Master thesis

The SLinks Language

Gilles Dubochet

Under the supervision of Philip Wadler, university of Edinburgh
and Martin Odersky, école polytechnique fédérale de Lausanne.
Presented in February 2005.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Abstract

This report presents a prototype interpreter for a simple functional language, called SLinks. The focus of this work is two-fold. One side is the design of a type inference system, based on the Hindley-Milner algorithm, with additional support for record and variant types. This type system is similar to row variable based systems as described by Rémy and Wand. The other side is the support of database querying from within the language. In particular, the focus is on automatic optimisation of SLinks expressions into SQL queries.

The resulting prototype, that tested and will be used to test techniques for Philip Wadler's upcoming Links language, was heavily inspired by Wong's Kleisi and CPL language. It does however extends it in various ways, in particular as far as record and variant operators are handled in typing and SQL optimisation.

Contents

Chapter 1

Introduction

This report presents the work I have done for my master's project under the supervision of professor Philip Wadler at the University of Edinburgh and of professor Martin Odersky. During the six months of the project I developed a simple but complete interpreter for a functional language called SLinks. SLinks stands for 'Simple Links' as it is used to prototype simple ideas for a 'Links' language that Wadler is developing and that is described in [1].

The primary goal of the project was to study how database access can be integrated transparently in a programming language without too much of a performance hit. The inspiration for such a technique very strongly comes from Limsoon Wong's Kleisli system and CPL language described in [2]. However, the goal of SLinks is different from Kleisli's: if the latter is purely a database query language, albeit very powerful and polyvalent, SLinks is the prototype for a general-purpose programming language that cannot make as many assumptions about the kind of expressions it must solve as a query language can.

Since SLinks was the first prototype for the Links language, the infrastructure to build and test the database access part did not exist at all. This is why a large part of the time of this project was used to put in place the infrastructure for the prototype interpreter. This is not to say that the development of the interpreter system was secondary. On the contrary, the type inference system in particular is a complex and very interesting system based on modern research. On a more personal side, the development of this type inference system was a very interesting activity as it was something completely new for me. In summary, the research and development done during this project can be divided into three major parts:

1. An interpreter for a simple but powerful programming language. The language is not particularly new by itself, it is based on lambda calculus with records and variants, but it also supports some other interest-

ing features such as collection comprehensions and of course database access operators. In short, SLinks is not a revolutionary language but is interesting and well suited to prototype the database access system integrated in such a functional language.

2. A type inference system for that language. This system is based on known research, but its integration with the database access part of the language is new: Kleisli's type system is simpler than Slinks'. In particular the records it supports are extensible which makes a big difference in the way database optimisations must be handled.

3. Database access and access optimisation. The optimisation algorithms are based on the work done in Kleisi, but because of the advanced types SLinks supports, the algorithms need to tackle different problems that Kleiski does not have. Research has also been done to find simpler ways to write these optimisations using type information but the results are not fully satisfactory. Database access is limited to relational SQL databases for now.

This report is subdivided into four major chapters. Chapters 2 and 3 are overview chapters that describe the environment created to support the research work.

Chapter 2 presents an overview of the SLinks language. It explains what the various operators are and how they are used. It also gives some insight about how the language is handled internally in the compiler. Examples of small programs in SLinks are given.

Chapter 3 is a short overview of the architecture of the SLinks interpreter. Some insight is given into parts of the system that are not treated in other chapters such as the front-end and the interpreter itself. This chapter also presents the major data structures used in the program.

Chapter 4 speaks about how types are used in SLinks and how they are calculated for a program.

Chapter 5 describes the way database access has been implemented in SLinks. It then describes in detail various optimisations on the code to maximise the efficiency of database-related operations.

To conclude, I would like to thank Philip Wadler for giving me the opportunity to come to Edinburgh and work on this fascinating subject. I would also like to thank Jeremy Yallop for the discussions we had about various interesting problems of computer science.

Chapter 2

The SLinks language

This chapter presents the syntax of the SLinks languages. In the first section of this chapter a grammar for the language is given and the basic operators are described. The presentation of this syntax is pretty verbose, and a reader familiar with lambda calculus with records and variants will probably not learn much. Since a good understanding of the syntax will greatly simplify the understanding of latter chapters I thought it was important to define it precisely. In the second section, another simpler syntax is presented that is used for internal computation. Transformation rules from one syntax to the other are provided. Finally, some examples of SLinks code are given.

SLinks is a small functional programming language based on the lambda calculus, but extended with various useful operators. Amongst the more interesting extensions, variants and records have been added to test a type inference system that supports them. Database operators, collections and list comprehensions have been added too, and are used to test SQL optimisations. SLinks is in essence very similar to Kleisli's CPL programming language, described in [2], but with more powerful record and variant operators.

SLinks is also a limited language. There are plenty of useful features missing to make it really usable. One of the most limiting misses, in my opinion, is the lack of recursive types which prevents any mutually recursive data structure such as arbitrary depth trees, linked lists (there are normal lists, though) etc. Another problem with the current version of SLinks is its syntax which is cumbersome in many regards. However, since this language is intended to be used as a prototype only, this work was not deemed necessary.

`e`	:= `if e then e else e`	Branch
	| `e + e`	Integer addition
	| `e - e`	Integer subtraction
	| `e * e`	Integer multiplication
	| `e / e`	Integer division
	| `e ++ e`	Floating-point addition
	| `e -- e`	Floating-point subtraction
	| `e ** e`	Floating-point multiplication
	| `e // e`	Floating-point division
	| `e ^^ e`	Floating-point power
	| `e & e`	String concatenation
	| `e == e`	Polymorphic equal
	| `e <= e`	Polymorphic less or equal
	| `e >= e`	Polymorphic more or equal
	| `e << e`	Polymorphic less
	| `e >> e`	Polymorphic more
	| `e <> e`	Polymorphic not equal

Figure 2.1: Additional sweet operators

2.1 Sweet syntax

SLinks' sweet syntax – so called because it is a sugared version of the internal compiler syntax – is the user syntax of SLinks. This section will present and explain the grammar for a sweet SLinks expression, the grammar for patterns and some additional binding operators. To conclude, the user interface of the SLinks interpreter will briefly be described. Some common operators will not be discussed in this chapter and are listed in figure 2.1.

Constant values

`e := c`

To start with, an expression can be a constant value '`c`'. A constant can be of one of the following types:

Boolean constants are either '`true`' or '`false`'.

Integer constants are base 10 natural numbers, positive or negative. Calculated integers have an unlimited range, however the range for in-line constant integers is limited and goes from -1073741824 to 1073741823, inclusive. Values '`4`', '`-2`', '`0`' and '`234582`' are integers; '`23.1`' and '`+11`' are not.

Float constants are base 10 real numbers, positive or negative. An n-plex multiplier can be specified after an '`e`' character. The range of values for float constant is infinite, but the precision is not: there is no guarantee that all digits provided will be used. Values '`4.`', '`-2.23`', '`-11.e-8`' and '`0.`' are all floats; '`23`' and '`14.e13.2`' are not.

String constants are arrays of characters encoded in ISO-8859-1 format.

Functions

```
e  :=  fun (p, ... p) -> e
     | e (e, ... e)
```

SLinks is a functional language and functions can be defined like any other value. The first of the above expressions is the declaration of a function (or lambda abstraction) with one or more parameters. Parameters '`p`' are defined by patterns – the syntax of which is described later in this section. A function with more than one parameter will automatically be curried. The second expression is the application of a function to a value. If the number of arguments is less than the number of parameters of the function, it will be partially applied.

Bindings and variables

```
e  :=  let p = e in e
     | letrec p = e, ... p = e in e
     | x
```

Local '`let`' definitions bind the value of an expression to a name in the body of the expression. Such definitions are not recursive, in the sense that the binding is not available in the value. Local '`letrec`' recursive definitions on the other hand allow all defined bindings to be used in all values. Recursive definition however requires all defined values to be functions, and is therefore only used to define recursive or mutually recursive functions. Limiting recursive declaration to function makes it possible to prevent the declaration of recursive data that the type system does not support

A variable '`x`' will take the same value as the latest value bound to its name by either a local definition, a global definition – described below – or a function.

Variables and binding names in definitions are defined as a string of lowercase and uppercase letters, digits, and underscores, not starting with a digit. Values '`tata`', '`ta_ta`', '`TatA`' and '`tat9a`' are names; '`9tata`' and '`ta-ta`' are not.

Records

```
e := {}
   | {l = e, ... l = e}
   | {l = e, ... l = e | e}
   | {e, ... e}
   | e.l
```

Records are unordered sets of labelled expressions, called fields. The label of each field must be different from all other labels in the record. The first expression above is the constructor for a record with no fields. The second expression is the constructor for a record with an arbitrary number of fields. The third expression is the extension of an existing record expression with an arbitrary number of additional fields. Since records are unordered, the constructors are commutative.

The record without labels, called a tuple, can be used whenever the additional power of records is not used. Obviously, the order of fields is important for this operator.

To access the content of a record, pattern matching is usually used. The last expression makes it possible to retrieve the value of a specified field from a record. However, the pattern matching mechanism allows to do the same in a much more powerful way, and this operator is only provided to simplify writing: it does not offer additional expressive power. Fields in a tuple can be accessed in the same way as fields in a record by using the integer position of the field in the tuple as its label. Position counting starts with 1.

A label must follow the same naming rules as a variable, but is always preceded by a '`#`' character. Values '`#tata`' and '`#ta_9ta`' are labels.

Variants

```
e := <l=p>
   | case e of <l=p> in e or ... <l=p> in e
   | case e of <l=p> in e or ... <l=p> in e | p in e
   | case e
```

Variants are constructs that link a value to a particular label. This encapsulation is done with the first operator. The next three operators select the execution of a specified sub-expression depending on the label of a variant value. These operators work as follows.

A variant value is provided as the '`case`' parameter for the operation. Then, depending on the label of the variant value, the corresponding '`in`' expression is evaluated, with the value of the variant bound to the corresponding '`p`' pattern. Each of the different available evaluation expressions is a called a case.

The open case operator has a final default case, after the '`|`' character. This case will be executed if no earlier case matched the label of the value. The binding pattern '`p`' will be bound to the original variant value.

Finally, the last operator, the closed case operator, is normally not useful as such for syntactic reasons, but is required to ensure type safety. This will be described in more detail in chapter 4. For an intuitive understanding of the closed case '`case e`' operator, it can be considered as an operator that guarantees that all possible labels that the variant value can take have been treated by earlier case operators. '`case <#a=4>`' will therefore fail: this operator only makes sense as part of the otherwise expression of an open case operator.

Collections

```
e := [bag]
   | [bag e, ... e]
   | e :bag: e
   | [set]
   | [set e, ... e]
   | e :set: e
   | [lst]
   | [lst e, ... e]
   | e :lst: e
```

Collections come in three different kinds: sets, bags and lists. Bags are unordered collections of expressions; sets are unordered and unique collections; lists are ordered collections. An expression is said to be unique if its calculated value is structurally different from any other value in the collection. A function value is always unique.

The above expressions in square brackets are the empty collection and the collection with an arbitrary number of elements for the three types of collections. '`:bag:`' merges two bags, which yields a bag containing all elements of both bags; '`:set:`' unions two sets, which yields a set containing all elements that exist in both sets, that is elements that would be non-unique if the two sets were merged; '`:lst:`' concatenates two lists.

Comprehensions

```
e := [bag e | f, ... f]
   | [set e | f, ... f]
   | [lst e | f, ... f]
f := pp <bag e
   | pp <set e
   | pp <lst e
   | e
```

SLinks supports collection comprehensions to modify or filter the content of collections. Comprehensions are described in detail in [3] and work as follows. A comprehension is an operator that returns a collection. One comprehension exists for every three types of collections supported by SLinks. The right part of the comprehension operator – after the vertical bar – contains a number of fields which define the origin of the data for the new collection. These fields are evaluated from left to right and can be either comprehension bindings or conditions.

A binding such as '`p <set e`', can be seen as a 'foreach' loop on a collection '`e`', where every element of the collection is successively pattern matched on '`p`'. When more than one binding field is present, the result will be that the right-most binding will be executed as an inner loop for the next binding on its left, and so on.

A condition filters elements of the collection resulting from the evaluation of the fields at its left. It only keeps elements that evaluate to true for this condition.

The left part of the operator – before the vertical bar – will be executed for every element of the resulting collection from the right part, and the result of it will be the value for the corresponding element in the returned value.

To better understand the meaning of a comprehension, here is a little example of the transformation of such an expression into more traditional 'foreach' loops and filter statements. The following comprehension and the expression in pseudo-code are equivalent. The result for this calculation will be '`[bag 6, 7, 7, 8]`'.

```
[bag 2+x+y | ^x <bag [bag 1, 2], ^y <bag [bag 3, 4, 5], y<<5]
```

```
foreach ^x in [bag 1, 2] do
  foreach ^y in (filter (fun x -> x << 5) [bag 3, 4, 5]) do
    2+x+y
```

In a more database-oriented fashion, it can be seen as a jointure between all the bindings' collections, and filtered by all conditions. In that definition, the above example comprehension would be represented in SQL-like code like this.

```
SELECT 2+x+y FROM (1, 2) AS x, (3, 4, 5) AS y WHERE y < 5
```

Finally, comprehensions also allow transformations between different types of collections. A bag comprehension with a binding field on a list for example will return a bag from the data of the list. Collection transformation are not allowed in all cases: only the transformation of a list in a set or a bag, of a set in a bag and of a bag in a set are allowed.

Tables

```
e := database e
   | table n with m from e
   | table n with m order o from e
   | table n with m unique from e
   | table n with m unique order o from e
   | sort_up (e)
   | sort_down (e)
```

These operators provide access to a database. The first operator is a connection to a database, where the sub-expression '`e`' is a record with the connection parameters. The connection parameters may vary from one database sub-layer to another. The current PostgreSQL database connection expects a record with the following labels: '`#name`', '`#host`', '`#port`', '`#user`', '`#pass`' (word), all of them as strings. If one or more labels are not provided, the system will replace them with default values; if more labels are provided, they will be ignored.

The next four operators yield data from a relation in a database. The first, simplest, operator simply returns all elements (called rows in relational database parlance) from a relation named '`n`'. The columns for the relation must be defined by the programmer, since they are required to guarantee type safety – as described in chapter 4. To do this, a model '`m`' must be provided. The syntax of such a model is equivalent to the type of the record that represents this row. Since type syntax will be defined later, a generic example of what such a model should look like will be given instead. For a relation with columns 'a', 'b' and 'c' of type integer, float and string respectively, the model will be '`{#a:int,#b:float,#c:string}`'. Note that the label name – that is, the label without the '`#`' in front – corresponds to the name of the columns in the database relation. If label names that do not correspond to column names are used, the system will fail at runtime. It is acceptable however not to define columns in the table's model: they will simply not be useable in the SLinks program. The result of the simple table operator is a bag of records, where every record is a record of the same type as the given model.

The additional '`unique`' argument removes all duplicate records from the resulting collection. A unique table is equivalent to an SQL query with a '`UNIQUE`' option. When the '`unique`' argument is present, the result value of the table operator is a set.

The additional '`order o`' argument allows an ordering to specified for the table. An ordering '`o`' could for example be '`[#a:asc,#b:desc]`'. In this example, the column 'a' will be ordered in ascending order, and for equal 'a' values, ordered in descending order on value 'b'. An ordered table is equivalent to an SQL query with an '`ORDER BY`' option on the corresponding

columns. The result value of a table operator with an '`order`' argument will always be a list, even if a '`unique`' argument is present too.

As an example of the table operator, consider the following database table, called 'tata'.

a	b
1	"one"
2	"two"

A simple example of the syntax of a table operator on this table, and the resulting value follows.

```
table "tata" with {#a:int,#b:string} from db
[bag {#a=1,#b="one"}, {#a=2,#b="two"}]
```

Table operators are in a very limited sense similar to an SQL '`SELECT`' statement in the scope of data they access. However, they lack the ability to filter the data returned by a table, or to join two tables together. Collection comprehensions are the tool that allows these transformation on table data and, as chapter 5 will explain, does this in an efficient way.

The sort operator finally sorts a collection into a list, either up or down. The sort operator is a somewhat limited operator since it does not allow the developer to specify the ordering. The real use of this operator is that it can automatically be optimised into a database query. More will be explained in chapter 5.

Transformations

```
e := float_of_int (e)
   | float_of_string (e)
   | int_of_string (e)
   | bool_of_string (e)
   | string_of_int (e)
   | string_of_float (e)
   | string_of_bool (e)
```

Because of the way SLinks handles types, it is not able to provide automatic type transformations. This is why a number of type transformation operators are needed.

Patterns

The sweet syntax of SLinks uses pattern matching when name bindings are required. Two different types of pattern are used: the simple pattern '`p`' is used for '`let`' and function definition; the conditional pattern '`pp`' is used for bindings in collection comprehensions. The grammar for both types of pattern is defined in figure 2.2.

```
 p  :=  ^x
      | ^{l=p, ... l=p}
      | ^{l=p, ... l=p | p}
      | ^{}
pp  :=  c
      | x
      | ^x
      | _
      | ^x&pp
      | ^{l=pp, ... l=pp}
      | ^{l=pp, ... l=pp | pp}
      | ^{}
```

Figure 2.2: Patterns

The simple pattern is pretty straightforward. It either binds a value directly to a name or, if the value is a record, to bind a number of selected fields each to a different name. The open record binding '`{l=p, ... l=p | p}`' binds the remainer of the record to the last pattern, called a row variable. The 'remainer' means the record with all fields that have not been bound in the left part of the operator. This construct is very useful as it allows the useage of only some fields in a record, without specifying what the remaining fields are. This allows a form of structural polymorphism very similar to what object-oriented programming languages allow. More about that can be found in chapter 4.

Conditional patterns are used in collection comprehensions. Normally, with this operator, bindings and conditions are separated, but conditional bindings, as their name implies, allow a condition to be specified at the same time as a binding. In a pattern, all binding variables are preceded with a '`^`' character. Non-binding variables (replaced by their value at runtime), or constants will be used as the condition, in the sense that only elements of the collection whose values correspond to the value in the pattern will be used; others will be dropped. As an example, the two following expressions are equivalent.

```
[bag x | ^{#x=^x,#y=4} <set value]
[bag x | ^{#x=^x,#y=^y} <set value, y == 4]
```

The `^x&pp` pattern allows to bind a value that will then be pattern matched further. For example '`^var&4`' requires the value to be 4, and binds it to '`var`'. The '`_`' operator is a wild-card character that binds a value to nothing.

Interpreter specific operators

The current version of SLinks is not a compiler, but an interpreter. For the purpose of the work that has been done on SLinks, this is exactly the same thing: the interesting parts of this work are not in the back-end (code generation or interpretation) but in the optimiser and front-end. The syntax described earlier in this section is sufficient to write any SLinks program but, in order to facilitate the usage of the interpreter, it has been slightly extended as explained below.

The interpreter is a looping process that takes as input an expression and returns the value that results from evaluating it. An expression is always terminated by a double semicolon '`;;`'. With the basic syntax, an evaluated expression will be 'lost' once evaluated and its result will not be available for further expressions. This is why two special operators, the '`def ^x = e`' and '`defrec ^x = e`' operators have been added. They are similar to the '`let`' and '`letrec`' where the body – the part after the '`in`' – is all expressions that are interpreted later. The '`defrec`' operator is limited in the sense that it only allows one binding and both are limited to simple bindings without patterns. Here is a simple example of an interpreter session.

```
? def ^four = 4;;
  Defined four as 4 : int
? four + 5;;
  9 : int
```

2.2 Core syntax

The user syntax described in the previous section is designed to be reasonably simple to write for a human programmer. However, this simplicity comes at the cost of decreased regularity and additional complexity. For the compiler or interpreter a more regular, smaller grammar is more suitable. This is why the sweet syntax is internally transformed into a similar but simpler core syntax.

Core syntax only has operators with a fixed number of arguments (except for '`letrec`') and has no pattern matching. The grammar for a core syntax expression is described in figure 2.3. For collection operators, a generic '`col`' collection type has been used in the grammar, but of course a different operator exists for bags, sets and lists every time. Operations on integers, floating-point numbers and strings, as well as conversion operators exist of course also in core grammar. In general, the core operators are similar enough to the operators described in the preceding section, and will not be defined in more detail. Some operations should be described in additional detail though:

$\mathtt{e}_c$	:=	`c`	constant
	|	`x`	variable
	|	`fun ^x -> `$\mathtt{e}_c$	abstraction
	|	$\mathtt{e}_c$`(`$\mathtt{e}_c$`)`	application
	|	`if `$\mathtt{e}_c$` then `$\mathtt{e}_c$` else `$\mathtt{e}_c$	branch
	|	`let ^x = `$\mathtt{e}_c$` in `$\mathtt{e}_c$	local definition
	|	`{}`	empty record
	|	`{l=`$\mathtt{e}_c$`|`$\mathtt{e}_c$`}`	record extension
	|	`let {l=^x|^x} = `$\mathtt{e}_c$` in `$\mathtt{e}_c$	record selection
	|	`let {} = `$\mathtt{e}_c$` in `$\mathtt{e}_c$	empy record selection
	|	`<l=`$\mathtt{e}_c$`>`$\rangle$	variant injection
	|	`case `$\mathtt{e}_c$` of <l=^x> in `$\mathtt{e}_c$`|^x in `$\mathtt{e}_c$	variant selection
	|	`case `$\mathtt{e}_c$	closed variant selection
	|	`[col]`	empty collection
	|	`[col `$\mathtt{e}_c$`]`	single element collection
	|	$\mathtt{e}_c$` :col: `$\mathtt{e}_c$	collection union
	|	`for ^x <col `$\mathtt{e}_c$` in `$\mathtt{e}_c$	collection loop
	|	`database `$\mathtt{e}_c$	database
	|	`table 'SQL' from `$\mathtt{e}_c$	database table

Figure 2.3: Core grammar

The collection loop replaces the collection comprehension. This operator really is a one-binding comprehension, without conditions – they will be replaced by another mechanism as described in the next section. Its syntax is changed to a '`for`' loop to better differentiate it from a full-fledged comprehension. The result of such a collection loop will be a collection obtained by merging all the collections calculated in the '`in`' body for every field in the original collection. In particular, and this will be important later on, if for one field, the body returns '`[col]`' (the empty collection), the resulting collection will drop this field and return a collection at least one smaller than the original one.

The database table is also quite different in the sense that, instead of hiding the actual database query, it exposes the SQL statement that will be used. This is somewhat limited since it restricts this operator to relational databases (no XQuery databases for example). But since this prototype only considered the problem of relational databases, this is no particular problem. It also makes the core syntax more useful when trying to understand what is going on in later optimisation phases.

The record selection and empty record selection are new operators that will be used extensively to replace the pattern matching mechanism. Record selection binds a name to the value of a field for the given label, and the remaining of the record's fields to another called the 'row variable'. The empty record selection guarantees that its value is an empty record. It is used in a similar way to the variant's closed case operator to ensure type safety, as described later in this report.

Core syntax is of course not used internally as a string language, but is represented using a tree data structure. The internal representation of the core language will be described in more detail in chapter 3.

2.3 From sweet to core syntax

The transformation from sweet to core syntax is a pretty straightforward mapping. These transformations are defined using transformation rules. Transformation rules define, for a given expression in sweet syntax at the top, in what expression in core syntax it will be transformed. Transformation rules are applied recursively.

Firstly record constructors are transformed with simple rules that simply replace record definitions with multiple fields by a string of nested one field record extensions. In the case of a closed record, the last extension is an empty record.

$$\frac{\texttt{\{l}_1\texttt{=e}_1\texttt{, ... l}_n\texttt{=e}_n\texttt{|e}_r\texttt{\}}}{\texttt{\{l}_1\texttt{=e}_1\texttt{| ... \{l}_n\texttt{=e}_n\texttt{|e}_r\texttt{\}\}}}$$

$$\frac{\texttt{\{l}_1\texttt{= e}_1\texttt{, ... l}_n\texttt{= e}_n\texttt{\}}}{\texttt{\{l}_1\texttt{= e}_1\texttt{| ... \{l}_n\texttt{= e}_n\texttt{|\{\}\}\}}}$$

Case operators are handled in a very similar way to records. The need for the closed case operator, that was somewhat unconvincingly described as necessary earlier in this chapter, now becomes quite evident, even without the description of the type system. The variables called '`@`' are new unique names generated by the syntactic sugar mechanism.

$$\frac{\texttt{case e of <l}_1\texttt{=x}_1\texttt{> in e}_1\texttt{ or ... <l}_n\texttt{=x}_n\texttt{> in e}_n}{\begin{array}{l}\texttt{case e of <l}_1\texttt{=x}_1\texttt{> in e}_1\texttt{|@}_1\texttt{ in ...}\\ \quad\texttt{case @}_{n-1}\texttt{ of <l}_n\texttt{=x}_n\texttt{> in e}_n\texttt{|@}_n\texttt{ in}\\ \quad\quad\texttt{case @}_n\end{array}}$$

$$\frac{\texttt{case e of <l}_1\texttt{=x}_1\texttt{> in e}_1\texttt{, ... <l}_n\texttt{=x}_n\texttt{> in e}_n\texttt{ | x}_r\texttt{ in e}_r}{\begin{array}{l}\texttt{case e of <l}_1\texttt{=x}_1\texttt{> in e}_1\texttt{|@}_1\texttt{ in ...}\\ \quad\texttt{case @}_{n-1}\texttt{ of <l}_n\texttt{=x}_n\texttt{> in e}_n\texttt{|x}_r\texttt{ in e}_r\end{array}}$$

Syntactic sugar for collections comes in two different flavours. One is simply for static collection values, where a multi-element collection is replaced by the union of many single-element collections, similar in principle to what is done for records or variants. The other one concerns collection comprehensions and their transformation into '`for`' collection loops. For this the principle is to take recursively the leftmost field in the right-part of the comprehension and then depending on its type, do the following:

For collection bindings transform it into a collection loop where the body of the loop is the comprehension without that field.

For conditions the body of the comprehension is replaced by a branch operator on the condition. The '`then`' branch is replaced by the old body expression in a single-element collection operator and the '`else`' branch is replaced by an empty collection. That way, if you remember what happens when an empty collection is returned by the body of a collection loop, all elements that do not fit the condition will simply be dropped from the final collection.

Finally, to terminate the recursion, when the right-part is empty, the collection comprehension is replaced by the corresponding one element collection.

$$\frac{\texttt{[col e}_1\texttt{, ... e}_n\texttt{]}}{\texttt{[col e}_1\texttt{] :col: ... :col: [col e}_n\texttt{]}}$$

$$\frac{\texttt{[col e|pp <col e}_b\texttt{, ...]}}{\texttt{for pp <col e}_b\texttt{ in [col e|...]}}$$

$$\frac{\texttt{[col e|...}_1\ \texttt{e}_c\texttt{, ...}_2\texttt{]}}{\begin{array}{l}\texttt{for ...}_1\texttt{ in}\\ \quad\texttt{if e}_c\texttt{ then}\\ \quad\quad\texttt{[col e|...}_2\texttt{]}\\ \quad\texttt{else [col]}\end{array}}$$

$$\frac{\texttt{[col e|]}}{\texttt{[col e]}}$$

The transformation for table operators is pretty straightforward. It simply creates a core syntax '`table`' operator with a simple '`SELECT`' query. The columns to be queried are inferred from the model provided in the sweet operator. The set of names of the columns from the model is written as 'labels(m)' for a given model '`m`' below. If an '`order`' or '`unique`' argument is present, the query is extended with either an '`ORDER`' or '`GROUP BY`' condition. From these transformations, it might seem that the type information provided by the developer are lost. This is not the case, they are kept in the type field in the internal representation of the code. Then, a hack is used in the type inference system to insert the type into the inference mechanism. This will be detailed more thoroughly in chapter 4

$$\frac{\texttt{table n with m from e}}{\texttt{table 'SELECT labels(m) FROM n' from e}}$$

$$\frac{\texttt{table n with m order o from e}}{\texttt{table 'SELECT labels(m) FROM n ORDER BY o' from e}}$$

$$\frac{\texttt{table n with m unique from e}}{\texttt{table 'SELECT UNIQUE labels(m) FROM n' from e}}$$

$$\frac{\texttt{table n with m unique order o from e}}{\texttt{table 'SELECT UNIQUE labels(m) FROM n ORDER BY o' from e}}$$

Pattern matching is replaced by binding the variables in the pattern matching, in the body of the operator, using record selection operators. The transformation rules below work on the '`let`' operator, but the principle is the same for function declarations, etc.

$$\frac{\texttt{let ^\{l}_1\texttt{=^x}_1\texttt{, ... l}_n\texttt{=^x}_n\texttt{|^x}_r\texttt{\} = e in e}_b}{\begin{array}{l}\texttt{let ^\{l}_1\texttt{=^x}_1\texttt{|^@}_1\texttt{\} = e in ...}\\ \quad\texttt{let ^\{l}_n\texttt{=^x}_n\texttt{|^@}_n\texttt{\} = @}_{n-1}\texttt{ in e}_b\end{array}}$$

$$\frac{\texttt{let ^\{l}_1\texttt{=^x}_1\texttt{, ... l}_n\texttt{=^x}_n\texttt{\} = e in e}_b}{\begin{array}{l}\texttt{let ^\{l}_1\texttt{=^x}_1\texttt{|^@}_1\texttt{\} = e in ...}\\ \quad\texttt{let ^\{l}_n\texttt{=^x}_n\texttt{|\{\}\} = @}_{n-1}\texttt{ in e}_b\end{array}}$$

The conditional bindings for collection comprehensions is not described as a transformation rule, but here is how it works: every constant of non-binding variable in the pattern is replaced by a binding variable, and an equality test is added to the comprehension between this new binding variable and the test value.

The heavy dependance on syntactic sugar is a simple and rather efficient way of extending the grammar for a language without complicating the underlying interpreter. I must say however that it comes at a price too. Easy debugging becomes quite painstaking, either for the user, who does not receive his error messages on the same code than he wrote, or for the compiler developer who needs to add significant complexity to his program in order to remember to what original code de-sugared expressions correspond. Even in the latter case, the syntactic sugar is very difficult to hide from the end user. O'Caml, a language that is used for serious software production, has from time to time glitches in the reporting of errors because of its syntactic sugar.

No real effor was put in SLinks to try to solve the problem mentioned before. For that matter, the error reporting is SLinks is pretty much as atrocious as it can be, but better error reporting was not the goal of this project either.

2.4 Program examples

The following section presents a number of simple one expression example programs in SLinks. For database access examples, the tables will not be defined but are pretty self-describing.

Function application by adding 2 and 2 in a rather over-complicated way.

```
(fun ^x -> x+2)(2);;
```

Partial application of a function to obtain a function that adds 2 to any integer value.

```
(fun (^x, ^y) -> x+y)(2);;
```

Variants to define a – somewhat – polymorphic function to add 2 to an integer or floating point value.

```
fun ^x -> case ^x of
  <#int=^x> in x+2 or
  <#float=^x> in x++2.0;;
```

Pattern matching to define a new integer addition function that only takes a single parameter such as '`{#left=2,#right=2}`'.

```
fun ^{#left=^x,#right=^y} -> x+y;;
```

Row variables to modify a field in a record without any requirement for the remaining fields. This function will add 2 to the the integer value of the '`#x`' field. For '`{#x=1,#y=2}`' it will yield '`{#x=2,#y=2}`' and for '`{#x=1,#y=2,#z=2}`', '`{#x=2,#y=2,#z=2}`'.

```
fun ^{#x=^i | ^r} -> {#x=i+1 | r};;
```

Recursive binding to define a function to calculate the factorial of a value in a recursive way.

```
letrec ^fact = (fun ^x ->
  if (x<=2) then
    x
  else
    x * fact(x - 1);;
```

Set comprehension to filter and modify the content of a set. The resulting set is '`[set 3, 4]`'.

```
[set e + 2 | ^e <set [set 1, 2, 3, 4], e << 3];;
```

Conditional binding to filter the content of a list. The '`def`' statement is used to hold values between two expressions. The resulting list is '`[lst "one"]`'.

```
def ^list = [lst {#a=1,#b="one"}, {#a=2,#b="two"}];;
[lst b | ^{#a=1,#b=^b} <lst list];;
```

Comprehension binding on multiple collections. Notice how the set and bag expressions differ due to the inherent definition of these collection, where the bag cannot have duplicated elements. The result for the set expression is '`[set 2, 3, 3, 4]`' and for the bag expression '`[bag 2, 3, 4]`'.

```
[set a+b | ^a <set [set 1, 2], ^b <set [set 1, 2]];;
[bag a+b | ^a <bag [bag 1, 2], ^b <bag [bag 1, 2]];;
```

Database is defined for further use with some non-default connection parameters.

```
def ^db = database {#name="daba",#user="me"};;
```

Table is queried to get its content.

```
table "tata" with {#a:int,#b:string} from db;;
```

Comprehension is used to only select the 'a' column from a table.

```
[set x.#a |
  ^x <bag (table "tata" with {#a:int,#b:string} from db)
];;
```

Comprehension is used to only select some rows that have the value of field 'a' larger than 5 from a table.

```
[set x.#b |
  ^x <bag (table "tata" with {#a:int,#b:string} from db),
  x.#b >> 5
];;
```

Comprehension is used to join the content of two tables on a join condition. This expression returns a bag of tuples with an author and a list of publications from this author. This is a typical example that cannot be solved with an SQL query: the data model of SLinks offers a neat solution to this problem.

```
[bag {auth.#name,
      [lst pub.#title |
        ^pub <lst (table "publication" with {#publication_id:int,
                                             #author_id:int,
                                             #title:string}
                                        order [#title:asc] from db),
        pub.#author_id == auth.#author_id
]}|
  ^auth <bag (table "author" with {#author_id:int,
                                   #name:string}
                             from db)
];;
```

Chapter 3

SLinks system structure

This chapter will first describe the design of the SLinks system. Comments about the implementation language and important data structures will be made. Then the front-end – the parser, lexer and the syntactic sugar mechanism – and the back-end – the interpreter – will be explored in a little more detail. The two other major parts of the system, the type inference system and the optimisers, are covered in more detail in chapters 4 and 5 respectively.

This chapter should not be seen as a design documentation for SLinks but rather as some highlights into the development process. For more detail about SLinks, looking directly at the code is probably the best idea.

SLinks is written in O'Caml, a functional object-oriented high performance programming language described in [4]. It is also similar in many ways to SLinks: it is functional and supports variants very much, but not exactly, like those found in SLinks – records are much weaker though.

I was quite happy with O'Caml as a development language. Despite not being as pure as other functional languages such as Haskell, it nevertheless allows to develop very clean applications. Its impureness also allows simple access to features such as simple input-output or re-writeable variables. I did try to use such features as little as possible though, as it was part of the requirements for this project. O'Caml is also a pretty fast language, both fast compilation and fast execution. This is very agreeable when testing or debugging a program.

O'Caml supports a powerful module system. My system did not really require all the power the module system offers – in particular functors –, and it took me some time to get used to the way theses modules should be used. Actually, I had started in the first month to develop my application with O'Caml objects, but I found them very disappointing and the relationship between functional and object-oriented code awkward. My supervisor also wanted me to code in a purely functional way. Modules integrate in my

opinion much better with a functional language such as O'Caml. They are tricky at first though, since they seem in many ways similar to objects, but have limitations and possibilities that require a very different way of thinking about them.

3.1 System design

SLinks is based on a standard structure found in most compilers or interpreters. The system is divided into four major passes, executed sequentially for each input. The front-end reads the text based input and transforms it into a more practical data structure, the abstract syntax tree described later in this section. The type inference mechanism checks the type of the expressions in the program. The optimiser transforms the abstract syntax tree into another equivalent one that should improve the performance at execution. The last part is the back-end that either, in an interpreter, executes the abstract syntax tree and returns the calculated value or, in a compiler, transforms it into machine code to be executed later.

SLinks is an interpreter; its back-end calculates values directly. However SLinks is really a prototype for a compiler. This means that while the three first parts are usually quite lightweight in an interpreter, so as not to reduce the performance, this is not true in SLinks. Both the type inference algorithm and the optimisers are rather complicated functions that would make more sense in a compiler.

Getting the O'Caml module structure right took a little time because values in different modules cannot be dependant on each other – dependency is strictly hierarchical. At the beginning, each time I added a new element to the system, the module structure broke. Finally, I came up with a system that separates the declaration of data types and functions which more or less solved all problems. This is very unintuitive when one is accustomed to object-oriented programming but works very well once I got used to it.

Figure 3.1 shows the module hierarchy. Modules that define data types are represented as rounded grey boxes. Other modules that define functions are white square boxes. 'Utility', which contains useful functions in addition to the standard O'Caml library, and 'SLinks', the main module for the system, are depend or are required for more or less all other modules. Their dependencies have not been explicitly represented in order to simplify the figure.

The most important data structure in the system is the abstract syntax tree – described in module '`Sl_syntax`'. It is a representation of the core grammar for the language. The syntax tree type defines a node type for each different operator in the grammar. Each node type has a field for each

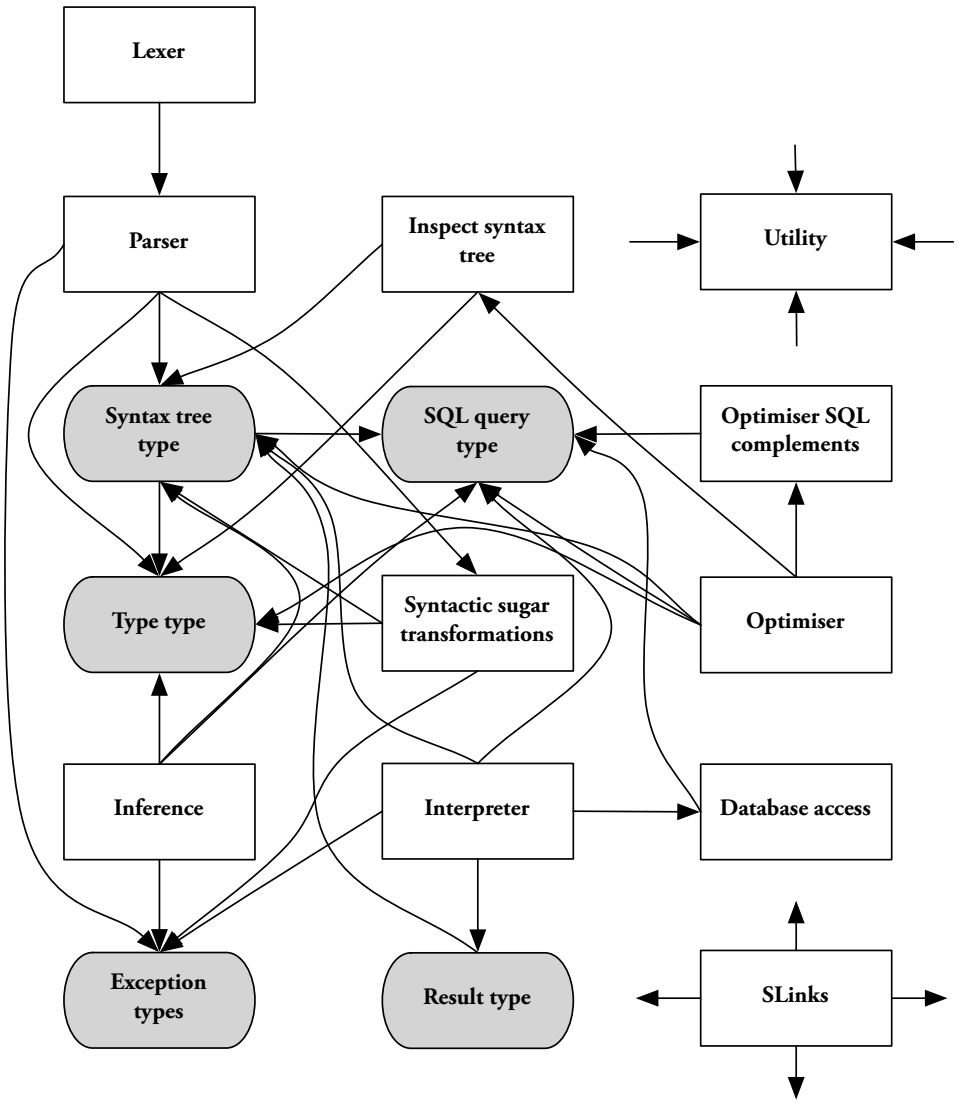

Figure 3.1: The module structure

parameter of the operator. This makes it extremely easy to modify the structure of an expression since each node is a sub-expression by itself that can easily be moved around or modified alone.

The type for each sub-expression is also attached to the corresponding node. In the second half of my project, when working on the database optimisations, I regretted this implementation choice: the type of a sub-expression is not really useful information for type based optimisations. The type of a binding would have been more interesting. Section 5.2 on page 49 explains the problem in more detail. However changing the syntax tree to keep the type information on bindings would have required extensive changes to many parts of the system, and in particular the inference algorithm. Since the problem appeared late in the project, I did not have the time to make a major overhaul of the structure and used various ways to circumvent the problem as explained in the section mentioned above. I would recommend to anyone wanting to make major work on SLinks in the future to consider changing it.

I think the general structure of the application is pretty much right. However for many sub-parts, if I had to restart the development now, I would have done it differently. But I suppose this was more or less unavoidable since I started my development without knowing exactly where it would go: I did not have enough knowledge about type inference and the database related optimisations to get it right the first time. Despite regular and extensive re-factoring, the design of the system always has been and still is one length behind what would really be needed.

3.2 Parser and syntactic sugar

The lexer and parser of SLinks are built around the automatic lexer and parser generator provided with O'Caml. These applications, 'ocamllex' and 'ocamlyacc' work in a similar fashion to the more common 'lex' and 'yacc'. This method is easy to put in place and works generally well. For parsing, the principle is basically that a number of rules are defined, each composed of a list of tokens. A token can either be a lexeme, called a terminal token, or a reference to a set of rules defined in the parser and called a non-terminal token. As soon as all tokens of a rule have been processed in the right order, the rule is fired and an attached operation is executed. In SLinks, this operation is either a syntactic sugar transformation, as described below, or a constructor for a syntax node.

When a rule in the parser that does not directly correspond to a syntax node is fired, it is necessary to transform it. Instead of calling the constructor for the syntax node, a special function is called instead that will return an

entire sub-expression in core syntax corresponding to the expression in sweet syntax that fired the rule. These transformation functions correspond to the transformation rules from sweet to core syntax described in section 2.3 on page 16.

If I had to re-implement the front-end now, I think I would have done it differently. There is a temptation to implement as much as possible of syntactic sugar transformations and type inference in the parser. However, despite being indeed easier for simple cases, this approach becomes problematic in more complex cases. The current approach in SLinks does the type inference as a separate task – at the very beginning I had tried to insert some limited type checking in the parser but abandoned the idea quickly – but includes syntactic sugar transformations into the parser. This approach has a few shortfalls.

- The 'ocamlyacc' parser is limited to $LR(1)$ grammars. Some of the strange features in the language, such as the '^' in front of the record in a binding, are due to the fact that they were required for the language to be $LR(1)$.
- Handling syntactic sugar directly during the parsing works but is not the simplest and cleanest solution. The parser, when parsing parts of an expression that will be transformed, must return a representation of this data for the transformation function that is specific to it. That means that the abstraction between parser and syntax transformers is not very good.

These two problems could be solved by using a much simpler parser that parses the input to a very generic syntax tree – where pattern and expressions are treated the same for example – and then transform this first tree into the final syntax tree. This would solve the first of the above problems or at least reduce it since a more generic tree is more likely to be in $LR(1)$ form. The second problem would disappear since the relationship between the parser and the syntax transformers would be defined solely through the syntax of the generic syntax tree. Syntax transformers would probably also be simpler as a tree transformation is a very natural form to represent such syntactic transformations.

3.3 Interpreter

The interpreter is a very straightforward mechanism that visits the abstract syntax tree bottom-up and returns the resulting value for each sub-expression. This means that the calculations are immediate; no laziness is provided which, as I explain in chapter 5, might not be optimal for database

operations. The interpreter is also an erasure interpreter. That means that no type information is used at runtime.

While visiting the syntax tree, the interpreter carries an environment with it that defines all variable bindings declared earlier in the tree. This simple environment is sufficient for the interpretation of single expressions. With the '`def`' and '`defrec`' pseudo-operators, the starting environment can be non-empty. This inter-expression environment is handled separately by the main function of the interpreter which is a more natural place for it to be handled.

As a side note and to conclude this chapter, the interpreter probably is the simplest part of the entire SLinks system. If I recall well, it took me an afternoon to implement and then five more minutes to correct the only bug it contained. It is quite disturbing [1] to think that this simple part is really the only one to provide results – actually the database also provides results but would also have been an easy part if I had found a good database library more quickly. All other parts that took me six month to master merely prepare for the actual calculation.

[1] Maybe not really disturbing but certainly a change in perspective about what a compiler really is

Chapter 4

Automatic type checking

This chapter will first introduce the rationale for using type inference to guarantee type safety. It will then describe the language used to represent types in SLinks. The algorithm to automatically infer a type will be presented and how the behaviour of this algorithm can formally be represented by type rules. Finally a complete description of the type rules for every operators will be provided.

The SLinks language grammar, as described in chapter 2 allows the developer to write a very broad range of programs ... most of them wrong. A correct expression in the syntactic sense carries no guarantee that it will not fail at runtime. For example, the following expressions, although syntactically correct, will fail or behave in unexpected ways at runtime.

```
1 + "hello"
(fun ^{#a=^a} -> a)({#b=4})
```

The reason for their failure is quite evident: they require the interpreter to calculate expressions which do not make any sense. A string cannot be added to an integer in any sensible way or the '`#a`' field cannot be extracted from a record that only contains the '`#b`' field. By checking the type of expressions during compilation it is possible to remove certain incorrect expressions at an early stage – that is, before they are executed. Type checking at compile-time is called static type checking as opposed to dynamic type checking which is done at runtime.

Static type checking does not guarantee that any compiled expression will execute without failing or misbehaving, a property that I will call static type safety. The type system that is used will influence on what classes of incorrect expressions are rejected by the compiler. The definition of a type system can incorporate such complex type information as the unit of a number, the value range for an expression or even, in a very open sense of 'type', the memory usage. Such systems can guarantee that a distance in feet is never added to a distance in meters, that an integer value will

never overflow or that a program will never run out of memory. On the other hand, one might argue that a simple assembler for an integer only processor is statically typed since every register has the same integer type. Obviously the type safety offered by such an assembler is at best theoretical: one integer might be a distance while the other a memory pointer and adding these two values would not make sense.

SLinks uses a middle-of-the-road type system, similar in power to the type system found in O'Caml for example. It guarantees that the parameters of a function or of an operator are of a compatible type. That means that the examples above would be detected as incorrect. Such a type system has been proved as very useful in real-life programming to detect common bugs. I myself was absolutely delighted by the comfort it offered when developing the interpreter in O'Caml, when compared to more traditional weaker static type safety.

Furthermore SLinks checks types without requiring any explicit input from the programmer – except for the table operator. Of course, the programmer implicitly provides the type information because the operators he uses have type constraints attached to them. The type inference algorithm described later in this chapter will show exactly how. But in short, the '`+`' operator for example makes it explicit that both arguments must be integers, while '`++`' requires floating-point numbers. Similarily, constant values have a format that makes their type unambiguous: '`2`' is an integer, '`2.`' is a float.

4.1 Type language

Arguably the most fundamental part of a type system is the type language. Its expressiveness will determine what kind of guarantees the type system can offer. As explained in the introduction to this chapter, the SLinks type system must be sufficient to determine the type of an expression to a degree that allows to determine whether these expressions are valid parameters to an operator or function.

Figure 4.1 shows the grammar for SLinks' type language. A description of the non-trivial types follows.

The unknown type is used when the abstract syntax tree is created by the parser and when types have not been calculated yet.

The abstraction type is the type of functions. The left type is the type of the argument, the right type is the type returned by the function.

The record type define records by exposing what fields are present in them. Providing simply the type and label name of the fields in

k	:=	$undefined$	Unknown type
	|	$boolean$	Boolean
	|	$integer$	Integer
	|	$float$	Float
	|	$string$	String
	|	$database$	Database
	|	$'i$	Type variable
	|	$k \rightarrow k$	Abstraction
	|	$\{f_k, \ldots f_k\}$	Record
	|	$\langle f_k, \ldots f_k \rangle$	Variant
	|	$[bag\ k]$	Bag
	|	$[set\ k]$	Set
	|	$[lst\ k]$	List

Figure 4.1: Type grammar

the record is not expressive enough to support the possibilities of the SLinks language, in particular the row variable of record selection operations. This is why SLinks uses a type language for records similar to the system described by Wand in [5] and refined by Rémy in [6], [7] and by Wand in [8].

The possible fields in the record are defined by a data structure called a type row. A type row is a set of field descriptors. The grammar can be found in figure 4.2. A field in the type row can be of three different kinds:

- A field is said to be present if it is known that a field with the given label is present in the record. The type of the field is given too.
- A field is said to be absent if it is known that a field with the given label cannot be present in the record.
- A row variable is a special form of type field that can be replaced by any other type row – with some limitations described in section 4.2. For a more intuitive definition of a row variable, it can basically be seen as meaning "and more fields". Row variables in records allow an interesting form of polymorphism for functions. Take the following function that moves a point 2 units up in the x axis for example.

  `fun ^{#x=^x|^r}->{#x=x+2|r}` : $\{\sharp \mathrm{x} : int, 'a\} \rightarrow \{\sharp \mathrm{x} : int, 'a\}$

  This function can be applied equally to any point, be it in a one, two or n dimensional space or for that matter to any record that

$$
\begin{array}{rcll}
f_k & := & l : k & \text{Field present} \\
 & | & l : - & \text{Field absent} \\
 & | & {}'i & \text{Row variable}
\end{array}
$$

Figure 4.2: Type field grammar

has a '`#x`' field of type *int*. The absence of a row variable in a row means that the fields for this record are fixed and no row polymorphism is possible. Such a row is called 'closed row' as opposed to the 'open row' when a row variable is present.

The variant type is handled exactly the same way as a record: a row is used to describe what labels the variant value can have. The behavioural distinction between records and variants is not made by using a different row type representation but arises naturally from the type properties of the operators on both types.

The meaning of the fields in a row for a variant are different than those for the record, of course. A present field basically means that the execution of this expression can potentially yield a variant with the given label name and type, an absent field means it cannot. The meaning of a row variable is not very intuitive outside of an abstraction type – it is very important for type inference though. It basically means that the expression might be a sub-expression for an expression that has a variant type with more present fields in the row. It will become clearer how it works in section 4.2, but for now it is enough to consider that an expression of type variant always has a type with a row variable. In abstraction types with the argument type a variant, a row variable in this row means that the function can handle variants with other fields than those described. A closed row means it is strictly limited to the given set of variants. The following expression for example declares a function that will take as a parameter variants with label '`#int`' or '`#float`' exclusively.

```
fun ^x ->
  case x of <#int=^i> in <#int=i+1>
          or <#float=^i> in <#float=i++1.0>
```

$\langle \sharp\text{int} : int, \sharp\text{float} : float \rangle \rightarrow \langle \sharp\text{int} : int, \sharp\text{float} : float, {}'a \rangle$

The type variable is the type of an expression for which the type is undefined. Type variables again allow a form of polymorphism. The following example defines a function that will return the parameter

value and that can be a value of any type.

```
fun ^x -> x : 'a → 'a
```

Type variables and their polymorphic properties are also a fundamental part of the type inference algorithm described later.

The type system as described above is sufficient to give a type to a very broad range of SLinks expressions. The record and variant types in particular are very powerful types that provide type safety for complex expressions with an astonishingly simple mechanism. Wand in [8] even argues that a language with a type system with records and row variables is equivalent to a simple object-oriented language with "protected instance variables, publicly accessible methods, first-class classes and single inheritance".

The described version of the type system has one major limitation though: recursive types are not supported. That means that expressions that would be perfectly correct in untyped SLinks cannot receive a safe type with the existing language. Take for example the following expression. It will be rejected by the compiler even though the interpreter would, for certain parameters, be able to execute the function.

```
letrec ^nest = (fun (^x,^n) -> if n >> 0 then
  {#a=(nest (x, n-1))}
else x) in ...
```

In the SLinks interpreter, types are represented in a tree form similar to that of the syntax tree for the language grammar. This simplifies the handling of types as will be shown in the next section for the type inference algorithm.

4.2 Type inference algorithm

Since, as explained in the introduction to this chapter, the programmer does not provide the type of any expression in his program, it is necessary to have an algorithm to automatically calculate the single least general type of any expression often called the "principal type scheme". The principal type can be considered as the least type that any possible value the expression can take will at have. This operation is called type inference. SLinks' type inference system is based on an algorithm called 'algorithm $\mathcal{W}$' that has been first described by Milner in [9] and then improved and re-described in a shorter, clearer way by Damas and Milner in [10]. Algorithm $\mathcal{W}$ however only works on lambda calculus so needs to be extended to work with the broad range of operators provided by SLinks and in particular record, variant and table operators.

Type inference with algorithm $\mathcal{W}$ works on the principle that inference can be reduced to a problem of type unification. The general idea is that when an expression can come from either of two sub-expressions, the expression must receive a type that is compatible with both the types of the sub-expressions. For example a branch statement has two sub-expressions, one for the true and one for the false branch. At compile time it is unknown which branch will be selected, but it is known that the value of the branch expression will be one of the two values of its branches. Therefore, the branch expression must receive a type that is compatible with the type of both sub-expressions.

To make it more clear, an example is used to show how this would work in practice. Consider the branch operator again. If one branch has type '$int \rightarrow \{\sharp \mathrm{a} : int, {}'a\}$' and the other has type '$'b \rightarrow \{\sharp \mathrm{b} : int, \sharp \mathrm{c} : string\}$', the branch operator itself will have type '$int \rightarrow \{\sharp \mathrm{a} : int, \sharp \mathrm{b} : string\}$'. The reason is simple to understand: both sides of the branch are functions, so the branch itself must return a function. However, the type of this function must be such that no matter what branch is executed at runtime, the type the branch operator claims to have is compatible with the actual value it has.

The argument on one side is an integer and on the other can be anything. If the branch operator requires the argument to be an integer, the expression will work no matter what side of the branch is executed at runtime. Of course, it the second branch is executed, the function parameter could also have been a float or a variant, but by restricting it to an int. It is guaranteed that there will be no error at runtime, and this is exactly what static type safety is.

For the return value, the problem is basically the same. One side says the return type is a record that must contain field '`#a`' but can contain more fields. The other side says that the return record must contain fields '`#a`' and '`#b`' and only these. Both types are compatible in the sense that if the 'more fields' of the first record happens to be exactly the '`#b`' field, type safety can be guaranteed. This requires an assumption to be made and that must be taken into account for the remaining of the type inference. This assumption is made in the form of an equivalence that basically says that the 'more fields' represented by type row variable '$'a$' is equivalent to the field '$\sharp \mathrm{b} : string$', and this equivalence must be applied on every type calculated by the inference. Since the 'more fields' must come from the declaration of an open record with a row variable in the code, type inference must take into account that this row variable really has type '$\{\sharp \mathrm{b} : string\}$'. The following code that contains the branch operator I have been describing is an example of such a case. Notice that the '`x`' parameter of the outer function that is used to add the 'more fields' to the first branch is required to have type '$\{\sharp \mathrm{b} : string\}$'. One might also notice that despite the fact that only the

true branch will ever be taken, this is not considered for the type: type inference does no flow analysis.

```
fun ^x ->
  if true then
    fun ^a -> {#a=a+1|x}
  else
    fun ^a -> {#a=1,#b="two"}
```

$\{\sharp \mathrm{b} : string\} \rightarrow int \rightarrow \{\sharp \mathrm{a} : int, \sharp \mathrm{b} : string\}$

Of course, in some cases there does not exist a compatible type between two expressions. For example if in the branch example above one branch had type '*int*' and the other '$float \rightarrow \{'a\}$' the unification cannot find a compatible type. In this case, the unification fails and the program is rejected because it cannot be guaranteed as being type safe.

Algorithm $\mathcal{W}$ is an algorithm that does exactly that: it unifies the type of expressions that, according to the structure of the expression, must have a compatible type. Furthermore, it uses the type information encoded into operators to give base types when it can. For example in expression '`x+y`', because the plus operator only works on integers it is possible to make an assumption that x and y both actually are integers. Furthermore, algorithm $\mathcal{W}$ also supports polymorphism in the '`let`' statement. Algorithm $\mathcal{W}$ will be described below as well as the unification algorithm.

Unification algorithm

Earlier in this section I mentioned that the unification algorithm returns a new type. This is not quite true for the real implementation. Algorithm $\mathcal{W}$ requires a unification algorithm that returns what is called a substitution. A substitution is a list where the elements define an equivalence between type variables or type row variables and types or type rows. A type can then be substituted, which means all type variables or type row variables that are present in the given substitution are replaces by their correspondant. A substitution returned by the unification function is such that when either of the two types that have been unified is substituted with it, the resulting type is the unified type such as described above. This also implies that the substitution for two types will produce the exact same type when applied to either original types. Finally, two substitution can be composed to create a new substitution that is equivalent to applying the two substitutions to a a type sequentially.

The general unification algorithm is implemented as the '`unify`' function in the inference module. The main function is a recursive descent on the tree structure of the two types simultaneously. In general, at every step of the descent, the node of the first type must be of the same type as the

node of the second type, otherwise the unification fails. When reaching the leaves of the type tree – constant types – an empty substitution is returned. That means that, as expected, if both types are exactly equal the returned substitution is the identity. But there are other cases.

- When one side is a type variable, a new substitution is returned that says that the type variable is equivalent to the other type. If the other side is exactly the same type variable, no substitution is returned but if the other side is a type that contains the type variable, the unification fails. This is necessary to prevent recursive types that would need to be handled explicitly – Slinks does not support them.

- When both sides are functions, both the argument and the return type of the function are unified and the resulting substitutions are composed. That simply means that all assumptions made on both sides of the function type need to be taken into account.

- When both types are records or variants, the rows need to be unified with a special algorithm. This algorithm is described below.

When two rows need to be unified, there are three different cases to handle separately depending on whether a row variable is present.

- Both rows are closed. In that case the unification only works if the rows are equal as far as their field labels are concerned and if the types of every field pair – fields with the same label in both rows – can be unified. The resulting substitution is the composition of all substitutions for the unification of all field types. The absent fields are not taken into account for this unification.

- One row is closed, the other is open. All fields present in the open row must be present with an unifiable type in the closed row, similarly to what happens with a closed row, otherwise unification fails. The fields that are present in the closed row but not in the open one will be used to create a substitution that substitutes the type row variable from the open row to these fields. The resulting substitution is the composition of the row substitution plus all substitutions from field type unification.

- Both rows are open. The substitution returned in this case must transform both rows into a new row that is the union of all fields of both rows plus a fresh row variable. If a field with the same label is either present in one row and absent in the other or is present in both and has non-unifiable types, the rows are not unifiable. The way the algorithm solves this problem is as follows. First the two sets of fields, without the row variables, are simply merged. Then, for every field

in this new super-row, all other fields are searched for a field with the same label.

- If it does not exist, nothing happens.
- If it exists but is in a different state – present of absent – as the original one, unification fails.
- If is exists and is in the same state, one of the two fields is removed and if they are present fields, their types are unified.

The result of this first operation is to calculate a reference row. From there, everything is quite simple: both rows are intersected with the reference row. The returned row is used as the equivalence for the row variable of that row in a substitution. These substitutions are composed with the substitutions between field types obtained while calculation the reference row to obtain the return value

Algorithm $\mathcal{W}$

Algorithm $\mathcal{W}$ is implemented as the '`doubleyou`' function in the inference module. There follows a very short overview of the algorithm. For more detail, please consult [10]. The algorithm is implemented as a recursive descent on the syntax tree. A special data structure called the environment is passed along with the descent. The function returns a pair with the following items.

- A substitution that represents all assumptions about types that needed be made in the sub-expression for it to be type safe and that should be applied everywhere else for the entire expression to be type safe.
- The syntax tree with an attached type resulting from the inference on the sub-expression under the given environment and with all types already substituted by the returned substitution.

The environment contains a mapping from syntax variable name to type. In short it means that the sub-expression should be typed assuming that the encountered variables have the provided types. Furthermore, the types in the environment are extended with some additional information to allow them to support polymorphism. This information is simply a set of variables that should be treated as polymorphic in the type. Normally through substitutions all instances of a type variable should have the same type through the entire program. However by enforcing this everywhere, polymorphism can no longer exist. The following example shows such a situation. The – somewhat useless – '`f`' function should be polymorphic and should be applicable to values of any type. Its type when declared is '$'a \rightarrow 'a$' where '$'a$' is a polymorphic type variable. In the branch statement's condition, the function will be unified and its type will become '$bool \rightarrow bool$'. This does

not mean however that in the branch sub-expressions the function no longer can be used on integers. This means that type '$'a$' must not be substituted to type '*bool*', and that is exactly what a polymorphic type does.

```
let ^f = fun ^x -> x in
  if f(false) then f(4) else f(2);;
```

During the recursive descent, for every node in the syntax tree all sub-expressions are visited, the type of the sub-expressions that must be compatible are unified together and the type of sub-expressions that must have a known type – such as in an addition operator – are unified with this known type. If the operator for this node binds a variable name, such as for the '*let*' operator, the body of the operator is calculated with an environment extended by the variable with, as attached type, the type of the value bound to the name. All type variables in the value type that do not show up in any of the types in the environment at that point – they are said to be free – are flagged as polymorphic. This last operation is called the closure of the type.

The substitutions resulting from these unifications are composed with the substitutions returned by the sub-expressions to calculate the resulting substitution. Every sub-expression must have all its types substituted with this new substitution. Since the new substitution contains amongst others the substitution returned by a sub-expressions itself and implicitly the substitution returned by sub-expression calculated earlier than this sub-expression – the environment has been substituted with them – it is possible to only substitute it with the substitution composed from the unification results and substitutions from sub-expressions calculated later.

To define what types are unified with what other types for the different nodes, a formalism called type inference rules will be used. An inference rule simply give the type for an expression at the bottom in relation to the type of its sub-expressions calculated under a specified environment. Transforming an inference rule into the algorithm for a specific operator in algorithm $\mathcal{W}$ is quite straightforward.

- Every sub-expression in the upper part of the rule must be calculated under the specified environment, either the same as the original expression below or extended with an additional given binding.

- The first time a type is used it is bound to a type variable – its type is not known yet. Two types that have the same name must also be the same.

  - That means that if two sub-expressions have the same type in a

rule, they must be unified to make sure they are indeed compatible types.

  - That means that if a type of one sub-expression is used in the environment of another, the same type must be used with suitable substitutions applied.
  - That also means that the name of the type for the operator in the lower part of the rule says how it is constructed from the types calculated by the inference on its sub-expressions.

- If a sub-expression in the upper part has an explicit type, the calculated type for that expression must be unified with the constant type.

4.3 Type Inference rules

The preceding section presented algorithm $\mathcal{W}$ in a general manner, and described how inference rules can be used to define it. In this section, the inference rules for the most important operators of the SLinks language are presented. For more detail about the actual operators, please read chapter 2. The '$A \vdash \ldots$' in front of expression in the rules below is the environment. If the environment must be extended with a binding, the notation is '$A \cup (\texttt{var} : \textit{type}) \vdash \ldots$'.

Basic operators

When a type is enclosed in 'floor' brackets such as '$\lfloor \tau \rfloor$', it means the type is instantiated, that is any polymorphic type variable in it is replaced by a non-polymorphic one. A type enclosed in 'ceiling' brackets such as '$\lceil \tau \rceil$' is the closure as described above.

Variable

$$\frac{}{A \vdash \texttt{x} : \lfloor \tau \rfloor}$$

Lambda abstraction

$$\frac{A \cup (\texttt{x} : \tau_p) \vdash \texttt{e} : \tau_r}{A \vdash \texttt{fun ^x -> e} : \tau_p \rightarrow \tau_r}$$

Application

$$\frac{A \vdash \texttt{e}_\lambda : \tau_p \rightarrow \tau_r \quad A \vdash \texttt{e}_p : \tau_p}{A \vdash \texttt{e}_\lambda\texttt{(e}_p\texttt{)} : \tau_r}$$

Condition

$$\frac{A \vdash \texttt{e} : \mathit{bool} \quad A \vdash \texttt{e}_t : \tau \quad A \vdash \texttt{e}_f : \tau}{A \vdash \texttt{if e then e}_t \texttt{ else e}_f : \tau}$$

Binding

$$\frac{A \vdash \texttt{e} : \tau \quad A \cup (\texttt{x} : \lceil \tau \rceil) \vdash \texttt{e}_b : \tau_b}{A \vdash \texttt{let \^{}x = e in e}_b : \tau_b}$$

Recursive binding

$$\frac{A \cup (\texttt{x} : \tau) \vdash \texttt{e} : \tau \quad A \cup (\texttt{x} : \tau) \vdash \texttt{e}_b : \tau_b}{A \vdash \texttt{letrec \^{}x = e in e}_b : \tau_b}$$

Operators on constants

For each of the following families of operators, I will only present a single operator, the other ones being treated the same way.

Constant

$$\frac{}{A \vdash \texttt{c} : \mathit{bool}}$$

Operators on constants

$$\frac{A \vdash \texttt{e}_1 : \mathit{int} \quad A \vdash \texttt{e}_2 : \mathit{int}}{A \vdash \texttt{e}_1 \texttt{ + e}_2 : \mathit{int}}$$

Type transformations

$$\frac{A \vdash \texttt{e} : \mathit{int}}{A \vdash \texttt{float\_of\_int (e)} : \mathit{float}}$$

Records

Empty record

$$\frac{}{A \vdash \texttt{\{\}} : \{\}}$$

Record extension

$$\frac{A \vdash \texttt{e}_1 : \tau_1 \quad A \vdash \texttt{e}_r : \{l_1 : -, \rho\}}{A \vdash \texttt{\{l}_1\texttt{=e}_1\texttt{|e}_r\texttt{\}} : \{l_1 : \tau_1, \rho\}}$$

Record selection

$$\frac{A \vdash \texttt{e} : \{l_1 : \tau_1, \rho\} \quad A \cup (\texttt{x}_1 : \tau_1;\ \texttt{x}_r : \{l_1 : -, \rho\}) \vdash \texttt{e}_b : \tau_b}{A \vdash \texttt{let \{l}_1\texttt{=\^{}x}_1\texttt{|\^{}x}_r\texttt{\} = e in e}_b : \tau_b}$$

Empty record selection

$$\frac{A \vdash \texttt{e} : \{\} \quad A \vdash \texttt{e}_b : \tau_b}{A \vdash \texttt{let \{\} = e in e}_b : \tau_b}$$

Variants

Variant injection

$$\frac{A \vdash \texttt{e} : \tau}{A \vdash \texttt{<l=e>} : \langle l : \tau, \rho \rangle}$$

Variant selection

$$\frac{A \vdash \texttt{e} : \langle l_1 : \tau_1, \rho \rangle \quad A \cup (\texttt{x}_1 : \tau_1) \vdash \texttt{e}_1 : \tau \quad A \cup (\texttt{x}_r : \langle \rho \rangle) \vdash \texttt{e}_r : \tau}{A \vdash \texttt{case e of <l}_1\texttt{=x}_1\texttt{> in e}_1 \texttt{ | x}_r \texttt{ in e}_r : \tau}$$

Empty variant selection

$$\frac{A \vdash e : \langle \rangle}{A \vdash \texttt{case e} : \tau}$$

Collections

Rules are presented here for generic operators on '`col`' collections, but they apply to all three types of collections (sets, bags and lists).

Empty collection

$$\frac{}{A \vdash \texttt{[col]} : [col\ \tau]}$$

Singleton collection

$$\frac{A \vdash \texttt{e} : \tau}{A \vdash \texttt{[col e]} : [col\ \tau]}$$

Collection union

$$\frac{A \vdash \texttt{e}_1 : [col\ \tau] \quad A \vdash \texttt{e}_2 : [col\ \tau]}{A \vdash \texttt{e}_1 \texttt{ :col: e}_2 : [col\ \tau]}$$

Collection loop

$$\frac{A \vdash \texttt{e} : [col_{in}\ \tau_v] \quad A \cup (\texttt{x} : \tau_v) \vdash \texttt{e}_b : [col_{out}\ \tau]}{A \vdash \texttt{for \^{}x <col}_{in}\texttt{ e in e}_b : [col_{out}\ \tau]}$$

Tables

The table operator as I mentioned earlier uses a hack to infer its type. The type 'τ' comes from the model of the table provided by the developer (and is a closed record). This type is stored in the type field of the table node and is directly available without any inference whatsoever.

$$\text{Table}\ \frac{A \vdash \texttt{e} : \mathit{database}}{A \vdash \texttt{table 'sql' on e} : [\mathit{col}\ \tau]}$$

Chapter 5

Database access optimisation

This chapter will start with an explanation of the rationale for optimising database access and the way it has been done in SLinks. It will then detail the various optimisations that have been implemented in the SLinks prototype.

Examples in this chapter will be presented using a mixed sweet and core syntax. Since optimisations are made on the core syntax, it must be used to explain what is going on, but since it sometimes is pretty complicated to read, sweet operators will be used where it does not interfere with the understanding of the algorithm.

The database access operators in SLinks have been presented in earlier chapters, but without any detail about the rationale for them to be as they are. Let me explain. Traditionaly, languages to access databases – query languages – have been different from general purpose programming languages. To access data from a program written in a general purpose language, an expression in a query language is typically inserted in the code of the host programming language. This is not the case in SLinks: the table operator in the sweet syntax is completely free of any query language. It also is by itself extremely limited: it only allows to collect all tuples of a table, without any condition or other fancy transformation on the result such as those provided by typical query languages. But SLinks is not limited in that respect; it is SLinks itself, although designed as a general-purpose programming language, that serves as a query language. Comprehensions allow to define complex conditions and transformations on collections and allow to define jointures equivalent to those found in a typical query language. But comprehensions are also constructs that are very suitable to handle collections that do not come from a database, and are therefore also a useful part of SLinks as a general purpose language. See figure 5.1 for a few examples of SLinks expressions and corresponding SQL queries. In these examples the table operator is abbreviated.

```
[set x.#a|^x <set table "tata"]
SELECT a FROM tata

[set x|^x <set table "tata", x.#b << 4]
SELECT * FROM tata WHERE b < 4

[set [set x.#a+y.#a|^y table "titi"]|^x <set table "tata"]
SELECT tata.a+titi.a FROM tata JOIN titi
```

Figure 5.1: SLinks versus SQL, examples

This mixing of general purpose and query specific features in a single language has various advantages. I will shortly describe three of them which seem to me particularly noteworthy.

- It improves code reusability. The syntax to handle collections in SLinks is completely oblivious to where the collection comes from. That means that the same piece of code can be used to handle program values or values coming from a database. This also means that changing the source of data from a program value to a value coming from a database can be done in a completely transparent way.

- It can improve the safety of the program. By treating database operators as any other operator in the language, they will benefit from the safety provided by the type system. This is not the case with traditional languages were the query is usually treated independently from the context it appears in. Often queries are treated as simple strings in the host language, and are therefore not even statically checked for syntactic correctness let alone type safety.

- It is simpler to program. It basically divides by two the number of languages a developer needs to know.

However, despite the advantages described in the preceding paragraph, there is a very strong reason not to mix both general purpose and query languages in an intricate way: performance. The performance problem when no separated query language is used is twofold:

1. To be efficient, a query needs to be optimised aggressively. An enormous amount of research and development has gone into the optimisation of query languages, such as SQL. To give an example of what can be done, modern query optimisation uses information about the content of the database to select the one solution to the query that will be the most efficient for the real data set. It is evident that this cannot be done at compile time. That means that if the database is accessed

with a query language it cannot optimise, the query will be unnecessarily slow. This is also true if the language sends many basic queries and does the actual data processing itself: to be efficiently optimised, the query sent to the database must be as complete or extensive as possible.

2. If the calculation of the query – as opposed to pure data access – is handled by the program, all necessary data must be transferred from the database to the process running the program, which is typically much more data than the actual result of the query. This can represent a very major bottleneck for performance and for typical simple queries rather more than the first point.

SLinks, in its core syntax, already uses a query language – SQL – for access to tables. However, these table queries are very simple: they simply download the entire content of a table into the program and then handle the resulting data with comprehensions. This is therefore not a solution to the problems mentioned above. To obtain some real performance in SLinks it needs to be optimised so as to push as much as possible of the treatment of database data to a database query. More precisely various optimisations must select a sub-expression around the table operator and merge it into the query attached to the table operator. This sub-expression must be as large as possible, in order to obtain the largest possible query for the reasons explained above. On the other hand, it must also be small enough so as not to contain calculations that the database management system is unable to do.

Obviously, these optimisation basically bring the language back to the beginning: queries in a specialised language embedded in the general purpose language. However doing this automatically during optimisation has many advantages. Firstly, since the programmer is unaware – or at least can be – of the transformation, all the advantages for integrated queries described above remain. Furthermore, my own experience in programming with non-integrated database access, such as that provided by JDBC for example, shows that there is a strong tendency to use simple queries and then use the mechanism provided by the host language to treat the data. Writing complex queries is a complicated business and when used inside a language, it is natural to use the host language – Java in the above example – to solve problems rather than another alien language – SQL in that case. That means that by taking out the responsibility for writing the database queries out of the hands of the programmer, even though it might arguably prevent some specific hand-coded optimisations, will in general provide better results.

Kleisli and CPL [2] have already been mentioned before as a strong inspiration for SLinks. One of the closest similarity is the optimisation

mechanisms that will be presented in this chapter. Fundamentally, most algorithms presented here are similar to those used in Kleisli and described in [11]. However, there is a major difference between both languages: CPL is a query language while SLinks is not. This means that in Kleisli, the optimisation algorithms are built around the assumption that they will be written in a certain way, which is an assumption that cannot be made in a general purpose programming language. Furthermore, the possibility to access row variables in SLinks, which is not possible in CPL, dramatically complicated the optimisation algorithms.

More precisely there is one feature in SLinks that CPL does not support which is responsible for most of the added complexity: the record selection operator. In CPL, the only way to access a field in a record is through a 'dot' operator. Since dot is applied directly to the record expression or variable and not bound to a variable to be used later, the label of the field extracted from a record is available right where it is used. On the other hand in SLinks, where field selection is implemented as a special binding operator, the label of the field from which a variable value comes has typically been calculated earlier. Row variables complicate the whole thing even more since a record, or parts of it, is not bound to a single variable but potentially to many. In Kleisli this problem could also partially arise since a record selection operator à la SLinks could be simulated – with no row variable though – as '`let \a == r.#a in ...`' which is similar to SLinks' '`let {#a=^a|r} in ...`'. However as far as I can tell, this situation is simply not considered in Kleisli's optimisation algorithms. On the other hand, Kleisli supports optimisations for queries on multiple databases that have not been considered in SLinks.

I must say however that as far as the genericness of optimisations in SLinks is concerned, there still is room for improvement. In particular, there are two major problems:

- A SLinks program is usually composed of separate expressions. The resulting value for each of them is stored in the environment with a '`def`' or '`defrec`' operator. Later expressions are then able to reuse these values. However, because of the architecture of the system, the optimiser does not have access to the values in the environment. This limits the scope of some optimisations that could inspect these values and make less restrictive assumptions about what they are. Type information for the values in the environment is available to the optimiser, which is already useful but not for all optimisations. If the system architecture was changed to provide these values, there would be no requirement for conceptual changes to the algorithm: the only reason it is not done is because I did not have time to add it.

- Since the interpreter is not lazy, an expression must be evaluated at once. That means that the scope for optimising database access is at

most the expression. For example the following two expressions would not produce optimised code, despite it being possible.

```
def ^tata = table "tata" with {#a:int,#b:int} from db;;
[bag t.#a | ^t <set tata, t.#b >> 3];;
```

When the second expression is executed, the database will already have been queried and it will be too late to optimise anything. In a compiler, instead of an interpreter, it might be possible to mitigate the problem by replacing the variables referring to a table value by the value itself. This would allow the expression to be optimised. On the other hand, this would make the table operator behave as if it was lazy, since its evaluation would no longer happen at the expected place. This is acceptable if no side effects exists, but a database is inherently a source of side effects: if the database is modified between the declaration of the table in the source and the place it will be executed after the optimisation, the results might be wrong. Solving or mitigating this problem is a complicated business that has not been considered at all in this project.

This chapter concentrates on SQL optimisers. However, there is another simple optimiser in SLinks used both to normalise the syntax tree for the SQL optimisations and for me to remember how to write an optimiser when I started with that. The unused variables optimiser removes all '`let`', '`letrec`' and record selection operators that bind variables that will never be used in their bodies. It also removes all record selection operators that define a row variable that will only be used as the value variable for a later record selection. This obviously changes the type of an expression, but not the runtime behaviour. Since type checking is done before optimisation, such a type-changing optimisation is acceptable.

Optimisation principle and visitor function

Before starting with the optimisation algorithms, I need to define what could be called an 'algorithmic behaviour' that is used thoroughly in the following algorithms: a visitor function. A visitor function is a recursive descent through the abstract syntax tree returning an abstract syntax tree. That means that for any syntax node, the return value is the syntax node with every sub-expressions replaced by the result of the same recursive function applied to this sub-expression – a syntax tree itself. This function basically returns a copy of the syntax tree. Obviously, for some nodes, a different behaviour will be applied: that is where the optimisations will take place.

In some cases, a visitor function, instead of returning directly a syntax node will be wrapped in a variant type. This allows either the normal syntax node to be returned or a token to say something about the state of the visited

expression.

To optimise the syntax tree, the SLinks interpreter will successively call a number of visitor functions, called optimisers, on the syntax tree. Each of them provides one specific optimisation. The order in which they are executed is of course very important: some optimisers are dependant on the transformations made by another optimiser to the syntax tree – most SQL optimisers require the renaming optimisation to have been applied already. On the other hand some optimisers cannot be run after others – for example selection optimisation cannot be run after the join optimisation.

5.1 Optimising projections

A table in a relational database can conceptually be seen as a two dimensional matrix with columns, one for every field, and rows, one for every tuple or element. Projecting a table is the operation that creates a new table with a sub-set of the columns from the original table.

The projection optimisation in SLinks works by studying the way the collection resulting from a table operator will be used, and in particular what fields in the collection's records will actually be used. It then projects the table on only the required columns. The following expression for example is a typical case where such an optimisation would be useful.

```
for x <set (table "ex" with {#a:int,#b:int} on db) in
  let {#a=a | y} = x in [set a]
```

The following example shows the original SQL expression before the optimisation. It contains all columns present in the table. But since only column 'a' will ultimately be used in the expression's body, the second case of the example shows a new query, projection of the first, that for this particular code is equivalent. This is what the projection optimisation aims to do.

```
SELECT a, b FROM tata
SELECT a FROM tata
```

Projecting a query does not usually simplify the query itself, but reduces the amount of data being transferred from the database management system to the SLinks process. The improvement can be particularly important if the dropped columns have expensive types, such as large 'varchar' columns.

Optimisation algorithm

This optimisation, even though it is very simple to describe, has been a major source of troubles for me during this project. It certainly is the one optimisation rule that saw the most different versions, and I am still

not quite happy with it now – it is better than at the beginning but by no means perfect.

The projection optimisation seemed to be a very good candidate to use for type-based only optimisation. This would mean that instead of using the syntax tree to decide what to optimise, the type of some selected sub-expressions is analysed and used to optimise the syntax tree. However, despite the theoretical beauty of this solution, it was not as smooth as expected: the way the type inference system is implemented, and the way types are stored in the syntax tree makes it difficult. More precisely, the problem is that since all types are unified, the type of the table row that contains all possible fields will be unified with the type of the loop variable of the collection to optimise. That means that the inferred type of the record representing a table row will always contain all the declared fields of the table and hence will not be useful. To make this work, it would be necessary to store the type of the variable before it is unified, but this would require a modification to algorithm $\mathcal{W}$ and to the syntax tree.

The current solution is a somewhat hybrid approach – which really means sub-optimal I am afraid – that uses both the syntax structure and type information to do the optimisation.

The projection optimisation algorithm is called '`sql_projections`' in the SLinks implementation. The general principle is as follows:

- A visitor function searches for collection loops on tables, bottom up. For each of them it calls a '`needed_fields`' function on the body of the collection that checks what columns from the table are used. It can return three tokens.
  - A '`‘None`' token to say that the loop variable of the collection is completely unused. In this case the SQL query for the table is modified to something like '`SELECT NULL FROM ...`'. It is not possible to drop the query altogether because the number of rows returned still needs to be obtained. It would be even better to use '`SELECT COUNT(*) FROM ...`' but it would require a modification of the database back-end.
  - An '`‘All`' token that says that the loop variable is used directly without any field extraction. In this case the query cannot be optimised.
  - A '`‘Some`' token that lists what fields are used. In that case the SQL query is modified to select only the columns corresponding to the fields that are actually used.
- The '`sql_projections`' is a visitor function that will visit the tree. It takes as argument the name bound to the loop variable. For most

nodes, the node is simply returned but with a token composed by calling a '`merge_needed`' function on the tokens returned by applying the function on the child expressions. The following nodes have a different behaviour.

A variable with the same name as the function's parameter returns ''`All`'.

An apply is the case where type information is used. If the function part of an apply is a variable, the type of the variable is looked up, if it is an in-line function, the function is retyped. The type of the function is then analysed. The parameter part must be either a type variable, in this case ''`All`' is returned, or a record type in which case ''`Some`' is returned with a field list composed of the present labels of the record type.

A record selection on a variable with the same name as the function's parameter returns a ''`Some`' with the label name for this operator. It then merges this with the result of the function called on the body both for the original loop variable name and for the row variable. This is needed since remaining fields from the original record are now present both in the original loop variable that is still available and to the new row variable.

An empty record selection on a variable with the same name as the function's parameter returns ''`None`'.

- The '`merge_needed` ' function returns ''`All`' if any of its parameters is ''`All`', ''`Some`' with a field list composed of the union of all fields in the field list of all ''`Some`' parameters and ''`None`' otherwise.

5.2 Optimising selections

Like the projection, a selection will reduce the size of a table, but this time for the other dimension. Selecting a table is the operation that creates a new table with a sub-set of the rows from the original table. The selection will be made according to a condition on the content of the fields for each row.

The collection optimisation in SLinks works by studying the branch operators in the body of the collection loop. A branch where one of the branches is an empty collection is a candidate for being inserted into the query. If the branch condition can be executed in SQL, the branch operator is removed and the query is completed with an additional selection. The following expression for example is a typical case where such an optimisation would be

useful.

```
for x <set (table "ex" with {#a:int,#b:int} on db) in
  let {#a=a | y} = x in let {#b=b | z} = y in
    if a << 4 then [set a]
    else [set]
```

It is possible to remove the branch altogether and modify the query for the table to '`SELECT a, b FROM tata WHERE a < 4`', which is a selection from the original table. This is what the projection optimisation aims to do.

The advantage of this optimisation is similar to that of the projection optimisation, as it reduces the amount of data to be transferred from the database to the SLinks process. In favourable cases this reduction can be extremely important, typically when a single tuple is needed from the entire table, which is quite a common situation. The query itself might also be shorter to execute if the condition only uses indexed columns, which should always be the case in a well-designed database.

The selection optimisation algorithm relies on a sub-algorithm for condition extraction. I will start by explaining this algorithm that will also be used for the next optimisation on jointures. This algorithm looks in an expression for branch operators that return an empty collection for one branch. They are the kind of operators that can be transformed into an SQL selection. The functional also needs to make sure that this particular condition can be pushed to the query.

Optimisation algorithm

The selection optimisation algorithm is called '`sql_selections`' in the SLinks implementation. The general principle is as follows:

- A visitor function searches for collection loops on tables, bottom up. For each of them it calls the '`extract_tests`' function on its body which tests whether some condition in the code can be pushed into the query. '`extract_tests`' returns positive and negative conditions to be added to the query, a modified body expression with the conditions that will be added to the query removed and a data structure called the origin. The origin is a representation of a tower of record selection operators that define variables which are used in one of the conditions to be pushed to the query and which were declared in the body of the collection loop but can equally be defined before – that means they do not depend on any value derived from the collection loop. The visitor then builds a new table operator with a new query obtained from the '`select`' function with the given positive and negative conditions. A new collection loop operator is built with the new table and the

new body. Finally, the origin is attached before the loop with the '`add_origin`' function and the resulting expression is returned.

- The '`extract_tests`' function is a visitor function, but with a modification: instead of simply visiting the expression, it keeps track of where variables come from. To use this, it keeps what is called a binding environment. A binding is simply a data structure that gives the origin for a particular variable, which can be one of the following:

  - The loop variable of a collection loop on a table. Since bindings are used to push tests into a query, it is important to know which variables come directly from the table since they are the ones that can be optimised.
  - The field and the row variable in a record selection on an earlier variable.
  - The variable is unavailable for condition extraction. That means its value is calculated and is not simply a sub-part of the value of an earlier variable.

  The collection loop binding is passed to the function by the top level visitor, when it detects a collection loop on a table. The bindings are then refined while the function visits the expression. A record selection on a variable will create a record selection binding, all other cases will create an unavailable binding. It is important that all variables, even the unavailable ones be bound since the absence of binding also has a meaning: the variable was already available before the sub-expression – that is it is available to use inside the table operator of the collection loop to optimise.

  When the '`extract_tests`' function encounters a branch node with one of its branches an empty collection, it has found a candidate for optimisation. It will call the '`extract_condition`' function on the condition of the branch with the current binding environment. '`extract_condition`' can return an SQL condition that is equivalent to the SLinks condition and a corresponding origin. In this case, it returns the SQL condition either as a positive of negative condition, depending on which branch the non-empty collection is, the expression without the branch itself, and the origin. '`extract_condition`' can also return a token to say that the condition cannot be transformed into SQL. In this case, the node is returned unmodified.

- The '`extract_condition` ' function looks at both sides of a comparison to check whether it can be used as a SQL condition. The rules for this are as follows:

- At least one side of the comparison, or both, must be a field extracted from the loop variable. That means that it must correspond to columns of the database. To check that, the binding environment is used. If the variable can be traced back through at least one record selection binding to a loop variable binding, it corresponds to a database column.
- The other side can be a constant.
- The other side can be a variable declared earlier than the collection loop, in which case it is usable in the table query. A variable is declared earlier if it is not present in the binding. Alternatively, it is also available if it can be traced back through a list of record selection bindings to an absent binding. In this latter case, the variable is available, but requires an origin corresponding to the tower of record selections needed to create the variable.

If those rules are true, '`extract_condition` ' returns the condition transformed into SQL syntax and if needed, the origin. If they are false, it return a token saying the condition cannot be pushed.

- The '`select` ' function returns a new query with the '`WHERE`' parameter extended with the provided positive and negative conditions. Positive conditions are bound together with logical '`AND`' operators. Negative conditions are bound with logical '`OR`' operators and the whole bunch is negated. The eventual original conditions of the query and the positive and negative conditions are then all merged with logical '`AND`' operators.

- The '`add_origin` ' function simply adds a tower of record selection operators before an expression based on the provided origin.

5.3 Optimising jointures

As opposed to the two earlier optimisations that constrain the size of a single table, the join operation works on two tables. If a table is considered as an array, the jointure simply is the Cartesian product of two tables. This is equivalent to saying that the join operation creates a new table, called the jointure, with all columns of both original tables and where the rows are every row of the first table appended to every row of the second. The number of rows in the jointure table is therefore $m * n$ where m and n are the number of rows of the original tables.

The join optimisation in SLinks looks for nested collection loops, which are equivalent to a jointure, and transforms them into a jointure table. There are a few conditions to be fulfilled in order for this optimisation to be applied though.

- A condition must exist between a field from the first and a field from the second table. This is necessary in order for this optimisation to produce a speed-up, as explained below.

- The two tables must come from the same database. Obviously a single query cannot normally span multiple databases.

- The two tables must have fields of different names. This is not theoretically necessary – it would be possible to rename appropriate fields – but the implementation of this renaming etc. is something that I did not have time to include in the current algorithm. This mechanism should not change the principle of the algorithm though.

The following expression for example is a typical case where such an optimisation would be useful.

```
for ^x <set (table "ex" with {#a:int,#b:int} on db) in
  for ^y <set (table "wai" with {#c:int,#d:int} on db) in
    if (x.#a == y.#c) then
      [set {#b=x.#b,#d=y.#d}]
    else [set]
```

In an optimised version of this code, the inner loop and the branch could both disappear with a new query for the table in the outer loop. This new query, with a join on a condition would be as follows.

```
SELECT ex.a, ex.b, wai.c, wai.d FROM ex JOIN wai WHERE ex.a = wai.c
```

When the situation is favourable, this optimisation can be extremely useful both to reduce the time for treating the data – the database can use indexes to help calculate the join – and the amount of data to transfer. However, as opposed to the two last optimisations, that will never be unfavourable, this one can lead in unfriendly cases to lesser performance. The reason is that the numbers of rows for an unconditional jointure is $m * n$ while the number of rows to transfer for the same information using two separated queries is only $m + n$. Even though the join optimisation is limited to situations where a condition between the two tables exists, which should hopefully reduce the number of rows to be downloaded, there is no guarantee it will be reduced below $m + n$. In real-life cases it tends to be so however, this is why this optimisation is useful in practice.

Optimisation algorithm

The join optimisation algorithm is called '`sql_joins`' in the SLinks implementation. The general principle is as follows:

- A visitor function searches for collection loops on tables, bottom up. For each of them – called outer loops – it executes a '`check_join`'

function on the loop's body that looks for another collection loop on a table – the inner loop – that can be joined with the outer table. '`check_join`' returns a '`'None`' token if the body does not contain any such table. In this case, the collection loop is returned unmodified. Otherwise '`check_join`' returns a '`'Some`' token which carries a number of parameters with it.

  - A list of positive and negative conditions between the two tables to join. They are represented similarly to those described in this chapter on page 51.
  - The SQL query attached to the inner table.
  - A modified body for the collection loop without the inner loop nor any of the branch operators corresponding to the joining conditions.

  In this case the visitor returns a new collection loop node with the new body and a new SQL query attached to the node. This new query is obtained through the '`join`' function that uses the positive and negative conditions as well as the two original queries, inner and outer, to calculate the new one.

- The '`check_join`' function visits the syntax tree to find collection loops on tables that can be joined with the outer table. If it encounters branch nodes with an empty collection for one branch, it visits the non-empty branch. It also passes through binding operators and updates a binding as described on page 51 of this chapter.

  If it encounters a collection loop operator it calls the '`extract_tests`' function on the loop's body. This function is the same as the one described in this chapter on page 51. However, it is called with the binding from the outer loop extended with a new table binding for the inner loop. This means that it will be able to extract tests between both tables. If no test is extracted, the function returns '`'None`'. On the other hand, if at least one test exists, the function will continue by calling the '`project`' function. If '`project`' returns a new joined body, the function returns '`'Some`' with the positive and negative conditions from the '`extract_tests`' function, the inner loop query and a new body expression. This new body is the one returned by the '`extract_tests`' function, without any of the branch operators calculated as query conditions and extended with the origin as described on page 52 of this chapter .

  Otherwise, if '`project`' returns with a token that says the body cannot be joined or if it encounters something else than a collection loop, it will stop and return the '`'None`' token.

- The '`join`' function simply merges together two queries. The selected fields are simply concatenated, the table names are joined and a new '`WHERE`' parameter is built from both original queries' parameter and the positive and negative join conditions in a similar fashion to the one described on page 52 for the selection optimisation.

- The '`project`' function is a very simple function that replaces all instances of the inner loop variable by the outer one. It will return a token that says it cannot be joined if any of the two loop variables are used directly – only extracted fields can be used since the new loop variable's record will contain all fields of both original ones. This is a limit that could be circumvented by recreating, with record extension operators, the required loop variable's record. The needed information is available since the name of all fields for each loop variable's records are contained in the model of the table.

5.4 Ordering

Having the database sort a table before sending it to SLinks is something already provided by the table operator. The goal of this optimisation is to make it equivalent, from a performance point of view, to specifying an ordering in a table and to specifying it with a sort operator on a collection coming directly from the table. This means that the sort, instead of being calculated by SLinks, will be calculated by the database management system. The following expression for example is a typical case where such an optimisation would be useful. The second expression shows an equivalent expression but in optimised form.

```
sort_up(
  for ^x <set (table "ex" with {#a:int,#b:int} on db) in
    let {#a=^a,#b=^b|^y} = x in [set b])
```

```
for ^x <lst (table "ex" with {#a:int,#b:int}
                       order [#b:asc] on db) in
  let {#a=^a,#b=^b|^y} = x in [lst b])
```

The optimised version removes the sort operator and adds the corresponding ordering to the table. The ordering depends on what field are present in the list when the sort operator is applied, in this case only the '`#b`' field. Furthermore, the type of the collection operators between the sort operator and the table must be changed to list operators.

I must say I am not particularly happy with this optimisation, and this for two reasons:

- The ordering mechanism provided by SQL is very minimal. The ordering comparison cannot be modified, only the default comparison

provided on the type can be used. That means that the sort operator in SLinks must also have the same limitation, and make the same assumptions about ordering as SQL. This works with basic types, numbers or strings, for example, but is quite limited for a general purpose language. It becomes even more problematic when more complex types are used. Complex database types will typically be represented in SLinks as records. That means that the comparison on this particular record must be equivalent to the comparison on the corresponding database type, which cannot be done in the current version of SLinks as 'less' or 'more' comparisons on records are undefined.

Furthermore this also means that the optimisation is extremely delicate. If the value in the collection is transformed, even in a trivial way, it is no longer possible to push it to the sort operator of the database because if would no longer correspond to the ordering built in the database. It might be possible to test if the transformation is strictly monotonic, which would then allow to push the ordering nevertheless, but this is something that I did not pursue.

- It requires a comparison to be defined on records. Equality and inequality are no problem, but the less and more comparisons are ill-defined on records. Since the fields in a record are theoretically unordered, the order in which the comparison should be done on the different fields' values cannot be determined. This problem is solved in the current prototype in a rather unsatisfactory way: direct 'less' or 'more' comparisons on record are undefined – or return false every time – but when the ordering optimisation is applied, it will use the order in which the fields are declared to select how they should be pushed into the table. For example, in the example above, if the returned set was '`[set {#a=a,#b=b}]`', the ordering would have been '`[#a:asc,#b:asc]`', on the other hand, if the returned set was inverted like this: '`[set {#b=b,#a=a}]`', the ordering would have been inverted too like this: '`[#b:asc,#a:asc]`'.

Optimisation algorithm

The sort optimisation algorithm is called '`sql_sort`' in the SLinks implementation. The general principle is as follows:

- A visitor function searches for sort nodes, bottom up, and for each one executes the '`push_sort`' function on the sort operator's body. This push function can return in two states. It returns a new syntax tree for the sort body that is modified so as to be ordered in the same way as the sort operator would have ordered it. In this case the sort operator is dropped and is replaced by the new sort body. It returns

a ‘‘`No_push`’ token that notifies that no sorting can be done. In that case, the sort operator is unchanged.

- The ‘`push_sort`’ function is a visitor function that searches for the source of the collection that is ordered. If it is a collection loop on a table, it executes an ‘`expr_fields`’ function on the loop body to find out what elements of the table are used in the elements of the returned collection, and in what order. This function can return three different tokens.

  1. The ‘‘`Not_found`’ token indicates that the collection is not composed directly from elements of the database. In this case, the sort cannot be pushed and ‘‘`No_push`’ is returned.
  2. The ‘‘`Found_all`’ token indicates that the record returned by the table is used directly as the collection's elements, without being split up into fields. The order in which the fields will be ordered depends on the order in which they have been declared in the table operator – which is not compatible with a strict definition of records.
  3. The ‘‘`Found`’ token indicates that a specific field or set of fields is used explicitly as the collection's elements. If there are many of them, the order in which they are declared in the collection element record is returned and used to define the order in which they will be ordered – which is again not compatible with a strict definition of a record.

  In the two latter cases, the function returns a new syntax tree with the orderings returned by the ‘`expr_fields`’ function added to the SQL query, and the entire expression modified to use list operators everywhere, to keep type correctness.

  If the source of the collection that is ordered is not a collection loop on a table a ‘‘`No_push`’ token is returned.

- The ‘`expr_fields`’ is a binding visitor function such as described in this chapter on page 51. The bindings are obviously used to define which variables used as the collection's elements come from the database, and as which field. The function first finds the single element collection operator that builds the collection for this collection loop. Currently, and this might be improved, if it finds a condition or the application of a function before the single element collection, it returns a ‘‘`Not_found`’ state, which is correct but too conservative.

  It then visits the body of the single element collection. If the body is the looping variable for the collection loop, the ‘‘`Found_all`’ state is returned. If it is a variable attached to a field of the looping variable,

the ‘‘`Found`’ state with a single field is returned. If it is a tower of record extensions ended with an empty record, the ‘‘`Found`’ state is returned with the corresponding list of fields. In all other case, the ‘‘`Not_found`’ state is returned.

Chapter 6

Conclusion

SLinks is very much a work in progress. It has been used to test various concepts but, as a prototype, it does have various rough edges and limitations that I described through this report. There are also plenty of ideas that I would have liked to add to the prototype but that I did not have time to implement. Altogether, however, I believe that SLinks served its purpose: to demonstrate how database access features can be added to a functional programming language with static type checking and advanced records and variants with row variables. It also is a good framework on which more can be built, both to improve the existing database optimisation techniques and to add new ones.

As a prototype, a large number of future improvements or research on the SLinks interpreter are possible. Of course, I suppose these will mostly depend on what is needed for the Links language, but here are a few that I would find particularly interesting.

- Currently only relational SQL databases are supported. The SLinks type model is powerful enough to represent more or less any data structure as explained in [2]. Adding support for XML database through XQuery for example would be a natural improvement of the current system. This would also be as far as I know something completely new.

- The SLinks interpreter and language are still very rough on the edges: inexistent error reporting, limited operators, sometimes unfriendly syntax etc. This means that it is practically impossible to write serious – that is large – programs in SLinks and until now only small example expressions have been used. This is problematic for two reasons. Firstly debugging is much weaker that way: real-life programs can come up with real-life bugs. Secondly it makes it difficult to assess exactly how usable the language really is. To test optimisations

or type inference it is sufficient though, but with a little additional work it could be used to test plenty of other problems too.

- Rethink the way types are represented in the abstract syntax tree. As I mentioned in the report, if I had to rewrite the interpreter from scratch now I would change the way types are kept, and I think this should be a priority if SLinks is to be used for further prototyping of algorithms that rely on the type.

To conclude on a more personal note, I would like to speak very shortly about what this project meant for me. Before starting this project, having only had a school-work approach to problems, I did not really realise what research was all about. It took me quite a while to get used to the idea that the best place to find information is in complicated articles with plenty of Greek letters and not on neat and pedagogic slides or summaries. In a sense, this was quite a shock as this approach to problems is very different from what I had been used to. I must say however that after six month of this new way of understanding things, I am starting to enjoy it. Articles are much more interesting than slide shows: they contain more than simply the end result. The research procedure, the way the researcher that publishes the paper thinks or even his personality always transpire through an article. This is very interesting as it gives an insight into how research is done and for a student about to graduate I suppose this is pretty much as important as it can get.

Apart from this 'metaknowledge', this project was also very interesting from a purely experience-gathering point of view. Complex static type inference was something completely new to me as were large parts of the optimisations techniques used. The development of a larger and rather complex application all by myself – during my studies almost all development I made was in teams – was very useful: I think I am a better programmer now, particularly with functional languages. Altogether, I think this project was an overwhelmingly useful experience. The only regret I have is that after six months in Edinburgh, I still can't understand a person with a strong Scottish accent ...

Bibliography

[1] P. Wadler, "Links." 2004.

[2] L. Wong, *Querying Nested Collections*. PhD thesis, University of Pennsylvania, 1994.

[3] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong, "Comprehension syntax," *ACM SIGMOD Record*, vol. 23, no. 1, pp. 87–96, 1994.

[4] X. Leroy, *The Objective Caml system, release 3.08*, 2004.

[5] M. Wand, "Complete type inference for simple objects," in *Proceedings of the IEEE symposium on Logic in computer science*, pp. 37–44, 1987.

[6] D. Rémy, "Type checking records and variants in a natural extension of ML," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 77–88, 1989.

[7] D. Rémy, "Type inference for records in a natural extension of ML," Tech. Rep. 1431, Institut national de recherche en informatique et automatisme, Rocquencourt, 1991.

[8] M. Wand, "Type inference of objects with instance variables and inheritance," Tech. Rep. NU-CCS-89-2, Northeastern university, Boston, 1989.

[9] R. Milner, "A theory of type polymorphism in programming," *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[10] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212, 1982.

[11] S. B. Davidson and L. Wong, "The Kleisli approach to data transformation and integration," in *The Functional approach to Data Management: modeling, analyzing, and integrating heterogeneous data* (P. Gray, L. Kerschberg, P. King, and A. Poulovassilis, eds.), pp. 135–165, Springer-Verlag, 2004.