} (\ \overrightarrow{x}'\ ) \texttt{=>}\ (S[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](e)) & (\overrightarrow{x} \cap I(\sigma) \neq \phi) \\ (\ \overrightarrow{x}\ ) \texttt{=>}\ (S[\sigma](e)) & (\overrightarrow{x} \cap I(\sigma) = \phi) \end{cases}$$

(where the $\overrightarrow{x}'$ are names not in $\overrightarrow{x} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(e)$).

$$S_m[\sigma]((\ \overrightarrow{x}\ ) \texttt{=>} e \;\texttt{where}\; \overrightarrow{gd}\,) \equiv \begin{cases} (\ \overrightarrow{x}'\ ) \texttt{=>} & (\overrightarrow{x} \cap I(\sigma) \neq \phi) \\ \quad (S[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](e)) & \\ \quad \texttt{where} & \\ \quad S_g[\sigma[\overrightarrow{x} \mapsto \overrightarrow{x}']](\overrightarrow{gd}) & \\ (\ \overrightarrow{x}\ ) \texttt{=>} & (\overrightarrow{x} \cap I(\sigma) = \phi) \\ \quad (S[\sigma](e)) & \\ \quad \texttt{where}\; S_g[\sigma](\overrightarrow{gd}) & \end{cases}$$

(where the $\overrightarrow{x}'$ are names not in $\overrightarrow{x} \cup \mathrm{fv}(\sigma) \cup \mathrm{fv}(e) \cup \mathrm{fv}(\overrightarrow{gd})$).

Figure C.4: Substitution on core matches.

$$\frac{v \Downarrow_{O_V} v'}{v \Downarrow_O v'} \qquad (\Downarrow_O \text{val})$$

$$\frac{e \Downarrow_O e'}{\texttt{new}\ x\ \texttt{in}\ e \Downarrow_O \texttt{new}\ x\ \texttt{in}\ e'} \qquad (\Downarrow_O \text{new})$$

$$\frac{e \Downarrow_O e'' \quad e' \Downarrow_O e'''}{\texttt{let}\ x = e\ \texttt{in}\ e' \Downarrow_O \texttt{let}\ x = e''\ \texttt{in}\ e'''} \qquad (\Downarrow_O \text{let})$$

$$\frac{v \Downarrow_{O_V} v'}{v\ @\ x \Downarrow_O v'\ @\ x} \qquad (\Downarrow_O \text{conc})$$

$$\frac{v \Downarrow_{O_V} v'' \quad v' \Downarrow_{O_V} v'''}{v\ v' \Downarrow_O v''\ v'''} \qquad (\Downarrow_O \text{app})$$

$$\frac{e \Downarrow_O e'}{(e) \Downarrow_O e'} \qquad (\Downarrow_O \text{tup}_1)$$

$$\frac{e_1 \Downarrow_O e'_1 \ \ldots \quad e_k \Downarrow_O e'_k}{(e_1, \ldots, e_k) \Downarrow_O (e'_1, \ldots, e'_k)} \qquad (\Downarrow_O \text{tup}_2)$$

$$\frac{S[x/x'](e) \Downarrow_O e'}{\texttt{case}\ x\ \text{of}\ \{\ x' \texttt{=>}\ e\ \} \Downarrow_O e'} \qquad (\Downarrow_O \text{case}_1)$$

$$\texttt{case}\ x\ \text{of}\ \{\ x' \texttt{=>}\ e \mid \mathit{match}\ \} \Downarrow_O \mathit{REDUNDANT} \qquad (\Downarrow_O \text{case}_2)$$

$$\frac{\texttt{case}\ x\ \text{of}\ \{\ x'\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>}\ e\ \} \Downarrow_O e'}{\texttt{case}\ x\ \text{of}\ \{\ (x')\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>}\ e\ \} \Downarrow_O e'} \qquad (\Downarrow_O \text{case}_3)$$

$$\texttt{case}\ x\ \text{of}\ \{\ (x') \texttt{=>}\ e \mid \mathit{match}\ \} \Downarrow_O \mathit{REDUNDANT} \qquad (\Downarrow_O \text{case}_4)$$

$$\frac{v \Downarrow_{O_V} v' \quad \overrightarrow{m} \Downarrow_{O_M} \overrightarrow{m}'}{\texttt{case}\ v\ \text{of}\ \{\ \overrightarrow{m}\ \} \Downarrow_O \texttt{case}\ v'\ \text{of}\ \{\ \overrightarrow{m}'\ \}} \qquad (\Downarrow_O \text{case}_5)$$

(for `case` expressions not matching any other rule and where $\Downarrow_{O_M}$ is extended to sequences of matches in the obvious way).

Figure C.5: The optimisation relation for expressions.

$$x \Downarrow_{O_V} x \qquad (\Downarrow_{O_V}\ \text{vid})$$

$$con \Downarrow_{O_V} con \qquad (\Downarrow_{O_V}\ \text{con}_1)$$

$$\frac{v \Downarrow_{O_V} v'}{con\ v \Downarrow_{O_V} con\ v'} \qquad (\Downarrow_{O_V}\ \text{con}_2)$$

$$\frac{v \Downarrow_{O_V} v'}{x\ .\ v \Downarrow_{O_V} x\ .\ v'} \qquad (\Downarrow_{O_V}\ \text{abst})$$

$$\frac{e \Downarrow_{O} e'}{\texttt{fun}\ f\ \texttt{=}\ \{\ x \texttt{=>} e\ \} \Downarrow_{O_V} \texttt{fun}\ f\ \texttt{=}\ \{\ x \texttt{=>} e'\ \}} \qquad (\Downarrow_{O_V}\ \text{fun})$$

Figure C.6: The optimisation relation for values.

$$\frac{e \Downarrow_{O} e'}{\{\ x\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e\ \} \Downarrow_{O_M} \{\ x\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e'\ \}} \qquad (\Downarrow_{O_M}\ \text{vid})$$

$$\frac{e \Downarrow_{O} e'}{\{\ con\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e\ \} \Downarrow_{O_M} \{\ con\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e'\ \}} \qquad (\Downarrow_{O_M}\ \text{con}_1)$$

$$\frac{e \Downarrow_{O} e'}{\{\ con\ \overrightarrow{x}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e\ \} \Downarrow_{O_M} \{\ con\ \overrightarrow{x}\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e'\ \}} \qquad (\Downarrow_{O_M}\ \text{con}_2)$$

$$\frac{e \Downarrow_{O} e'}{\{\ (\ \overrightarrow{x}\ )\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e\ \} \Downarrow_{O_M} \{\ (\ \overrightarrow{x}\ )\ \langle\ \texttt{where}\ \overrightarrow{gd}\ \rangle \texttt{=>} e'\ \}} \qquad (\Downarrow_{O_M}\ \text{tup})$$

Figure C.7: The optimisation relation for matches.

Index

λ-calculus, 9
FreshML type system, 10
FreshML types, 41

Abstract syntax trees, 17
Abstraction types, 10
Anonymous functions, 31
Atoms, 10

Backup strategy, 23

Capture-avoiding substitution, 10, 69
Comments, 72
Concretions, 10
Conventions, 11
Core expression optimisation, 40
Core language, 18, 28

Datatype declarations, 27
Development environment, 21
Development model, 21

Encapsulation, 19
Environments, 55
Expression evaluation, 18, 55

Grammar, 79, 81

Implementation, 25
Infix operators, 72
Integers, 66
Interpreter structure, 15

Lexing and parsing, 17, 26
Lists, 67

Matches, 31
Modules, 16, 19

Operational semantics, 55, 73

Pretty-printing, 19
Project aims, 13
Prototype interpreter, 24

Red-black trees, 25, 27
Requirements, 13, 71
Restricted types, 42

Sample tests, 66
Substitution algorithm, 40

Testing, 19, 65
Theory, 20
Top level environment, 63
Top level typing context, 63
Type inference, 18, 41
Type variables, 72
Typing algorithm, 44
Typing judgement, 43

Version control, 23

Project proposal

The original project proposal is included on the following pages.

1 Introduction

The aim of this project is to produce the first implementation of a small language, FML, designed by Dr Andrew Pitts in the Computer Laboratory.

The language is an experimental language developed in association with Dr Pitts' research into methods of reasoning about portions of code containing variable-binding operations.

In the first instance it is intended to produce an interpreter taking input from a source file on disk. Later it is hoped to produce an interpretive shell into which FML programs can be typed and evaluated. (This is not entirely trivial due to the way the language type-checking behaves).

The project will be implemented in ML using Standard ML of New Jersey under Linux as the development environment.

The bulk of the project is the construction of the interpreter itself, but there is additional work to be done in preparing a concrete syntax and operational semantics for the language and in understanding the new concepts behind the language.

2 Special Resources

The only "special resource" is my own machine, which I shall be using for the project.

Machine Configuration

- 400MHz Pentium II processor
- 128Mb RAM
- 15Gb disk space
- 4Gb Seagate SCSI tape drive
- Linux operating system

Backing Up

The following precautions have been put in place to guard against hardware failure:

- Thrice-daily backups to Thor. The Thor filespace will hold a total of three days' snapshots (quota permitting).
- Daily backups to Pelican, holding snapshots from as long ago as disk quota permits.
- Daily backups to tape using my machine's tape drive.

3 Starting Point

At present my knowledge of ML is limited to that learnt in Part IA of the Computer Science Tripos. As the project is getting underway I shall be enhancing my knowledge of the language to include structured data types, the module system and other aspects to enable me to complete the project.

The project will also entail my learning how to use the ML-Lex and ML-Yacc tools to provide fast generation of an efficient parser in ML.

To start work on much of the project also requires my understanding of the FML language itself! The concepts behind the language are reasonably tricky to understand, and there is the added difficulty of having to understand a significant amount of the research papers relating to the subject. My knowledge of semantics and type systems are limited to the Part IB and Part II courses on those subjects, respectively.

This project is the first interpreter or compiler I have written and additional reading will be required to learn more about the implementation of functional languages, a topic which I knew nothing about before starting this project.

4 Project Description

The FML language is described, in the words of the designer, as "a meta language for programming with bound names modulo renaming". The language is similar to a subset of ML and has the key feature that the type system guarantees that well-typed programs are insensitive to renaming of bound names.

The aim of this project is to produce the first interpreter for this new language.

To get some idea of the concepts behind FML, consider a typical "naïve" ML function defining the operation of *capture-free substitution*, usually donated by something like $M[T/x]$, where M is a λ-term:

$$
\begin{array}{lll}
\textbf{datatype}\ \text{Lambda} = & \text{Var} & \textbf{of}\ \text{string} \\
 & \mid\ \text{App} & \textbf{of}\ \text{Lambda} \times \text{Lambda} \\
 & \mid\ \text{Abst} & \textbf{of}\ \text{string} \times \text{Lambda};
\end{array}
$$

$$
\begin{array}{ll}
\textbf{fun}\ \text{sub}\ T\ x\ (\text{Var}\ v) & = \textbf{if}\ v = x\ \textbf{then}\ T \\
 & \qquad\quad \textbf{else}\ \ \text{Var}(v) \\
\mid\ \ \text{sub}\ T\ x\ (\text{App}(abs, arg)) & = \text{App}(\text{sub}\ T\ x\ abs, \\
 & \qquad\quad \text{sub}\ T\ x\ arg) \\
\mid\ \ \text{sub}\ T\ x\ (\text{Abst}(bv, body)) & = \text{Abst}(bv, \text{sub}\ T\ x\ body);
\end{array}
$$

This may well fail (the failure manifesting as an incorrect result) due to the third clause when presented with input causing capture of free variables. To check for such problems requires a reasonable amount of additional code.

In FML it is easy to write a program insensitive to the names of the bound variables in the λ-terms being represented by the meta language; α-conversion is automatically performed by the language as appropriate to ensure a correct substitution function without the need for the writer to worry about variable capture, etc.

The project breaks down into several distinct stages, whose scheduled dates of work are given in the next chapter.

The initial stages consist of requirements analysis, understanding the FML language (including evaluation semantics and typing rules) and design of the interpreter structure. Issues such as the representation of the FML abstract syntax trees will need to be considered at this stage.

In addition a very small functional-style language will be implemented in the initial stages with just a couple of constructs, in order to gain more experience at implementing functional languages before the main coding starts.

Furthermore a concrete syntax of the FML language needs to be specified; the current research paper proposes a syntax which has some non-standard constructs which would more usefully be written slightly differently for ease of comprehension of FML code. The operational semantics specified in the paper also needs a more concrete specification (for example to use an environment rather than performing substitutions during function applications).

Some features need adding to the existing language to produce a useable interpreter, in particular integers and strings!

Major parts of the interpreter are:

- the top-level front end (including the reporting of error messages);
- the lexical analyser;
- the parser;
- the translator to convert to core FML;
- the type checker;
- the evaluator.

In the coding period, it is intended to start work on the abstract syntax tree definitions for FML and produce the translator to core FML, the type checker and the evaluator. These are the most complicated parts of the project and hence are scheduled ahead of easier parts such as the production of a lexer and parser.

Should time permit, there are some possible extensions:

- an interactive top-level shell;
- investigating good ways of presenting error messages caused, say, as a result of the input of an untypable program;
- implementation of ML-style `let`-bound polymorphism.

5 Schedule of Work

Date	Work
Thu 7 Oct Mon 11 Oct Fri 15 Oct	*Start of lectures* Phase I form deadline Draft proposal deadline
Fri 22 Oct	Proposal deadline **Start initial stages:** Requirements analysis, understanding FML typing and semantics, preparation of a concrete syntax and operational semantics, preparation of the development environment, implementation of a very simple functional language.
Fri 19 Nov	Aim to have initial preparatory work complete. **Coding:** Abstract syntax tree datatypes, start work on core translator.
Fri 3 Dec	*Full Term ends*
Wed 8 Dec Xmas - New Yr.	*Residence period ends* Aim to have translation done by now. Start on type inference. *Holiday*
Thu 13 Jan	*Residence period starts* Aim to have type inference algorithm working.
Tue 18 Jan	*Full Term starts* Start work on evaluator.
Fri 21 Jan Fri 28 Jan Fri 4 Feb	**Write progress report**. Have progress report finished. Aim to have evaluator finished by now. Start work on lexer and parser. **Progress report deadline**
Fri 18 Feb	Aim to have code finished. Get code to Dr Pitts. If time permits, write interactive shell and any other extensions.
Fri 3 Mar	Coding completed. Testing to start.
Fri 17 Mar	*Full Term ends.*

Wed 22 Mar	Testing to have been completed. Start writing dissertation. *Residence period ends*.
Easter Vac.	Continue writing dissertation.
Thu 20 Apr Tue 25 Apr Fri 5 May	*Residence period starts*. Get draft dissertation to Dr Pitts. *Full Term starts*. **Final dissertation to Dr Pitts** (at the latest).
	Final corrections. Printing and binding.
Fri 12 May	**Dissertation completed and handed in.**
Fri 19 May	**Dissertation deadline**.