The FREGE Programming Language (*Draft*)

by Ingo Wechsung

last changed February 24, 2014
3.21.285

Abstract

This document describes the functional programming language FREGE and its implementation for the JAVA virtual machine. Commonplace features of FREGE are type inference, lazy evaluation, modularization and separate compile-ability, algebraic data types and type classes, pattern matching and list comprehension.

Distinctive features are, first, that the type system supports *higher ranked polymorphic types*, and, second, that FREGE code is compiled to JAVA. This allows for maximal interoperability with existing JAVA software. Any JAVA class may be used as an abstract data type, JAVA functions and methods may be called from FREGE functions and vice versa.

Despite this interoperability feature FREGE is a pure functional language as long as impure JAVA functions are declared accordingly.

What is or who was Frege?

Friedrich Ludwig Gottlob Frege was a German mathematician, who, in the second half of the 19th century tried to establish the foundation of mathematics in pure logic. Although this attempt failed in the very moment when he was about to publish his book *Grundgesetze der Arithmetik*, he is nevertheless recognized as the father of modern logic among philosophers and mathematicians.

In his essay *Funktion und Begriff* [1] Frege introduces a function that takes another function as argument and remarks:

> Eine solche Funktion ist offenbar grundverschieden von den bisher betrachteten; denn als ihr Argument kann nur eine Funktion auftreten. Wie nun Funktionen von Gegenstnden grundverschieden sind, so sind auch Funktionen, deren Argumente Funktionen sind und sein mssen, grundverschieden von Funktionen, deren Argumente Gegenstnde sind und nichts anderes sein knnen. Diese nenne ich Funktionen erster, jene Funktionen zweiter Stufe.

And, as if this was not confusing enough, he continues later:

> Man mu bei den Funktionen zweiter Stufe mit einem Argumente unterscheiden, je nachden als dies Argument eine Funktion mit einem oder eine solche mit zwei Argumenten erscheinen kann; denn eine Funktion mit einem Argumente ist so wesentlich verschieden von einer solchen mit zwei Argumenten, da die eine nicht an eben der Stelle als Argument auftreten kann, wo die andere es kann.

In my opinion, this makes *Frege* a very good name for a functional programming language with a strong type system.

Acknowledgments

Heartily thanks go to the whole functional language community and especially to the authors and contributors of *The Haskell 98 Report* [2] and *Haskell 2010 Language Report* [3]. This documents structure closely reproduces that of the latter one as knowledgeable people will easily spot.

I am especially grateful to Simon Peyton Jones, John Hughes and Philip Wadler. By publishing their knowledge and wisdom in numerous papers and books accessible through the internet these men of genius enrich the world and make it a better place.

Contents

List of Figures

Chapter 1

Introduction

FREGE is a functional language influenced by HASKELL with the following features:

- haskell (`www.haskell.org`) like syntax
- type safety through a strong type system with type inference. The type inference mechanism is based on and derived from the paper *Practical type inference for arbitrary-rank types* by Simon Peyton Jones [4], to whom I am greatly indebted.

  Type inference by the compiler means that it is almost never necessary to declare the type of variables, functions or expressions.
- lazy evaluation: expressions are only evaluated when they are needed.
- modularization through packages like in JAVA
- rich type system with basic types, functions, regular expressions, lists, tuples and user defined algebraic types. In addition, types from the host language may be used as abstract types.
- user definable operators
- type classes (interfaces) and instances (types that implement interfaces) provide a form of controlled polymorphism. For example, a sorting function may require that the values to be sorted must support comparisons. This is also a clean and type safe way to overload functions and operators.
- pattern matching with guards.
- interface to JAVA. In fact, FREGE is compiled to JAVA and all primitive types and operations are borrowed from JAVA.

If you know HASKELL or another functional language, FREGE will be easy to learn for you. This document contains boxes that highlight differences to HASKELL that look like this:

Difference to HASKELL 98/2010: Look for paragraphs like this to learn what is different in FREGE.

FREGE is

not object oriented

no replacement for already established functional programming languages like Haskell, Scala, F# and others. Nevertheless, FREGE may be interesting

- for JAVA programmers that are interested in pure functional programming.
- as a substitute for HASKELL when a functional programmer needs to do work in or for the JAVA platform.

1.1 Differences to Haskell 2010

Note: Readers not familiar with HASKELL may want to skip this section.

Module system FREGE's module system is based on that of JAVA. A FREGE program is is a collection of packages. Each FREGE source file defines exactly one package and compiles to a JAVA source file with the definition of a `public class`.

Types Numeric literals are not overloaded in FREGE.

Strings Strings are primitive types in FREGE and are implemented as JAVA's `java.lang.String` type. Conversions to and from lists of characters are provided.

Regex Another primitive FREGE type is `Regex`. It makes powerful and fast working functions on strings possible. A `Regex` can also be used as pattern for string arguments.

What Frege has and Haskell 98 does not have

- support for regular expressions in the language
- records with field labels that do not pollute the name space
- definitions that live in the scope of a data type
- pattern guards as proposed by Simon Peyton Jones in [8] and meanwhile implemented in HASKELL 2010.
- seamless access to any JAVA class and its public members and methods

1.2 Program structure

In this section, we introduce the language structure and at the same time give an outline of the organization of this document.

1. At the topmost level, a FREGE program is a set of *packages*, described in chapter 5.
2. The top level of a package consists of a collection of *declarations*, of which there are several kinds, all described in chapter 4. Declarations define things such as ordinary values and functions, data types, type classes, fixity information.
3. At the next lower level are *expressions*, described in chapter 3.
4. At the bottom level is the lexical structure, described in chapter 2.

The last section describes the native interface (chapter 8).

Examples of FREGE program fragments in running text are given in typewriter font. Sometimes examples are given in a form of colored pseudo code, with indexed identifiers in *italics* as in `if` e_1 `then` e_2 `else` e_3, where the italicized names are supposed to be mnemonic, such as e for expressions, p for patterns, etc.

Chapter 2

Lexical Structure

2.1 Notational Conventions

In this and the subsequent chapters, subsets of the FREGE grammar are given in running text. The complete grammar is the set of rules that is the union of all those subsets.

A grammar rule defines a nonterminal symbol as a sequence composed of terminal symbols, nonterminal symbols or subrules indicating that the enclosed sequence [is optional] or may occur {zero or more} times. Nonterminal symbols and variable terminal symbols are written in *italics*, constant terminals in bold `typewriter` font. The definition of a nonterminal starts in the left margin and may consist of alternative rules. Alternatives are separated by a line break, some indent and a vertical bar.

In this section particularly, the lexical syntax of terminal symbols of the grammar will be defined by regular expression. We use regular expressions as defined in the documentation of class `java.util.regex.Pattern` in [9]. Regular expression will appear in coloured typewriter font like this `\s?`.

In order to make things more readable, sometimes the name of a terminal symbol is used in a regular expression. The meaning is here to replace the terminal symbol with the regular expression defining it. For example:

digits:
        `\d+`
float:
        `digits(\.digits)?`
This is the same as:
float:
        `\d+(\.\d+)?`

Likewise, instead of `foo|bar` we sometimes write `foo`|`bar`.

All regular expressions are to be understood to be anchored at the start of the yet unprocessed portion of the program text.

Some symbols like parentheses, separators and so on stand for themselves and are specified verbatim each time they occur in the grammar. To distinguish verbatim symbols like `<-` `,` `;` `::` etc. from meta-syntactical symbols such as $\vert$ and $[\ ..\ ]$ and from regular expressions, we write them in a different colour.

Frege uses the Unicode character set.

Compilers will support program text stored in files with encodings that are supported by the Java platform. The standard encoding is UTF8. For detailed information consult the Java API documentation [9].

While it is possible to compose names and operator symbols from valid Unicode symbols, one should keep in mind that extensive use of this feature will make the program text difficult, if not impossible, to understand for members of different cultural background.

2.2 Lexical program structure

program:
$\{$ *line* $\}$
line:
$\{$ *whitespace token* $\}$ *whitespace*
whitespace:
`\s*`
token:
varid $\vert$ *conid* $\vert$ *keyword* $\vert$ *qualifier* $\vert$ *parentheses* $\vert$ *specialsym*
$\vert$ *lexop* $\vert$ *literal*

A program is made up of lines. Source code is broken into lines before tokenization by appropriate input reading functions that will recognize and strip line separator characters typical for the underlying operating system.

With the exception of documentation text there is no token that would extend over more than one line.

Each line contains zero or more tokens separated by whitespace. Still more whitespace can occur before the first token or after the last token.

Note that the definition of *whitespace* allows for the empty string of whitespace characters. Consequently, tokens may appear not to be separated at all.

The possibility of zero length whitespace does not mean that whitespace may be dismissed altogether. On the contrary, whenever two tokens appear in sequence where a non empty prefix of the second token might also be a valid suffix of the first one, nonempty whitespace is required to allow for unambiguous tokenization. In other words, the tokenization algorithm will recognize the longest prefix of the remaining characters on the current line that form a valid token.

Consider the following example:

Example:
a+2
a2
a 2
a
2

There are 3 tokens in the first line and one token in the second line. Since a digit is a valid suffix of an identifier, a space must occur between a and 2 to obtain two tokens, as shown on the third line. Another possibility to separate the tokens would be to write them on different lines, as shown in the last two lines.

2.3 Comments

Comments can appear everywhere whitespace can.

comment:
    linecomment | *blockcomment*
linecomment:
    `---?.*` (line comment extends to end of line)
blockcomment:
    `(?s)\{--?(.|`*blockcomment*`)*-\}`

Note that the character sequence making up a block comment may extend over multiple lines [1]. Because block comments do nest, any occurrence of `-}` or `{-` within the commented text will interfere with the nesting.

Difference to Haskell 98/2010:
A user defined operator (see section 2.5) must not start with the comment introducing characters {- or --.

2.3.1 Documentation Text

Block comments starting with `{--` and line comments starting with `---` are treated as *documentation text*. Unlike an ordinary comment, a documentation text is a token and hence is not only lexically but also syntactically relevant.

There are only certain places where documentation text may appear, as will be detailed in this section. In order not to complicate matters, subsequent sections will not mention documentation text anymore.

A documentation text may appear:

[1] This is the only exception to the rule that no token crosses line boundaries.

1. before the `package` keyword that starts a package. This will be the documentation for the package.

2. in place of a top level declaration or a declaration in the `where` clause of a data, class or instance declaration. It can also appear immediately before such a declaration. This will be the documentation for the subsequent declared item.

3. immediately either before or after a *constructor* in a data declaration. This will be the documentation for that constructor.

4. immediately before or after a constructor field. In the latter case, no comma must be written before the next constructor field.

For convenience, in cases 1 and 2 a sequence of documentation comments optionally separated by semicolon can be written. The text of the documentation comments will be concatenated with an interleaving paragraph break.

Example:

```
--- this package is documented
{-- second paragraph of package doc -}
{-- third paragraph of package doc-}
package D where

--- this is the list of Fibonacci numbers
fib = 1:1:zipWith (+) fib (tail fib)
--- document type D
data D =
    --- document constructor C
     C { name ::  String --- document name, no comma here
         {-- document age -}
         age ::  Int }
    | N     --- document constructor N
```

Documentation text will be copied verbatim and it will be available in the binary results of compilation (e.g. JAVA class files), so that documentation processing tools can access it to generate documentation in various formats.

2.4 Identifiers and Keywords

qualifier:
```
\p{Lu}(\d|_|\p{L})*\.
```
varid:
```
\p{Ll}(\d|_|\p{L})*'*
```

conid:

```
\p{Lu}(\d|_|\p{L})*
```

qvarid:

qualifier qualifier varid | *qualifier varid* | *varid*

qconid:

qualifier qualifier conid | *qualifier conid* | *conid*

These rather complicated regular expressions deserve some further explanation.

We distinguish lexically between two classes of identifiers. Names for functions, values and local variables are *varid*s and start with a lowercase letter. Names that start with an uppercase letter (*conid*s) stand for value constructors, type constructors, type classes, type aliases or name spaces.

Sometimes it is necessary to name an item that is defined in another package or in the scope of a type or type class. Thus we need qualified names, defined here as *qvarid* and *qconid*. They are formed by writing one or two *qualifier*s before a *varid* or *conid*.

A *qualifier* consists of an identifier starting with an uppercase letter and an immediately following dot. The identifier may denote name spaces, types or type classes. A *qualifier* like `Foo.` is a single token and thus may not contain spaces.

According to this, the syntax allows reference to items in the following ways:

$N.T.v$ $N.T.C$
$T.v$ $T.C$ $N.v$ $N.C$ $N.T$
v C T

where N would be a name space, T a type or class name, C a data constructor name and v the name of a function or pattern binding.

There are rare cases where it is possible to confuse the dots in the qualifiers with the special operator `.` explained later, an example can be found here. Fortunately, such constructs can be disambiguated with spaces or parentheses.

Note: Unlike in other languages, a FREGE identifier cannot start with an underscore.

Name Resolution and Scope

Names appearing in expressions and types are resolved by the following rules, where N, T and C stand for *conid*s and v for *varid*s:

Names of the form v: every enclosing lexical scope provided by a `let`, lambda expression or case alternative is searched in turn for the name. If it is found, then it refers to an item defined in a `let` expression or a (part of a) pattern in a lambda expression or case alternative. Otherwise, it must be a globally visible item. If v

appears in the scope of a data definition, class definition or instance definition and there is a variable or function binding with the name v then it is resolved to mean this binding except when this is an implementation of a type class operation which has a simple name in the current package. In that case, the name resolves to that class operation. Otherwise, it must be a global function or variable binding or a class member.

Names of the form T or C: T may appear in type signatures, where it denotes a type constructor, type name or class name, either an imported one or one that is declared in the current package. In expressions and patterns, C denotes a value constructor.

Names of the form $N.T$ or $N.C$: N must be a name space denoting an imported package, a data type or a class. T must be a class name, type name or C must be a data constructor from this name space. While it is possible, that a type and a data constructor have the same name this does not introduce ambiguities because syntactically either a type name T or a data constructor C can be meant, but not both.

It is also possible that a type name and a name space of an imported package have the same name. In this case, only the name space of the imported package is searched. If one needs to access C in the name space of the type $N.N$ one needs to write a qualified name of the form $N.N.C$.

Names of the form $N.v$ or $T.v$: N must be name space denoting an imported package or T must denote a data type, type alias or a class. v is the name of a function or pattern binding in N or T. Again, if there is a name space N and a type T and $N = T$, then only N is searched.

Names of the form $N.T.C$ or $N.T.v$: Fully qualified names denote a function or pattern binding or a data constructor belonging to type or class T from name space N.

Keywords

Some character sequences that would otherwise be matched by rule *varid* are keywords and will be recognized by the scanner as distinct terminal symbols.

```
abstract: abstract
case: case
class: class|interface
data: data
derive: derive
do: do
else: else
false: false
forall: forall
```

```
if: if
import: import
in: in
infix: infix
infixl: infixl
infixr: infixr
instance: instance
let: let
mutable: mutable
native: native
of: of
package: package|module
private: private
protected: protected
pure: pure
public: public
then: then
throws: throws
true: true
type: type
where: where
```

The words `pure` and`mutable` are only recognized as keywords when immediately followed by the `native` keyword, otherwise they are regarded as ordinary *varid*.

2.5 Operators

The FREGE language permits user defined infix operators. Valid infix operators are sequences of either letters or non word characters that obey the following additional rules:

1. Certain sequences of 1 or 2 non word characters form terminal symbols with special syntactic meaning (rule *specialsym*). These symbols can not be used as infix operators.
2. Operator symbols may not contain characters used as quotation marks (rule *quotechar*).
3. Operator symbols may not contain parentheses, brackets or braces (rule *parentheses*).

An infix operator denotes a function or a value constructor.

Operators may be introduced with a top level infix declaration (rule *fixity*).

fixity:
    infix precedence lexop { *lexop* }
infix:
    `infix` | `infixl` | `infixr`
precedence:
    `[123456789]` | `1[0123456]`
symop:
    `\W+`
wordop:
    `\w+`
infixop:
    `symop` | `wordop`
lexop:
    `` `infixop` `` | `symop`
specialsym:
    `::` | `->` | `<-` | `=>` | `\|` | `=` | `-` | `!` | `?` | `,` | `;` | `\.` | `\\` | `_`
parentheses:
    `\(` | `\)` | `\[` | `\]` | `\{` | `\}`
quotechar:
    `["'#` ` ]`

The infix declaration has two purposes:

- It makes the lexical analyzer recognize operator symbols made up of symbol characters (rule *symop*). The lexical analyzer recognizes only operator symbols introduced in an infix declaration and operator symbols from imported packages (see also subsection 2.5.2).
- It causes the parser to interpret expressions differently based on the operators associativity (left, right or none) and precedence. Operators with higher precedence bind their operands before operators with lower precedence, so the precedence is to be taken as an indication of binding power. Precedences range from 1 (weakest binding) to 16 (tightest binding).

See also the syntax of *binary expressions* in section 3.5, the example in Figure 2.1 and the table of predefined operators in Figure 3.3.

2.5.1 Rules for using backquotes

Every sequence of characters forming a valid operator symbol that is enclosed in backquotes will be recognized as an operator token. If the operator was not previously introduced through a fixity declaration it will be assumed that it is non-associative and has a precedence of 16.

```
infix 12 `==` `!=`        -- non associative
infixr 13 `++`            -- right associative
infixl 14 div             -- left associative word operators,
infixl 14 `mod`           --  backticks don't matter here
infixr 4 `:`              -- right associative
infixr 16 `**`            -- ditto, but binds tighter than `:`

a == b != c               -- syntax error, set parentheses explicitly
a ++ b ++ c  == d         -- (a ++ (b ++ c)) == d
a ** b ** c : d : e       -- (a ** (b ** c)) : (d : e)
a `mod` b                 -- mod a b
f div 2                   -- div is not used as operator here
```

Figure 2.1: Parsing of expressions containing operators

As outlined above, a *symop* not enclosed in backquotes can only be recognized when there is a fixity declaration or an import that introduces it. Hence, to introduce a fresh *symop* one must write it within backquotes in the fixity declaration itself.

For *wordop*s matters are different. Like in HASKELL it is required that one always explicitly indicates when one wants to *use* an identifier as operator. Thus, *wordop*s must always be quoted with backquotes when they are in infix position. However, in the infix declaration all that matters is to announce the character sequence an operator is made of. Thus, backticks are not strictly needed when introducing word operators.

2.5.2 Imported operators

A package import (see also section 5.3) makes all operator symbols introduced in the imported package known to the lexical analyzer. Yet, depending on the import statement, the corresponding function may not be in scope. To access them nevertheless, it is possible to qualify operators:

qlexop:

qualifier lexop | *qualifier qualifier lexop*

Difference to HASKELL 98/2010:

- fixity is a lexical and syntactical property of certain operator symbols
- (consequently) fixity declarations are permitted at top level only
- an operator whose fixity was not declared is taken to be non-associative and to have precedence 16 (tightest biding)
- to use an operator `op` from name space `M` one writes `M.` `` `op` ``

2.6 Unary operators

There are two symbols that may be used as unary operators:

unop:
`!` | `?`

Unary operators can not be qualified. It is strongly discouraged to use them as names for own functions.

The unary opeartor `!` is the boolean negation function; in patterns it has special meaning that signals strict patterns.

The unary operator `?` is currently unassigned and reserved for future use.

2.7 Literals

Literals are textual representations of values of certain simple types. All literals are valid expressions as well as patterns.

literal:
boolliteral | *numericliteral*
| *charliteral* | *stringliteral* | *regexliteral*
numericliteral:
integerliteral | *floatliteral*

Difference to HASKELL 98/2010: Literal syntax is adopted from JAVA. Every literal determines a fixed type.

2.7.1 Boolean Literals

The boolean values are represented by the keywords `true` and `false`. Boolean values are of type `Bool`.

boolliteral:
`true` | `false`

2.7.2 Numeric Literals

The syntax of numeric literals follows closely that of JAVA, except that some exotic form of floating point literals are not supported. In addition, there are literals for big integers.

Furthermore, for all numeric literals, the syntax of the integral part has been slightly extended: it is possible to separate trailing groups of 3 digits each with an underscore. This enhances legibility greatly with big numbers.

> Example: The literal for the long integer value fifty-two billion four hundred and twenty-five million two hundred and fifty-four thousand five hundred and twenty-four can be written `52_425_254_524L` or `52425254524L`.

Integer Literals

There are literals for values of 3 different integer types: `Int`, `Long` and `Integer`.

integerliteral:
    intliteral | *longliteral* | *bigintliteral*
intliteral:
    as defined in JAVA, see section 3.10.1 in [6]
longliteral:
    as defined in JAVA, see section 3.10.1 in [6]
bigintliteral:
    `\d+(_\d\d\d)*[nN]`

FREGE adopts the syntax for integer literals from JAVA. An integer literal that would have type `int` in JAVA has type `Int` in FREGE. An integer literal that would have type `long` in JAVA has type `Long` in FREGE.

In addition, a sequence of decimal digits followed by one of the letters `n` or `N` (think *natural* number) is a literal of type `Integer`, the data type of integral numbers of arbitrary size. Note that leading zeros do not indicate octal numbers as with the other integer literals.

Floating-Point Literals

There are literals for values of the `Float` and `Double` types. The syntax is a subset of that for JAVA floating point literals as specified in section 3.10.2 of [6]. Not supported are floating point literals that do not start with a digit and hexadecimal floating point literals.

floatliteral:
    as defined in JAVA, see section 3.10.2 in [6]
    except hexadecimal notation and literals
    that start with a decimal point

A literal that would have type `float` in JAVA has type `Float` in FREGE. A literal that would have type `double` in JAVA has type `Double` in FREGE.

2.7.3 Character and String Literals

Character Literals

Character literals are like `char` literals in JAVA and have type `Char`.

charliteral:
as defined in JAVA, see section 3.10.4 in [6]

> Note: Since FREGE does not preprocess its source texts, a character literal like `'\u89ab'` will not work.

String Literals

String literals are like `String` literals in JAVA and have type `String`.

stringliteral:
as defined in JAVA, see section 3.10.5 in [6]

> Note: JAVA programmers: Please observe that the string concatenation operator is `++` in FREGE

.

Literals for Regular Expressions

The FREGE language supports regular expressions as a built in data type. Consequently it is possibe to specify regular expressions literally. Such literals denote values of type `Regex` unless they are not well formed by the rules of the regular expression language. In the latter case, the compiler issues an error message and the program containing the ill formed literal does not compile.

regexliteral:
`(\\|[^\])*`

A regular expression literal is enclosed in grave accent marks and is a sequence of 0 or more characters that are not grave accent marks unless they are escaped with backslashes.

The regular expression language is the one implemented in the `java.util.regex` package. It is documented along with the class `java.util.regex.Pattern` in [9]. The only difference is that the grave accent mark is a special character that signals the end of the regular expression. If one wants to match a grave accent mark, one must write a backslash followed by a grave accent mark in the regular expression.

Regular expression literals are compiled with the flags `CANON_EQ`, `UNICODE_CASE` and `UNICODE_CHARACTER_CLASS`. The latter two may be reset with embedded flag expressions `(?-u)` and `(?-U)`, respectively.

> Note: A single backslash in a regex literal is the escape character for the regular expression language. Thus, for instance, the literal `\ba` means *"the regular expression that matches the letter 'a' after a word boundary"* and not *"... that matches the backspace character followed by the letter 'a'"*.

It is also possible to construct a string that contains a pattern and compile that to a pattern value. However, regular expression literals are superior compared to string literals with pattern text

- because there is one level of backslash-interpretation less, thus one needs to write only half the number of backslashes
- invalid regular expression literals are flagged at compile time, not when they are about to be used
- regular expression literals will be replaced with references to read only pattern values that are built at program startup time. Thus one can safely use regular expression literals everywhere without performance penalty due to repeated pattern compilation. This has the added benefit that one can immediately see what the regular expression is and does not have to look it up somewhere else in the program code.

The bottom line is: one should use regular expression literals whenever possible.

2.8 Layout

FREGE permits the omission of the braces and semicolons by using *layout* to convey the same information. This allows both layout-sensitive and layout-insensitive styles of coding, which can be freely mixed within one program. [2] Because layout is not required, FREGE programs can be straightforwardly produced by other programs.

Informally stated, the braces and semicolons are inserted as follows. The layout (or "offside") rule takes effect whenever the open brace is omitted after the keyword `where`, `let`, `do`, or `of`. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments). For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted).

The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace. Within these explicit open braces, no layout processing is performed for constructs outside the braces, even if a line is indented to the left of an earlier implicit open brace.

[2] Though, experience with HASKELL seems to show that literally all source code is written using layout.

Difference to HASKELL 98/2010: The token following a `where`, `let`, `do` or `of` keyword must either be an opening brace or it must be more indented than the current level, otherwise the layout algorithm will insert a closing brace which will result in a syntax error. In other words, the token sequence $\{\}$ will never be inserted. The layout handling is a purely lexical matter, hence it is not possible to insert a closing brace *"if an illegal lexeme is encountered at a point where a closing brace would be legal"*.
However, a closing brace is inserted before the keyword `in` regardless of the indentation unless it is already preceded by a closing brace. Hence, one can still write let expressions on a single line.

Chapter 3

Expressions

In this chapter, we describe the syntax and informal semantics of FREGE *expressions* and give translations into more basic expressions, where appropriate. Free variables and constructors used in these translations refer to entities defined by the standard package `frege.Prelude`.

3.1 Terms

term:

qvarid	(variable)
qconid	(constructor)
qconid { *initfields* }	(value construction)
literal	
_	(a single underscore)
(*qlexop*)	
(*qunop*)	
section	
( )	(unit constructor)
tuple	(see subsection 3.1.3)
list	(see subsection 3.1.4)

initfields:

initfield { , *initfield* }

initfield:

varid [= expr]

The most basic expressions are variables, value constructors and literals. Variables stand for the values they are bound to, often these values are functions. Likewise, a value constructor may stand for a value or for a function that constructs values depending on

other values.

Constructors that were defined with field labels can be applied to a list of field initializations enclosed in braces. Exactly the field labels that belong to the constructor must be specified. If the expression is omitted, the value of the variable currently in scope that has the same name as the field is used (punning). The expression is translated to an ordinary constructor application by reordering the expressions given for the fields so that they appear in the same order as in the constructor definition.

Translation:

Con $\{ \cdots, a, \cdots \}$	$=$	Con $\{ \cdots, a = a, \cdots \}$
Con $\{ a = x_a, c = x_c, b = x_b \}$	$=$	(Con x_c x_b x_a) if the order of the fields in the constructor was c, b, a

Difference to Haskell 98/2010: Values for all fields must be given, it is not allowed to leave out fields.

Literals stand for the value they represent.

A single underscore is technically a variable, yet can only appear in pattern bindings. There it signals that the corresponding part of a value is not bound.

Sometimes one needs to refer to a function whose name is an operator lexically. Any unary or infix operator may be used as a variable or constructor by enclosing it in parentheses. Another way to put this is to say that an operator enclosed in parentheses looses its syntactic properties.

Translation:

(op)	$=$	*the function defined by* op, where op is an operator
e_1 op e_2	$=$	(op) e_1 e_2, where op is a binary operator
f op e	$=$	f $((op)$ $e)$, where op is an unary operator

An expression of arbitrary complexity becomes a term syntactically when enclosed in parentheses.

Certain constructs exist to deal with partially applied binary operators and the data types "built in" the language, namely tuples and lists. They all are terms syntactically and are explained in the following subsections.

3.1.1 Sections

section:

(*binex lexop*)
| (*lexop expr*)

So called *sections* are means of creating functions on the fly by providing the first or second argument to an infix operator (see section 2.5), that is, to a function that takes (at least) two arguments.

For example, the expression $(e < 42)$ denotes one of the values `true` or `false`, depending on the value of e. We can now abstract out either side of the expression to get two functions:

1. a function that checks whether its argument is lower than 42
2. a function that checks whether e is lower than its argument

Sections permit us to do that syntactically by just leaving out the subexpression we abstract from. We write (< 42) for the first function and $(e <)$ for the second.

Translation: The following identities hold:

```
(- e)      =  negate e
(`op` e)   =  \x → x `op` (e)
(e -)      =  \x → e - x
(e `op`)   =  \x → e `op` x = (`op`) (e)
```

where *`op`* is a binary operator, *e* is an expression and *x* is a variable that does not occur in *e*

The precedence of the operator in a section where the operator comes first is irrelevant. It is so as if the subexpression in the section were always written in parentheses.

However, if the operator stands on the right, its precedence must be taken into account.

For example,the function $(*1 + x)$ multiplies its argument with $(1 + x)$. But $(1 + x*)$ is a syntax error. It must be written like this: $((1 + x)*)$.

Special role of the subtraction/negation operator

The operator $-$ is treated specially in the grammar. It can act as binary operator, indicating subtraction, or as an unary operator, in which case it indicates negation. The token $-$ will *always* refer to the Prelude definition of either subtraction or negation, it is not possible to redefine or qualify it.

Because $e1 - e2$ parses as an infix application of the binary operator $-$, one must write $e1(-e2)$ for the alternative parsing. Similarly, $(-)$ is syntax for $(\backslash x \backslash y \to x - y)$, as with any infix operator, and does not denote $(\backslash x \to -x)$ – one must use `negate` for that.

Finally, $(-e)$ is not a section, but an application of prefix negation, as described above. However, there is a `subtract` function defined in the Prelude such that (`subtract` e) is equivalent to the disallowed section. The expression $(+(-e))$ can serve the same purpose.

3.1.2 Unit

The *unit expression* `()` has type () (see section 6.1.4); it is the only value of that type.

3.1.3 Tuples

For a discussion of tuple types see section 6.1.4.

tuple:
    tupleconstructor | *n-tuple* | *strict-n-tuple*
tupleconstructor:
    `(` { `,` } `)`
n-tuple:
    `(` *expr* `,` *expr* { `,` *expr* } `)`
strict-n-tuple:
    `(` *expr* `;` *expr* { `;` *expr* } `)`

Tuples are written $(e_1, ..., e_k)$ where $2 \leq k \leq 26$. The constructor for an n-tuple is denoted by $(,...,)$ where there are $n-1$ commas.

Translation: The following identities hold:

(e_1, e_2)	=	$(,)\ e_1\ e_2$
(e_1, e_2, e_3)	=	$(,,)\ e_1\ e_2\ e_3$
... and so forth up to		
$(e_1, e_2, ..., e_{26})$	=	$(,,,,,,,,,,,,,,,,,,,,,,,,,)\ e_1\ e_2\ ...\ e_{26}$

Tuple construction does not normally cause evaluation of the tuple elements e_i. We say that tuple construction is lazy. Sometimes, though, this laziness is not desired. For example, if it is known to the programmer that the tuple elements will be evaluated sooner or later anyway, it can be a good idea to force strict evaluation. For this purpose, replace the commas with semicolons in the tuple construction expression.

3.1.4 Lists

Lists are also discussed in section 6.1.4.

list:
    `[]` (constructor for the empty list)
    | `[` *expr* { `,` *expr* } `]`
    | `[` *expr* [`,` *expr*] `..` [*expr*] `]` (arithmetic sequence)
    | `[` *expr* `|` *dlcqual* { `,` *dlcqual* } `]` (list comprehension)
dlcqual:

pattern ← *expr*
| *expr*
| `let` { *decl* { ; *decl* } }

Lists are written $[e_1, \cdots, e_k]$, where $k >= 1$. The list constructor : as well as the constructor for the empty list `[]` is considered part of the language syntax and cannot be hidden or redefined.

Translation: The following indentity holds:
$[e_1, e_2, \cdots, e_k] = e_1 : (e_2 : (\cdots (e_k : [])))$
The types of e_1 trough e_k must all be the same (call it t), and the type of the expression is $[t]$.

Arithmetic Sequences

The *arithmetic sequence* `[`e_1 [, e_2] .. [e_3]`]` denotes a list of values of type t, where each of the e_i has type t, and t is an instance of type class `Enum`.

Translation: Arithmetic sequences satisfy these identities:

`[`e_1 `..]`	=	enumFrom e_1
`[`e_1`,` e_2 `..]`	=	enumFromThen e_1 e_2
`[`e_1 `..` e_3`]`	=	enumFromTo e_1 e_3
`[`e_1`,` e_2 `..` e_3`]`	=	enumFromThenTo e_1 e_2 e_3

The semantics of arithmetic sequences therefore depends entirely on the instance declaration for the type t .

The intended semantics for *enumFrom* is to evaluate to a list that consists of the start element e_1, followed by all its successors in ascending order, while *enumFromTo* restricts the sequence of successors up to and including e_3. The *enumFromThen* and *enumFromThenTo* functions are variants thereof that produce successive elements whose ordinal values differ by the same amount as the ordinal values of the first two. Hence

Example: enumFrom a = enumFromThen a (succ a)

It is also possible to create decreasing sequences, by giving an e_2 that is smaller than e_1.

List Comprehensions

List comprehension is a powerful notation to describe lists based on other lists. It has the form $[e|q_1, \cdots, q_n]$, where $n >= 1$ and the qualifiers q_i are either

- *generators* of the form $p \leftarrow e$, where p is a pattern (see subsection 3.11.1) of type t and e is an expression of type $[t]$.

- *guards*, which are arbitrary expressions of type `Bool`.
- *local bindings* `let` { *decl* {; *decl*} }
  Because list comprehension qualifiers are separated with commas and the keyword `in` is missing, it is not possible to invoke the layout rule by omitting the braces in this case. For convenience, a single binding `let` { *decl* } can be written simply *decl*.

Such a list comprehension returns the list of elements produced by evaluating e in the successive environments created by the nested, depth-first evaluation of the generators in the qualifier list. Binding of variables occurs according to the normal pattern matching rules (see section 3.11), and if a match fails then that element of the list is simply skipped over. Thus:

Example: [x | (2, x) ← [(1, 'a'), (2, 'b'), (3, 'c'), (2, 'd')]]

yields the list `['b', 'd']`.

If a qualifier is a guard, it must evaluate to `true` for the previous pattern match to succeed. As usual, bindings in list comprehensions can shadow those in outer scopes; for example:

Example: [x | x ← x, x ← x] = [z | y ← x, z ← y]

Translation: A list comprehension can be rewritten by the following translation scheme:

```
LC [e | Q ]                 =  TQ [e | Q ] []
TQ [e | ] L                 =  (e) : L
TQ [e | b, Q] L             =  if b then TQ [e | Q] L else L
TQ [ e | p ← xs, Q ] L      =  let
                                   h [] = L
                                   h (p:ys) = TQ [e | Q] (h ys)
                                   h (_:ys) = h ys
                               in h xs
TQ [ e | let { decls }, Q ] L  =  let { decls } in TQ [e | Q] L
```

where e ranges over expressions, p ranges over patterns, xs ranges over list valued expressions, b ranges over boolean expressions, Q ranges over sequences of qualifiers and `h` and `ys` are fresh variables.

Example Let's translate the following program fragment:

```
nums = [1,2,3,4,5,6,7,8,9,10]
squares = [ n*n | n <- nums, n > 4 ]
```

We have first

```
nums = 1:2:3:4:5:6:7:8:9:10:[]
```

Note that we omitted the parentheses from the translation rule, which is ok since : is right associative.

Next we apply translation scheme LC to the list comprehension. This gives us
TQ [n*n | n ← nums, n > 4] []

The TQ scheme has 4 rules that cover all possible forms of a qualifier list. Our qualifier list starts with a generator, thus we have to apply the 3rd rule substituting `n*n` for e, `n` for p and `nums` for xs. L is the empty list and Q stands for the rest of our qualifier list, which is `n>4`. We get

```
squares = let
   h1 [] = []
   h1 (n:ys1) = ... // translation of TQ [ n*n | n>4 ] (h1 ys1)
   h1 (_:ys1) = h1 ys1
 in h1 nums
```

We now have to translate TQ [n*n | n > 4] (h1 ys1) by the second rule, since the qualifier list starts with a guard. We substitute e with `n*n` as before, b is the guarding expression `n>4`, Q is empty and L stands for the expression `(h1 ys1)`. The result is

```
if n>4 then TQ [ n*n | ] (h1 ys1) else (h1 ys1)
```

and after translating the last remaining TQ scheme by the first rule, the complete program is

```
nums = 1:2:3:4:5:6:7:8:9:10:[]
squares = let
   h1 [] = []
   h1 (n:ys1) = if n>4 then (n*n) : (h1 ys1) else (h1 ys1)
   h1 (_:ys1) = h1 ys1
 in h1 nums
```

Difference to Haskell 98/2010: In Frege, the types `Double` and `Float` have no standard instances for `Enum`, hence floating point arithmetic sequences are not available out of the box.

3.2 Primary Expression

Primary expressions are used for invocation of member functions, field selection, field update, array access and array update using a concise notation that will be translated by the compiler. The translations are given below in the usual way.

primary:

term

primary `.` *varid*	(member function application)
primary `.` *lexop*	(member function application)
primary `.` *unop*	(member function application)
qualifier {*varid* `?`}	(general field existence function)
qualifier {*varid* `=`}	(general field update function)
qualifier {*varid* `<-`}	(general field change function)
qualifier {*field* { `,` *field* }}	(update/change values function)
primary `.` {*varid* `?`}	(field existence check)
primary `.` {*varid* `=`}	(update field in value function)
primary `.` {*varid* `<-`}	(change field in value function)
primary `.` {*field* { `,` *field* }}	(value update/change)
primary `.` `[`*expr*`]`	(array element selection)
`do` { *dlcqual* { `;` *dlcqual* } }	(monadic expression)

field:

varid `=` *expr*	(field update)
varid `<-` *expr*	(field change)
varid	(field update with punning)

The syntax reveals that the `.` works like a left associative operator. This is so that primary expressions can be chained, much like in conventional programming languages. For example

```
a.[42].age.negate
```

could be the negated value associated with the `age` field of the value in the 43rd element of the array `a`.

3.2.1 Special interpretation of the `.`

The single symbol `.` as syntactic element used to form primary expression will only be recognized if the following conditions all hold:

- it is not surrounded by white space on both sides (line breaks count as whitespace)
- the previous token, if any, is not `(`
- the next token, if any, is not `)`

In all other cases, a single `.` will be recognized as •, which is the function composition operator.

These rules do not affect recognition of operators that contain one or more dots.

Difference to HASKELL 98/2010: The single . is an overloaded character. If it is enclosed within spaces or if it looks like a section, it will be as if the function composition operator •[1] had been typed in its place. In all other cases, it is taken as part of a primary expression or a qualified name.
This is to maximize compatibility with HASKELL, or to make it easy at least to port HASKELL source code.

3.2.2 Member function application

This is a convenient notation to apply functions defined locally to a name-space such as a data type or type class. This syntactic form is translated like this:

Translation:

$e.m$	$=$	$(T.m\ e)$	if the expression e has type t and the type constructor of t is T and there exists a function $T.m$
	$=$	$(C.m\ e)$	if m is an overloaded function belonging to type class C

The conditions will be checked in the order listed here and the first possible translation will be selected. If none of the conditions hold, the compiler flags a type error.

Because the compiler creates functions for access to fields in algebraic data types with field labels that happen to have the same name as the field label, this syntax is most prominent for extraction of field values.

For lexical reasons, when e is a nullary value constructor such as `Nothing` one cannot write for instance `Nothing.show` as a shorthand for `Show.show Nothing` This is because the lexical analyzer will tokenize this as *qvarid* and then during name resolution `Nothing` would be interpreted as name of a type or name space (which would probably fail). One can avoid this by writing one of `Nothing . show` (with space before the dot so as to make interpretation as qualified variable impossible) or `(Nothing).show`

3.2.3 Field Existence Test

The expression $T.\{field?\}$ denotes the function that checks, whether a value v with a type $Tu_1 \cdots u_k$ where $k >= 0$ was constructed with a data constructor in whose field list the label $field$ appears.

The expression $v.\{field?\}$ can be used in place of $(T.\{field?\}\ v)$ if the type of v is already known. The type of v can not be inferred from the former expression, only from the latter one.

[1] Unicode `U+2022`. Can be produced in Windows with ALT+`0149` (type number on the numeric keypad), in Linux depending on the input method with COMPOSE+`.=`

Translation: Let it be known that v is of type t, and let t be an algebraic data type with constructors $C1$, $C2$ and $C3$. Let label $field$ appear in $C1$ and $C2$. Then the following transformation can be applied:

$$
\begin{array}{lcl}
T.\{field?\} & = & \backslash v \rightarrow \text{case } v \text{ of } \{ \\
 & & C1 \cdots \rightarrow \textbf{true}; \\
 & & C2 \cdots \rightarrow \textbf{true}; \\
 & & C3 \cdots \rightarrow \textbf{false } \} \\
v.\{field?\} & = & (T.\{field?\}\ v) \qquad \text{– if } v \text{ has type } Tu_1 \cdots u_k
\end{array}
$$

The $\cdots$ following the constructors represent the correct number of _-patterns for the respective constructor.

If $(T.\{field?\}\ v)$ returns `true`, all field update and access functions for $field$ can be applied to v. If it is `false`, however, application of any of those functions results in a pattern match failure.

Example:

```
data T = C1 {name::String, age::Int}
chgName ::  T -> String -> T
chgName = T.{name=}
nullifyName ::  T -> T
nullifyName = T.{name=""}
age10Factory ::  String -> T -- where age = 10
age10Factory = C1 {name="", age=10}.{name=}
incrAge ::  Int -> T -> T
incrAge n = T.{age <- (n+)}
```

Figure 3.1: Some primary expressions and their types

3.2.4 Value Update and Change by Field Label

The expression $T.\{field=\}$ denotes a function that, when applied to two values v and u, creates a new value of type $Tu_1 \cdots u_k$ (where $k >= 0$) that differs from v only in the value for field $field$, which is set to u. This implies that v is also of type $Tu_1 \cdots u_k$ (where $k >= 0$) and u of the type that was given for label $field$ in the declaration of T.

The expression $T.\{field\leftarrow\}$ denotes a function that, when applied to two values v and g, creates a new value of type $Tu_1 \cdots u_k$ (where $k >= 0$) that differs from v only in the value for field $field$, which is set to the result of g applied to the value of $field$ in v. This implies that v is also of type $Tu_1 \cdots u_k$ (where $k >= 0$) and g of the type $(t \rightarrow t)$, where t was the type given for label $field$ in the declaration of T.

In both cases, v must have been constructed with a data constructor in whose field list the label $field$ appears. Otherwise, the result is undefined, and an attempt to evaluate

it will cause a pattern match failure.

Translation: Let it be known that v is of type T, and let T be an algebraic data type with data constructors C_1, C_2 and C_3. Let label f appear as the first of two fields in C_1 and as the second of three fields in C_2[2]. Then the following transformation is applied:

$$
\begin{array}{lcl}
T.\{f{=}\} & = & \backslash v \backslash u \to \textbf{case } v \textbf{ of } \{ \\
 & & \quad C_1\ a_1\ a_2 \to C_1\ u\ a_2; \\
 & & \quad C_2\ a_1\ a_2\ a_3 \to C_2\ a_1\ u\ a_3\ \} \\
T.\{f{\leftarrow}\} & = & \backslash v \backslash g \to \textbf{case } v \textbf{ of } \{ \\
 & & \quad C_1\ a_1\ a_2 \to C_1\ (g\ a_1)\ a_2; \\
 & & \quad C_2\ a_1\ a_2\ a_3 \to C_2\ a_1\ (g\ a_2)\ a_3\ \} \\
v.\{f{=}\} & = & (T.\{f{=}\}\ v) \\
v.\{f{\leftarrow}\} & = & (T.\{f{\leftarrow}\}\ v) \\
T.\{f = u\} & = & (\text{flip } T.\{f{=}\}\ u) \\
T.\{f \leftarrow g\} & = & (\text{flip } T.\{f \leftarrow\}\ g) \\
\text{flip } f\ a\ b & = & f\ b\ a \text{ – standard function} \\
v.\{f = u\} & = & (T.\{f{=}\}\ v\ u) \\
v.\{f \leftarrow g\} & = & (T.\{f \leftarrow\}\ v\ g) \\
v.\{f_1 = u_1, & = & v.\{f_1 = u_1\} \text{ – general rules for chained changes/updates} \\
\quad f_2 \leftarrow u_2, & & \quad .\{f_2 \leftarrow u_2\} \\
\quad f_3 = u_3, \cdots\} & & \quad .\{f_3 = u_3\}.\ \cdots \\
T.\{f_1 = u_1, & = & T.\{f_1 = u_1\} \\
\quad f_2 \leftarrow u_2, & & \quad \bullet\ T.\{f_2 \leftarrow u_2\} \\
\quad f_3 = u_3, \cdots\} & & \quad \bullet\ T.\{f_3 = u_3\}\ \bullet\ \cdots
\end{array}
$$

The a_i are auxiliary variables appearing nowhere else in the program.
The identities make it clear that values constructed with C_3 cannot be changed or updated, as this constructor has no field f.

Figure 3.2: Translation of change/update primary expressions

The update and change functions are created by the compiler whenever an algebraic data type has a constructor with field labels. They are employed by several forms of the primary expression, whose translation is given in Figure 3.2.

The last 2 rules of Figure 3.2 show that many changes and updates can be mixed in one pair of braces.

As before, in constructs like $v.\{field \cdots\}$ where v is an expression, the type of v can not be determined from that construct alone by the type checker. The type of v must be known, either trough a type annotation or the type of v must be unambiguously clear for other reasons.

[2] We pick here an arbitrary case for demonstrating purposes. The translation works no matter how many constructors exists, how many of them have a certain field and how many other fields exist. It is precisely the independence of the actual representation of the values (viz. the type signatures of the constructors) that make the label notation a valuable feature.

3.2.5 Array Element Selection

TODO: write me

3.2.6 Monadic Expression

A *monadic expression* provides syntactic sugar for application of the monadic operators `>>` and `>>=`. Note that the qualifiers share the same syntax with list comprehensions qualifiers but that the translation is quite different.

It is transformed according to the following rules:

Translation:

$$
\begin{array}{lcl}
\texttt{do}\ \{\ e\ \} & = & e \\
\texttt{do}\ \{\ e;\ q_2;\ \cdots;\ q_i\ \} & = & e\ \texttt{>>}\ \texttt{do}\ \{\ q_2;\ \cdots;\ q_i\ \} \\
\texttt{do}\ \{\ p \leftarrow e\ ;\ q_2;\ \cdots;\ q_i\ \} & = & e\ \texttt{>>=}\ (\backslash p \rightarrow \texttt{do}\ \{\ q_2;\ \cdots;\ q_i\ \}) \\
\texttt{do}\ \{\ \texttt{let}\ \{\ d_1;\ \cdots;\ d_i\ \}; & & \\
\qquad q_2;\ \cdots;\ q_i\ \} & = & \texttt{let}\ \{\ d_1;\ \cdots;\ d_i\ \} \\
 & & \texttt{in do}\ \{\ q_2;\ \cdots;\ q_i\ \}
\end{array}
$$

where the d_i range over local declarations, $q_2; \cdots; q_i$ is a non empty list of arbitrary qualifiers, e stands for an expressions and p for a pattern.

The first rule states that the last qualifier in a monadic expression must be an expression. The remaining rules give the translation for each possible form of the first qualifier.

The type and meaning of the monadic operations are defined in the type class Monad and in the instances for Monad.

3.3 Unary operator application

unary:
primary
| *qunop unary*

The unary operators are just functions with a funny name, however, the precedence of unary operator application is even higher than that of normal function application (see next section). This is so that one can write `foo !a` and this will be parsed like `foo (!a)`.

3.4 Function application

appex:
unary
| *appex unary*

Function application is the most basic notion in a functional programming language and is written $e_1\ e_2$. Application associates to the left, so the parentheses may be omitted in `(f x) y`. Because e_1 could be a data constructor, partial applications of data constructors are possible.

Example: The expression `(,) 1` denotes a function that, when applied to another expression `v`, constructs the tuple `(1,v)`.

Functions are curried In FREGE,all functions are considered *curried*; that is, every function f has type $t_1 \to t_2$ for certain types t_1 and t_2 and thus takes exactly one argument a of type t_1 and returns a value of type t_2. Data constructors are just functions, as far as expressions are concerned. Of course, t_2 can itself be a function type, say $t_3 \to t_4$. In that case the expression $f\ a$ denotes a value that *is* a function and can be applied to another value b of type t_3 immediately, yielding a value of t_4. And so forth, ad infinitum.

To be sure, for convenience, the language supports constructs that make it look like there were functios with more than one argument.

1. Function types can be written $t_1 \to t_2 \to t_3 \to t_4$ since the function type constructor $(\to)$ is right associative. One may view this as the type of a function that takes 3 arguments.
2. Function definitions can specify more than one argument.
3. Support on the lexical and syntactic level for "binary" operators provide special support for functions that conceptually have 2 arguments.
4. The left associativity of function application itself allows us to write `f a b c d` and nothing prevents us to think of this as "passing 4 arguments to the function f".
5. One can always write its own functions so that they take a tuple of values, which makes function application look very much like in conventional languages, e.g. `add(1,3)`[3]

To sum it up, nothing forces a FREGE programmer to think in terms of single argument functions only. Yet, it is important to keep in mind that behind the scenes this is all done with single argument functions.

3.5 Infix Expressions

Infix expressions are applications of infix operators (see section 2.5) written in natural notation. We also speak of *binary* expressions or operators.

[3]However, this will actually create a tuple on each function call.

binex:
    $binex_1$
$binex_{17}$:
    appex
$binex_i$:
    $binex_i$ $opleft_i$ $binex_{i+1}$
  | $binex_{i+1}$ $opright_i$ $binex_i$
  | $binex_{i+1}$ $opnone_i$ $binex_{i+1}$
  | $binex_{i+1}$
$opleft_i$:
    $lexop_i$ (left associative with precedence i)
$opright_i$:
    $lexop_i$ (right associative with precedence i)
$opnone_i$:
    $lexop_i$ (non-associative with precedence i)

The syntax shows that function application has higher precedence than any binary expression, followed by binary expressions of precedence 16 down to 1.

Translation:
e_1 `` `op` `` e_2 = (`` `op` ``) (e_1) (e_2)

3.6 Lambda Expression

A lambda expression defines an anonymous function.

lambda:
    \ *pattern* *lambda*
  | \ *pattern* `->` *expr*

The syntax of patterns and semantics of pattern matching are explained below, see section 3.11.

A lambda expression $\backslash p_1 \to \backslash p_2 \to \cdots \backslash p_k \to e$ can be abbreviated $\backslash p_1 \backslash p_2 \cdots \backslash p_k \to e$.

A lambda expression $\backslash p \to e$ has type $a \to b$ where a is the type of the pattern p and b the type of the expression e.

Difference to Haskell 98/2010: Each pattern must be introduced with a backslash. Separated this way, even complex patterns don't need parentheses around them, thus something like `\x:_ -> x` is valid.

Prec.	Assoc.	Operator	Purpose
16	right	`<~`	function composition
15	right	`**`	exponentiation
	none	`=~ !~ ?~ /~ ~ ~~ ~~~`	regular expression matching
14	left	`* / % ‘mod‘ ‘div‘ ‘rem‘`	multiplicative operators
13	left	`+ -`	additive operators
	right	`++`	string or list concatenation
12	left	`<<`	append to stream
		`‘bshl‘ ‘bshr‘`	bit shift left/right
11	left	`‘band‘`	bit-wise and
10	left	`‘bor‘ ‘bxor‘`	bit-wise or/exclusive or
9	none	`< <= > >=`	relational operators
8	none	`<=>`	compare two values
7	none	`== !=`	check for equality/unequality
		`=== !==`	check for identity
6	right	`&&`	logical and
5	right	`||`	logical or
4	none	`..`	the list of values from .. to
	right	`:`	list constructor
3	left	`>> >>=`	monadic bind operations
2	right	`@`	subpattern binding
1	right	`$`	application

Figure 3.3: Predefined Standard Operators

Translation: Application of a lambda expression is equivalent to a case expression (see section 3.9)
$(\backslash p \to b)\ e \quad = \quad$ `case` e `of` $p \to b$
where b and e are expressions and p is a pattern.

3.7 Conditional

cond:
`if` *expr* `then` *expr* `else` *expr*

A conditional expression `if` e_1 `then` e_2 `else` e_3 returns e_2 if e_1 evaluates to `true`, e_3 if e_1 is false and is undefined in all other cases.

The condition e_1 has type `Bool` and the possible results have the same type which is also the type of the whole expression.

3.8 Let Expressions

letex:
`let` { *decl* { `;` *decl* } } `in` *expr*

Let expressions have the general form `let` { $d_1; \cdots; d_n$ } `in` e, and introduce a nested, lexically-scoped, mutually-recursive list of declarations (let is often called letrec in other languages). The scope of the declarations is the expression e and the right hand side of the declarations.

Declarations appropriate for use in `let` expressions are described in section 4.4. The semantics of the nested declarations are described in subsection 4.4.3.

Pattern bindings are matched lazily, for example

```
let (a,b) = (3 `div` 0, 42) in e
```

will not cause an execution time error until `a` is evaluated.

The value computed by e is also the value of the whole let expression. It follows, that the type of the let expression is the type of e.

3.9 Case Expressions

casex:
`case` *expr* `of` { *alt* { `;` *alt* } }
alt:
pattern `->` *expr*
| *pattern guarded-exs*
| *alt* `where` { *decl* { `;` *decl* } }
guarded-exs:
guarded-ex { *guarded-ex* }
guarded-ex:
| *lc-qual* { `,` *lc-qual* } `=` *expr*

A case expression has the general form

$$\texttt{case}\ e\ \texttt{of}\ \{\ p_1\ match_1; \cdots; p_n\ match_n\ \}$$

where each $match_i$ is of the general form

$$\mid gd_{i_1}\ \texttt{=}\ e_{i_1} \mid \cdots \mid gd_{i_m}\ \texttt{=}\ e_{i_m}\ \texttt{where}\ \{\ decls_i\ \}$$

Each alternative $p_i\ match_i$ consists of a pattern p_i and its matches, $match_i$, which consists of pairs of guards gd_{i_j} and bodies e_{i_j}, as well as optional bindings ($decls_i$) that scope over all of the guards and expressions of the alternative. An alternative of the form

$$pat \texttt{ -> } expr \texttt{ where } \{ \ decls \ \}$$

is treated as shorthand for:

$$pat \mid \texttt{true = } expr \texttt{ where } \{ \ decls \ \}$$

A case expression must have at least one alternative and each alternative must have at least one body. Each body must have the same type, and the type of the whole expression is that type.

A guard is a list of expressions of type `Bool` or submatches of the form

$$pat \leftarrow expr$$

The expressions will be evaluated in the environment of the case expression extended by the bindings created during the matching of the alternative. The submatches may themselves create new bindings, these will be in scope for subsequent submatches of the same guard and for the expression associated with that guard.

A guard is matched successful if all expressions yield the result `true` and all submatches succeed. A failed match or an expression that evaluates to `false` causes the next guard to be tried if there is one, otherwise the alternative is not taken.

A case expression is evaluated by pattern matching the expression e against the individual alternatives. The matches are tried sequentially, from top to bottom. The first successful match causes evaluation of the expression that is associated with the guard that matched. If no match succeeds, the result is undefined and the program terminates. Pattern matching is described in section 3.11, with the formal semantics of case expressions in Figure 3.4.

3.10 Annotated Expression

expr:
- *topex* :: *sigma*
- *topex*

topex:
- *casex*
- *letex*
- *cond*
- *lambda*
- *binex*

An expression may be annotated with a type, the notation $e :: t$ means literally *expression* e *has type* t. If e has indeed this type, or a more general one, the value e will be computed at the given type, otherwise the compiler will flag an error.

Example: The expression on the right hand side of the function definition
foo a b c = a+b-c
has type $Num\ a => a$, i.e. it can have any type that is an instance of type class Num. However, if one writes
foo a b c = a+b-c :: Int
this restricts the type to Int.

3.11 Pattern Matching

Patterns appear in lambda abstractions, function definitions, pattern bindings, list comprehensions, do expressions, and case expressions. However, the first five of these ultimately translate into case expressions, so defining the semantics of pattern matching for case expressions is sufficient.

3.11.1 Patterns

Patterns have this syntax:

pattern:
  pattern `::` *sigma*
  | *atpattern*
atpattern:
  pvar `@` *atpattern*
  | *listpattern*
listpattern:
  matcherpattern `:` *listpattern*
  | *matcherpattern*
matcherpattern:
  pvar `~` *regexliteral*
  | *pconapp*
pconapp:
  pconapp strictpattern
  | *strictpattern*
strictpattern:
  `!` *pterm*
  | *pterm*
pterm:
  conid (constructor)
  | *conid* { [*patternfields*] }
  | *pvar* (variable, possibly strict)
  | *literal*
  | `( )` (unit constructor)

| `[ ]` (empty list)
| `[` *pattern* { `,` *pattern* } `]` (literal lists)
| `(` *pattern* `,` *pattern* { `,` *pattern* } `)` (tuples)
| `(` *pattern* `)`

pvar:

varid (variable)
| `_` (anonymous variable)

patternfields:

varid`=`*pattern* { `,` *varid*`=`*pattern* }

The pattern syntax is complicated at first sight only. A pattern is, syntactically, just a binary expression (see section 3.5) with the follwoing restrictions:

- the only infix operators allowed are `@`, `~`, `:` and constructors written in infix notation.
- only constructor applications are allowed
- no partial applications are allowed
- the only unary operator is `!`
- pattern tuples and lists are similar in appearance to their expression counterparts, except that they are made up of subpatterns instead of subexpressions. They are just syntactic convenient ways to write constructor applications.

In a constructor application pattern, the arity of the constructor must match the number of sub-patterns associated with it; one cannot match against a partially-applied constructor.

In contrast to expressions, constructors with field lists do not have to list all fields. It is even possible to have an empty pattern field list. A constructor with field list is transformed to an ordinary constructor application. The translation inserts anonymous variables for all missing fields.

All patterns must be linear - no variable may appear more than once. For example, this definition is illegal:

```
f (x,x) = x -- ILLEGAL; x used twice in pattern
```

Patterns of the form *pat*::*type* assert that the value matched with *pat* must have type *type*. Patterns of the form *var*@*pat* are called as-patterns, and allow one to use *var* as a name for the value being matched by *pat*. For example, the following function to "sort" a tuple

```
\(a,b) -> if b<a then (b,a) else (a,b)
```

constructs a new value that is equal to the original one in the else clause. To reuse the original value, one could write

```
\orig@(a,b) -> if b<a then (b,a) else orig
```

which is equivalent to

```
\orig -> case orig of (a,b) -> if b<a then (b,a) else orig
```

Patterns of the form _ are wildcards and are useful when some part of a pattern is not referenced on the right-hand-side. It is as if an identifier not used elsewhere were put in its place.

3.11.2 Informal Semantics of Pattern Matching

Patterns are matched against values. Attempting to match a pattern can have one of three results: it may *fail*, it may *succeed*, or it may *diverge*. When the match succeeds, each variable in the pattern will be bound to a value.

Pattern matching proceeds from left to right, and outside to inside, according to the following rules:

1. Matching a variable p against a value v always succeeds and binds p to v.
2. Matching the wildcard pattern _ against any value succeeds and no binding is done.
3. Matching a pattern of the form *pat*::*t* against a value v matches *pat* against v.
4. Matching a pattern of the form *pvar*@*pat* against a value v matches *pat* against v, and if this succeeds, binds *pvar* to v.
5. Matching a pattern $C\ p_1 \cdots p_k$ against a value v, where C is a constructor depends on the value:
   - If v is of the form $C\ v_1 \cdots v_k$ $(0 \leq k \leq 26)$, the sub-patterns p_i are matched left to right against the components v_i; if all matches succeed, the overall match succeeds; the first to fail or diverge causes the overall match to fail or diverge, respectively.
   - If v is of the form $C'\ v_1 \cdots v_n$, where C' is a different constructor to C, the match fails.
   - If the value is undefined, the match diverges.

   Constructors with arity 0 are just a special case without sub-matches.

   Constructor application patterns with field lists will have been transformed to ordinary constructor application patterns. Only the named field's subpattern will be matched against the corresponding subvalues. If the pattern field list is empty, just the constructor is checked.

6. Matching a boolean, numeric, string or character literal against a value succeeds if the value equals the value represented by the literal. It diverges if the value is undefined. Otherwise, the match fails.

7. Matching a regular expression literal (see section 2.7.3) against a value of type `String` diverges if the value is undefined. It succeeds, if the regular expression matches the string and fails otherwise.

8. If the pattern is of the form $m\ddot{}\#re\#$ it will be checked whether the regular expression matches the string as before and when this is the case, the variable m is bound to the matcher employed in the regular expression match. This, in turn, makes it possible to access the matched substring as well as the matched substrings corresponding to any parenthesized groups in the regular expression:

```
case "John Smith" of
    m~#(\w+)\s+(\w+)# -> case (m.group 1, m.group 2) of
             (Just a, Just b) -> b ++ ", " ++ a  -- "Smith, John"
```

9. A strict pattern $!p$ is matched against a value v by first evaluating v and then matching p against the result. Because evaluation of a value can diverge, matching a strict pattern can diverge in cases when matching a normal pattern would not. This can change the semantics of a program as the following example shows:

```
swap1 (a,b)   = (b,a)
swap2 (!a, b) = (b,a)
ex1 = swap1 (7 `div` 0, 42)   -- yields (42, 7 `div` 0)
ex2 = swap2 (7 `div` 0, 42)   -- diverges
```

The strict pattern is helpful in situations where the case that the value must not be evaluated is not important enough to justify the cost imposed by laziness:

Example:

```
– drop the first n elements from a list
drop n xs = case xs of
      [] → []
      _:ys → if n > 0 then drop (n-1) ys else xs
```

Here, the value n will not be evaluated if the list is empty, thus the application `drop (7 `div` 0) []` would compute the correct answer (the empty list). But for nonempty lists, n will be evaluated due to the comparison with 0. By making n a strict variable, the function will probably perform better in the vast majority of cases, however, the application above would then cause the program to abort.

Translation: Matches with regular expressions are translated like this:

$$\begin{aligned} &\texttt{case } v \texttt{ of } \#rx\# \to e &&= \texttt{case } v \texttt{ of } s \mid s \texttt{ ~ } \#rx\# \to e \\ &\texttt{case } v \texttt{ of } m\texttt{~}\#rx\# \to e &&= \texttt{case } v \texttt{ of} \\ &&& \quad s \mid \text{Just } m \leftarrow s =\texttt{~} \#rx\# \to e \end{aligned}$$

where rx stands for the regular expression and s is a fresh variable. (~) is a function defined in the Prelude that checks whether a string is matched by a regular expression. (=~) is a function defined in the prelude that performs a regular expression match and returns a result of type `Maybe Matcher`.

Pattern matches that force evaluation

Some pattern matches force evaluation of the value matched:

- all literal patterns and $m\texttt{~}\#rx\#$ need to fully evaluate the value.
- constructor application patterns for non-product types force evaluation of the value so that the constructor is known.
- strict patterns forces evaluation of the value in all cases, even if it has a product type. However, subcomponents of the value (if there are any) are not evaluated.

Example: In the example of the drop function above, the constructor patterns for the list argument force the list passed as argument to be evaluated so that the pattern match can see if it is an empty list or a list with at least one element.

Irrefutable patterns

A pattern is said to be irrefutable, if the match cannot possibly fail or diverge and if the matched value does not have to be evaluated during matching.

- Non-strict variable and wildcard patterns are irrefutable.
- $p\texttt{::}t$ is irrefutable, if p is
- $v\texttt{@}p$ is irrefutable if p is
- $C\ p_0\ \cdots\ p_k$ is irrefutable if all subpatterns are irrefutable and if C is the only constructor of the algebraic data type.
- All other patterns are refutable.

Irrefutable patterns are matched lazily. The value of the expression

Example: **let** swap (a,b) = (b,a) **in** swap undefined

would be ($fst\ undefined$, $snd\ undefined$), that is, binding of the local variables in *swap* does not extract the values from the argument, nor even evaluation of the argument.

Difference to HASKELL 98/2010: There exists no syntax to force a pattern to be irrefutable.

3.11.3 Formal Semantics of Case Expressions

The semantics of all pattern matching constructs other than `case` expression is defined by giving identities that relate those constrcuts to `case` expressions. The semantics of `case` expressions themselves are in turn given as a series of identities, in Figure 3.4. Implementations will behave so that these identities hold; this is not to say that they must use them directly, since that would generate rather inefficient code.

In Figure 3.4 e, e' and e_i are expressions; g and g_i are guards; q and q_i are qualifiers; p and p_i are patterns; v, x and x_i are variables; y is a fresh variable with a name that differs from all other names in the program; C and C' are algebraic datatype constructors (including tuple, list and record constructors); r is a regular expression literal and k is a boolean, character, string or numeric literal.

Rule (a) matches a general source-language `case` expression, regardless of whether it actually includes guards - if no guards are written, then `|true` is substituted. Also, the `where` clause may be missing, this will be equivalent to a `where` clause with 0 declarations. The rule simplifies a `case` expression with more than 1 alternatives into a nested `case` expression where each has at most 2 alternatives.

Rule (b) then supplements a `case` expression that has exactly one alternative with a catch-all alternative that evaluates to the undefined value. This guarantees that the `case` expression evaluates to the undefined value when all matches fail.

From here on, we have to deal with `case` expressions that have exactly two alternatives, where the second alternative has a simple wildcard pattern without guards and no `where` clause.

Rule (c) simplifies further by reducing multiple pattern guards to nested `case` expressions. The construction `case () of () ` $\rightarrow \cdots$ indicates that these `case` expressions do no pattern matching, but are just used to look at the guards. They are the only places where guards appear after the translation.

Our `case` expressions now have only patterns without guards or a ()-pattern with exactly one guard and no `where` clauses. A guard may be a list of qualifiers, they are further reduced to nested `case` expressions where guards consist of exactly one qualifier in rule (d).

Finally, rules (e) and (f) resolve the single-qualifier-guards. If the guard is a boolean expression, the `case` expression is transformed into a conditional, otherwise into a `case` expression that matches the pattern from the qualifier against the expression.

(a) `case` e `of` { $p_1\ match_1$; $\cdots$; $p_n\ match_n$ }
= `case` e `of` { $p_1\ match_1$; _ $\rightarrow$ `case` e `of` { $p_2\ match_2$; $\cdots$; $p_n\ match_n$ }}
where each $match_i$ has the form
$\mid g_{i,1} \rightarrow e_{i,1} \mid \cdots \mid g_{i,m} \rightarrow e_{i,m}$ `where` { $decls_i$ }

(b) `case` e `of` { $p_1\ match_1$ }
= `case` e `of` { $p_1\ match_1$; _ $\rightarrow$ `error "pattern match failure"` }

(c) `case` e `of` {
$p \mid g_1 \rightarrow e_1 \mid \cdots \mid g_n \rightarrow e_n$ `where` { $decls$ };
_ $\rightarrow e'$ }
= `case` e' `of` { $y \rightarrow$
`case` e `of` { $p \rightarrow$ `let` { $decls$ } `in`
`case` () `of` { () $\mid g_1 \rightarrow e_1$;
_ $\rightarrow \cdots$ `case` () `of` { () $\mid g_n \rightarrow e_n$;
_ $\rightarrow y$; } $\cdots$ }
_ $\rightarrow y$; }}

(d) `case` () `of` { () $\mid q_1, \cdots, q_n \rightarrow e$; _ $\rightarrow e'$ }
= `case` e' `of` { $y \rightarrow$
`case` () `of` { () $\mid q_1 \rightarrow \cdots$ `case` () `of` { () $\mid q_n \rightarrow e$;
_ $\rightarrow y$; } $\cdots$
_ $\rightarrow y$; }}

(e) `case` () `of` { () $\mid e_0 \rightarrow e$; _ $\rightarrow e'$ }
=`if` e_0 `then` e `else` e'

(f) `case` () `of` { () $\mid (e_0 \leftarrow p_0) \rightarrow e$; _ $\rightarrow e'$ }
=`case` e_0 `of` { $p_0 \rightarrow e$; _ $\rightarrow e'$ }

Figure 3.4: Identities for `case` expressions

Chapter 4

Declarations and Bindings

In this chapter, we describe the syntax and informal semantics of FREGE *declarations*.

program:
    `package` *packagename* [*inline*] `where` { *body* }
packagename:
    qconid
    | *qvarid* . *packagename*
    | *qconid* . *packagename*
inline:
    `inline` (*qname* {, *qname*})
body:
    topdecl
    | *topdecl* ; *body*
topdecl:
    fixity (see section 2.5)
    | *importdcl* (import declaration)
    | *typedcl* (type alias declaration)
    | *datadcl* (data type declaration)
    | *classdcl* (type class declaration)
    | *instdcl* (instance declaration)
    | *derivedcl* (derived instance declaration)
    | *decl*
decl:
    annotation (type annotation)
    | *binding* (function or pattern binding)
    | *nativefun* (native function)
decls:
    decl {; *decl*}

The *packagename* is a sequence of one or more identifiers where the last identifier starts with an uppercase letter. The package declaration opens a namespace with a name that equals the last component of the package name. Each top level item can be accessed with a qualified name that uses the namespace as qualifier, thus in

```
package my.fine.pack.with.a.long.name.X where
foo = 42
bar = let foo = 24 in foo + X.foo
```

`bar` evaluates to 66.

The *body* is a list of top level declarations. It is immaterial in which order the declarations are given.

The declarations in the syntactic category *topdecl* are only allowed at the top level of a FREGE package, whereas *decl*s can be used either at the top level or in the scope of a data type, class or instance.

Every top level declaration except fixity and import declarations can be preceded by one of the keywords `private`, `protected` or `public`, where `public` is the default and hence redundant. This determines the visibility of the declared item in importing packages. For detailed discussion of import and export see section 5.3.

For exposition, we divide the declarations into three groups: user defined data types, consisting of type and data declarations; type classes and overloading, consisting of class and instance declarations; and nested declarations, consisting of value bindings and type signatures.

Inline candidates

The `inline` clause in the package declaration causes the named items to become candidates for inlining. Inlineable functions may not:

- contain a let or where clause
- contain non-exhaustive patterns in lambda or case expressions
- be recursive
- reference private top level items

If these conditions are not met by a certain item, a warning will appear and the item will be ignored.

Note that all default class operations are automatically inline candidates.

If some module later imports the current package, the source code of the inlineable functions will be available and if the compiler options demand it, full applications of the said functions will be inlined.

For example, the ($) operator is an inline candidate of the Prelude and an application

```
f $ a
```

will be rewritten as

```
f a
```

if the compiler was instructed to do inlining.

4.1 Overview of Types and Classes

FREGE uses an polymorphic type system based on the traditional Hindley-Milner type system [4] to provide a static type semantics. The type system supports *type classes* or just *classes* that provide a structured way to introduce *overloaded* functions and values.

A `class` declaration (subsection 4.3.1) introduces a new *type class* and the overloaded operations that must be supported by any type that is an *instance* of that class. An `instance` declaration (subsection 4.3.2) declares that a type is an *instance* of a class and includes the definitions of the overloaded operations - called *class methods* - instantiated on the named type.

JAVA programmers are familiar with the concept of *interfaces* which serve a similar purpose like *type classes*, but there is a fundamental difference: A FREGE type class is not a type in its own right like a JAVA interface. Instances of type classes are types, instances of JAVA interfaces are objects.

4.1.1 Kinds

To ensure that they are valid, type expressions are classified into different *kinds*, which take one of two possible forms:

- The symbol $*$ represents the kind of all nullary type constructors.
- If k_1 and k_2 are kinds, then $k_1 \rightarrow k_2$ is the kind of types that take a type of kind k_1 and return a type of kind k_2.

Kind inference checks the validity of type expressions in a similar way that type inference checks the validity of value expressions. However, it is also allowed to state the kind of type variables explicitly, as long as the given kind is compatible with what kind inference would have inferred.

4.1.2 Syntax of Types

The syntax for FREGE type expressions is given below. Just as data values are built using data constructors, type values are built from *type constructors*. As with data constructors the names of type constructors start with uppercase letters.

The names of type constructors and data constructors will never be confused (by the compiler, that is) as the former only appear in expressions in the form of qualifiers and the latter never appear in types.

sigma:
- `forall` *tyvar* { *tyvar* } `.` *crho* (quantified type)
- *crho*

crho:
- *constraints* `=>` *rho* (constrained type)
- *rho*

rho:
- `(` *sigma* `)` → *rho* (higher ranked function type)
- *tyapp* → *rho* (function type)
- *tyapp*

constraints:
- *constraint*
- `(` *constraint* { `,` *constraint* } `)`

constraint:
- *classname tvapp* { *tvapp* }

type:
- *tyapp* → *type* (function type)
- *tyapp*

tyapp:
- *tyapp simpletype* (type application)
- *simpletype*

simpletype:
- *tyvar* (type variable)
- *tycon* (type constructor)
- `(` *type* `)`
- `(` *type* `,` *type* { `,` *type* } `)` (tuple types)
- `[` *type* `]` (list type)
- `(` *type* `|` *type* { `|` *type* } `)` (nested `Either`)

tyvar:
- *varid*
- (*varid*::*kind*)

tvapp:
- *tyvar*

| (*tvapp* { *tvapp* })

tycon:

qconid
| `[]` (list type constructor)
| `()` (unit type constructor)
| `(,{,})` (tuple type constructors)
| → (function type constructor (infix operator))

classname:

qconid

kind:

simplekind
| *simplekind* → *kind*

simplekind:

*
| (*kind*)

The main forms of type expressions are as follows:

1. Type variables written as identifiers beginning with a lowercase letter. A type variable can stand for a type of any kind.

2. Type constructors. Type constructors are written as identifiers beginning with an uppercase letter. The identifiers may be qualified. Type constructors denote either a user defined data type or a type alias.

   Special syntax is provided for the type constructors of the function type, list type, the trivial type and the tuple types.

   Type constructors can be constants like `Int` that denotes the integer type or they can be polymorphic like the list type.

3. Type application. Polymorphic type constructors (or type variables that stand for them) must be applied to type parameters to denote a complete type. For example, the list type constructor requires a type for the elements of the list.

4. A parenthesized type (t) is identical to the type t, but the parenthesis may be required for syntactical reasons.

Special syntax is provided to allow certain type expressions to be written in a more traditional style:

1. A *function type* has the form $t_1 \to t_2$. Function arrows associate to the right, thus $t_1 \to (t_2 \to t_3)$ can be written $t_1 \to t_2 \to t_3$.

   The function type constructor operator can be used in prefix form, that is $a \to b$ is equivalent to $(\to)\ a\ b$, as usual.

2. A *tuple type* has the form $(t_1, \cdots, t_k)$ where $k \geq 2$, which is equivalent to the type $(, \cdots,)\ t_1 \cdots t_k$ where there are $k-1$ commas between the parenthesis. It denotes the type of k-tuples with the first component of type t_1, the second component of type t_2 and so on.

3. A *list type* has the form `[`*t*`]`, which is equivalent to the type `[]` *t*. It denotes the type of lists with elements of the type *t*.

Although the tuple and list types have special syntax, they are not different from user-defined types with equivalent functionality.

Expressions, patterns and types have a consistent syntax. If t_i is the type of expression or pattern e_i, then the expressions $(\backslash\ e_1 \rightarrow e_2)$, `[`$e_1$`]` and (e_1, e_2) have the types $(t_1 \rightarrow t_2)$, `[`t_1`]` and (t_1, t_2), respectively.

In annotations, annotated expressions and patterns, the type variables in a type expression can be explicitly quantified (see rule *sigma*). In this case, it is a static error if a type variable that was not quantified appears in the type. Absent an explicit `forall`, all type variables are assumed to be universally quantified. For example, the type expression `a` $\rightarrow$ `a` denotes the type $\forall\ a.a \rightarrow a$. For clarity, however, we'll sometimes write quantification explicitly when discussing the types of FREGE programs.

More on explicit quantification is explained in the section dealing with higher rank polymorphism.

In type annotations as well as in class and instance declarations, the possible types represented by type variables (or applications of type variables) can be constrained to be members of a certain type class.

4.2 User-Defined Data Types

In this section, we describe algebraic and native data types (`data` declaration) and type synonyms (`type` declaration). These declarations may only appear at the top level of a module.

4.2.1 Algebraic Data type Declaration

datadcl:
`[ abstract ] data` *conid* `{` *tyvar* `}` `=` *constructors* `[ where {` *decls* `} ]`
constructors:
constructor `{|` *constructor* `}`
constructor:
`[ private ]` *conid* `{` *simpletype* `}` (traditional constructor)
| `[ private ]` *conid* `{` *labels* `::` *sigma* `{,` *labels* `::` *sigma* `}}` (constructor with fields)
labels:

$$\left[\ \texttt{!}\ \right] varid \left\{ \texttt{,} \left[\ \texttt{!}\ \right] varid \right\}$$

An algebraic data type declaration introduces a new type and constructors for making values of that type and has the form:

$$\texttt{data}\ T\ u_1 \cdots u_k\ \texttt{=}\ K_1\ t_{11} \cdots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \cdots t_{nk_n}$$

This declaration introduces a new type constructor T with constituent data constructors $K_1, \cdots, K_n$ whose types are given by

$$\forall\ u_1 \cdots u_k\ .\ t_{i1} \to \cdots \to t_{ik_i} \to T\ u_1 \cdots u_k$$

The type variables u_1 through u_k must be distinct and may appear in the t_{ij}; it is a static error for any other type variable to appear in the right hand side, except when said type variable is protected by a `forall` of a polymorphic record field.

It is possible to reference the newly introduced type constructor T on the right hand side of its own declaration, which allows to declare recursive types. For example, the following type is like the built-in list type:

```
data List a = EmptyList | Cons a (List a)
```

The declaration can be read "A list of elements of type a is either the empty list or an element of type a "consed" to another list of elements of type a".

There are some special cases for user defined data types:

product types are data types with exactly one constructor. The most prominent product types are tuples.

enumerations are data types where all constructors are constants, i.e.

```
data Color = Black | Red | Yellow
```

renamed types are product types where the constructor has exactly one field. Values of a renamed type will have the same representation as the constructor field at runtime, hence construction and deconstruction will incur no overhead. Yet at compile time renamed types are treated like any other algebraic data type. [1]

Visibility of constructors and abstract data types

The keyword `private` in front of a constructor declaration restricts the visibility of the constructor. Unqualified access is possible only from the declarations given in the `where` clause of the type. To access it in the rest of the package, the constructor name must be qualified with the type name. In other packages, the constructor is not visible at all.

The keyword `abstract` before a data declaration is equivalent to making all constructors private.

[1] This is equivalent to Haskell's `newtype`.

Strict or Eager Constructors

The default behavior for constructor applications is that the arguments are not evaluated. Evaluation will only take place when a value is deconstructed and a field value so obtained must be evaluated. Hence, operationally, a data structure does not store values but closures.

Sometimes a stricter mode is desired. A `!` before a field name requires that the constructor shall be strict in the corresponding argument. It is possible to have some, none, or all fields strict. Lazy and strict constructors can be mixed in one and the same data type.

Constructors with labeled fields

A different form for constructors is

$$K_j \ \{ \ f_{j1} :: t_{j1}, \cdots, f_{jk_j} :: t_{jk_j} \ \}$$

where the f_ji are field labels and the t_ji are the types.

The type of a labeled field can be *polymorphic*, that is, it may mention type variables in a `forall`-type that are not type parameters of the type constructor.

Example:

```
data F = F { fun ::  forall a .  [a] →  [a] }

bar F{fun} = (fun [0,1], fun [true, false])
```

As before, the type of the constructor is

$$\forall \ u_1 \cdots u_k \ . \ t_{j1} \to \cdots \to t_{jk_j} \to T \ u_1 \cdots u_k$$

Any number of constructors of an algebraic data type can employ the field list syntax. Constructors with and without field lists can be mixed. For convenience, when consecutive fields have the same type, they can be grouped so that the type must be written only once.

Translation:

data T $= \cdots \mid$ D $\{f_{a1}, f_{a2}, f_{a3} :: t_a, \ f_{b1} :: t_b \ \} \mid \cdots$

translates to

data T $= \cdots \mid$ D $\{f_{a1}, :: t_a, \ f_{a2}, :: t_a, \ f_{a3}, :: t_a, \ f_{b1} :: t_b \ \} \mid \cdots$

Every field in one and the same constructor must have a different label, but the same label can appear in a different constructor of the same `data` definition. All fields with the same label that appear in the same `data` definition must have the same type.

Difference to HASKELL 98/2010: The same field label with a possibly different type can appear in constructors of other types. This is because field names are only known locally in the type whose constructor introduced them. A name given to a field can thus in addition be used for a global variable.

For every field label appearing in an algebraic data type, the compiler defines automatically functions that extract a field from a value, update or change a field in a value [2] and check, if the given value has a certain field.

Translation:

data T = A t_a | B $\{f_{ba} :: t_{ba}, f_{bc} :: t_{bc}\}$ | C $\{f_{bc} :: t_{bc}, f_c :: t_c\}$

translates to

data T = A t_a | B t_b t_{bc} | C t_{bc} t_c **where**
    // For each of the fields f_b, f_{bc} and f_c, 4 functions will be generated,
    // we give here exemplary the ones for f_{bc}
    f_{bc} v = **case** v **of**
        { B _ f → f; C f _ → f }
    updf_{bc} v u = **case** v **of**
        { B a _ → B a u; C _ a → C u a }
    chgf_{bc} v g = **case** v **of**
        { B a f → B a (g f); C f a → C (g f) a }
    hasf_{bc} v = **case** v **of**
        { B _ _ → **true**; C _ _ → **true**; A _ → **false** }

The translation shows how the generated field functions look like. The names of the *upd...*, *chg...* and *has...* functions can not be mentioned directly by the programmer, as they are picked by the compiler in order to avoid name clashes. There exist suitable primary expressions to obtain and use the functions, though.

Type Namespaces

Each declaration of a data type T u_1 $\cdots$ u_k with $k \geq 0$ creates also a new namespace. This namespace will be populated with the declarations in the optional `where` clause of the `data` declarations. Each item v declared in the `where` clause of data type T u_1 $\cdots$ u_k

[2] This is, of course, nondestructive update, i.e. a new value that differs only in the value for the field is created.

can be accessed with the qualified name $T.v$ and, as explained in subsection 3.2.2, if e is of type $T\ u_1 \cdots u_k$, then the expression $e.v$ is equivalent to $(T.v\ e)$.

We call this feature *type directed name resolution* and it applies to algebraic datatypes as well as native ones. To obtain best results, the type checker collects not yet resolved names of a top level function in the form of type constraints, and tries to solve them all at once before the final type gets inferred. However, unlike with type class constraints, unsolved name resolutions are not allowed and thus never appear in the type of a top level function.

Difference to HASKELL 98/2010:

- Field names live in the namespace that is associated with the constructed type and are not global.
- Several forms of primary expressions deal with extraction, update and change of fields.
- Infix notation cannot be used in constructor declarations.
- No constraints can be given for constructors.
- There is no `deriving` clause. A derive declaration can be used instead.
- A data type with exactly one constructor that has exactly one field serves the same purpose as a `newtype` declaration.

4.2.2 Native Datatype Declaration

datadcl:

`data` *conid* { *tyvar* } = `mutable native` *nativeitem* [`where` { *decls* }]
| `data` *conid* { *tyvar* } = `pure native` *nativeitem* [`where` { *decls* }]
| `data` *conid* { *tyvar* } = `native` *nativeitem* [`where` { *decls* }]

nativename:

varid
| *conid*
| *varid* . *nativename*
| *conid* . *nativename*

A native data type declaration `data` $T\ u_1 \cdots u_k$ = `native` N introduces a new type constructor T that is associated with the JAVA type N. $T\ u_1 \cdots u_k$ is to FREGE an abstract data type. N may be a primitive java type like `int` or any java reference type. It is required to use fully qualified JAVA names for the latter.

The three syntactic forms help to distinguish between mutable and immutable (pure) data types, and data types whose values can be regarded as immutable under certain

circumstances and mutable in others. Truly immutable objects are rare in JAVA, so in most cases the `pure native` form will not be the appropriate one. A deeper discussion of this matters can be found in section 8.6.

For convenience, JAVA type names can be specified without quotes, as long as the components are valid FREGE identifiers. This excludes names starting with underscores or names that contain the $ sign and also JAVA generic type expressions like `java.util.List<Integer>`. Such JAVA types must be specified in the form of a string literal.

The generated code will contain the native type string and this will make the JAVA compiler complain if something is not in order.

Note that `void` is not a valid native type name since `void` is not a proper type.

For one and the same native type, more than one native data type declaration can exist. Becaue every reference type is its own supertype, the type checker will not regard those types as different (see also section 4.2.2).

Native data types can be used like any other type, for example, they can serve as building blocks for algebraic data types, or be parts of function types. However, since there is no constructor nor any knowledge about the implementation of the type, no pattern matching on values of native data types is possible. Basic operations on the type can be introduced with native function declarations.

FREGE borrows all fundamental data types like characters, numbers, booleans and strings from JAVA via native data type declarations.

Support for Java Class Hierarchy

The FREGE type checker takes into consideration the subclass relations between JAVA types. If T_1 and T_2 are type constructors of the same kind that are associated with JAVA class or interface types C_1 and C_2, and if C_2 is a super class of C_1 or an interface implemented by C_1, directly or through inheritance, then a type application $T_1\ a_1 \cdots a_n$ can appear wherever $T_2\ a_1 \cdots a_n$ is expected.

In addition, in the translation of an expression of the form $e.m$ where e has a type $Ta_1 \cdots a_n$ and T is associated with a native type C, not only the name space T is searched for m, but all name spaces of all known types that are associated with supertypes of C.

4.2.3 Type Synonyms

typdcl:

`type` *conid* $\left\{ \textit{tyvar} \right\}$ `=` *sigma*

A type synonym has the form

`type` $T\ u_1 \cdots u_k = t$

where $k \geq 0$ and the u_i are the type variables occuring in the type expression t. It is a static error if t contains a free type variable not listed on the left hand side among the u_i.

A type synonym T_a depends on another type synonym T_b if its right hand side mentions T_b or another type synonym T_c that in turn depends on T_b.

It is a static error if a type synonym T depends on itself. This means that cyclic definitions like

```
type T a = (a, X a)
type X a = T a
```

are forbidden.

A type expression $(T\ t_1\ \cdots\ t_k)$ where T is a type synonym is equivalent to the type expression t of the right hand side of the declaration of T where each variable u_i in t is replaced by the type expression t_i. Type synonyms cannot be partially applied in type signatures, it is a static error if during typechecking a type synonym declared with k type variables is found, that is applied to less than k type arguments. However, it is not required that the right hand side of a type synonym declaration is a type of kind $*$; expansion may produce a partially applied type constructor:

```
data T key value = T [(key, value)]
type X = T String

foo :: X Int
foo = [("x", 42)]
```

In the example above, expansion of type `X` produces a partially applied type constructor; to form a type one more argument is required.

Type synonyms are most often used to make type expressions more readable and programs more maintainable.

Liberalized Type Synonyms

Unlike in Haskell 2010, the right hand side of a type synonym may be a forall type or a constrained type. Consequently, one can write:

```
type Discard a = forall b. Show b => a -> b -> (a, String)

f  :: Discard c
-- :: forall b c. Show b => c -> b -> (c, String)
f x y = (x, show y)
```

```
g  :: Discard Int -> (Int,String)    -- A rank-2 type
-- :: (forall b . Show b => Int -> b -> (Int, String)) -> (Int, String)
g f = f 3 true
```

Such liberalized type synonyms may not be used where the right hand side - if written literally - would not be allowed. For example, given the definition above, the follwoing annotation is invalid, because it applies the synonym at a place where only ordinary types are allowed:

```
h :: Int -> Maybe (Discard Int)    -- illegal type argument (Discard Int)
```

4.3 Type Classes and Overloading

4.3.1 Class Declarations

classdcl:
`class` *conid* [*constraints* `=>`] *classvar* [`where` { *decls* }]
classvar:
varid

A *class declaration* introduces a new class and the operations (*class methods*) on it. A class declaration has the general form:

`class` C $(S_1\ u,\cdots,S_k\ u) => u$ `where` *decls*

This introduces a new class name C with class variabe u and so called superclasses S_i.

A class variable stands for potential types that will be instances of the type class. It is also possible that the class variable is higher kinded, i.e. that it is applied to other types in the class operation annotations. The kind of the class variable must be the same in all class operations and in all superclasses.

The S_i denote the superclasses of C. The superclass relation must not be cyclic. If S is a superclass of C, then each type that is an instance of C must also be an instance of S.

The class declaration introduces new *class methods* in its *decls* part. The class methods are not only visible in the scope of the class, but also at the top (package) level. Consequently, a class method can not have the same name as a top level definition or another class method.

A class method declaration is either

- a new operation introduced with a type annotation. The type given must mention the class variable and the class variable must not be constrained.

Optionally, a definition of the method can be given. The definition serves as a default implementation and will be used in instance definitions that give no instance specific implementation of that method.

A native function is annotation and definition in one declaration, so this too is allowed. This especially makes sense when one models JAVA inheritance hierarchies with type classes.

- a definition of one of the class methods of a superclass. There must be no annotation for such methods. The type of the class method will be derived from that of the superclass method by replacing the superclasses' class variable with the current class variable.

No other declarations are permitted in the *decls* part of a `class` declaration.

A `class` declaration without `where` clause is equivalent to one with an empty *decls* part. This can be useful for combining a collection of classes into a larger one that inherits all of the class methods in the original ones. Nevertheless, even if a type is an instance of all the superclasses, it is not *automatically* an instance of the new class. An instance declaration (with no `where` clause) is still required to make it one.

The same holds for types that happen to have implementations for all the operations of a class. That does not make them automatically instances of the class: there is no class membership without an instance declaration.

Example We give here a simplified version of some classes declared in the standard Prelude. See also section 6.2.

```
class Eq eq where
    (==) :: eq -> eq -> Bool
    (!=) :: eq -> eq -> Bool
    hashCode :: eq -> Int
    a != b  =  if a == b then false else true

class Ord Eq ord => ord where
    (<)  :: ord -> ord -> Bool
    ...

class Enum Ord enum => enum where
    ord   :: enum -> Int
    from :: Int -> enum
    a == b  =  (ord a) == (ord b)
    ...
```

The `Eq` class introduces two new overloaded operations and the function `hashCode`, with a default implementation for the `(!=)` method that makes use of the `(==)` operation.

The `Ord` class is a subclass of `Eq` and introduces more relational operations. The `Enum` class is declared as subclass of `Ord` and this makes it automatically also a subclass of `Eq`. Therefore, it is possible to give a default implementation for the `(==)` method.

4.3.2 Instance Declarations

instdcl:

`instance` *classname* [*constraints* `=>`] *type* [`where` { *decls* }]

An *instance declaration* introduces an instance of a class. Let

`class` C u `where` { *cbody* }

be a class declaration. The general form of the corresponding instance declaration is:

`instance` C $(X_1\ t_a,\cdots,X_n\ t_b)$ `=>` $T\ t_1\ \cdots\ t_k$ `where` { *decls* }

where $k, a, b \geq 0$, $a, b \leq k$ and $n \geq 0$.

If T is a type synonym, the expansion of the type expression must eventually lead to a type application of a type constructor[3]. Otherwise, T itself is already a type constructor. The type expression $T\ t_1\ \cdots\ t_k$ must have the same kind as the class variable of C.

An instance declaration may place arbitrary constraints $X_1\ \cdots\ X_n$ on all or some of the types represented by the type variables $t_1\ \cdots\ t_k$. For example

Example: instance Eq (Eq a) => [a] where ...

states that for a list type to be an instances of Eq, the list elements themselves must be of a type that is an instance of Eq. This is another way to say that in order to check list equality, one must be able to check equality on list elements also.

In this example, the instantiated class and the class mentioned in the constraint are the same. This is caused by the logic of relational operations, but is not a language requirement. Any other class could be mentioned as well.

There may be at most one instance per type class and type constructor. Because of this restriction, it is usually a good idea to design the instance as general as possible. The most general case is when the t_i are all distinct type variables.

Example:

```
class C this where ...
instance C [Int] where ...      – ok, but quite restrictive
instance C [a] where ...        – ERROR, [] is already an instance of C
instance C (a,a) where ...      – ok, but restricted
instance C (a,b,c) where ...    – most general instance
```

[3]Note that a function type like $a \to b$ is an application of type constructor $(\to\ )$

Difference to HASKELL 98/2010: The arguments to a type constructor need not be distinct type variables.
Type synonyms in instance declarations are allowed.

Instance methods Instance specific bindings of class methods and not yet implemented superclass methods are searched in the scope of the instantiated type and in the instance declarations *decls*. It is an error if both sources contain a binding for a class method. It is also an error if none of them contains such a binding unless there is a default implementation.

Annotations may be given; they will be checked to match the computed type of the instance method. This type is obtained by substituting the type expression describing the instantiated type for every occurrence of the class variable in the annotation of the class method. In addition, during type check, it will be checked that the definition of the instance method indeed has the computed type, as usual.

It is also possible to implement a class method with a native function.

If a class method has a default implementation but no instance specific implementation is found, the source code of the default implementation is taken to construct an instance specific implementation.

Each implementation of a class method that comes from the instance declarations or from a default implementation is linked back to the namespace of the instantiated type. This is so that the implementations of class operation *classop* specific for type T can be accessed under the qualified name $T.classop$.

Instantiating a Class Hierarchy with a single instance declaration. It is possible to declare an instance for a subclass and hence to make the instantiated type also an instance of all superclasses without giving explicit instance declarations for the superclasses, provided that

- the instance declaration contains implementations for the required class methods of the superclasses or
- the subclass gives default implementations for the superclasses' required methods.

For example, the type class `Enum` is a subclass of `Ord`, which is in turn a subclass of `Eq`. `Eq` requires definitions for `hashCode` and `==`. `Ord` requires a definition for `<=>`. `Enum` has a default implementation for `(Eq.hashCode)` and `(Ord.<=>)`. `Ord` has a default implementation for `(Eq.==)` in terms of `(Ord.<=>)`. Class `Eq` itself has a default implementation for `(Eq.!=)`. Thus it is possible to declare `Enum` instances without caring about the `Eq` and `Ord` methods. But because `Ord` has no implementation for `(Eq.hashCode)` one must provide one if one defines an `Ord` instance without there being also a `Eq` instance.

Note: If A is a superclass of both B and C, it is not valid to have just instances for B and C, but not A for some type T in a module, because the order instance declarations are processed is unpredictable. It is only guaranteed that instance declarations for any class are processed before instance declarations for its subclasses. Because T-specific implementations for As methods can only be given, explicitly or implicitly, in either the instance declaration for B or that for C, the validity of the program depends on the order of processing. Hence the correct thing to do is to instantiate A, B and C.

4.3.3 Derived Instances

For the Prelude classes `Eq`, `Ord`, `Enum`, `Bounded` and `Show` it is possible to *derive* instances automatically.

derivedcl:
`derive` *classname* [*constraints* `=>`] *type*

Derived instances provide convenient commonly-used operations for user-defined data types. For example, derived instances for data types in the class `Eq` define the operations `==` and `!=`, freeing the programmer from the need to define them. The precise details of how the derived instances are generated for each of these classes are given in chapter 9, including a specification of when such derived instances are possible.

Translation: A valid `derive` declaration is translated like this:

derive $C\ (X_1\ t_a, \cdots, X_n\ t_b)$ => $T\ t_1 \cdots t_k =$
instance $C\ (X_1\ t_a, \cdots, X_n\ t_b)$ => $T\ t_1 \cdots t_k$
where { $decls_C$ }

derive $C\ T\ t_1 \cdots t_k =$
instance $C\ (C\ t_1, \cdots, C\ t_k)$ => $T\ t_1 \cdots t_k$
where { $decls_C$ }

where $decls_C$ are compiler generated declarations whose concrete content depends on C and the structure of $T\ t_1 \cdots t_k$.

As with instance declarations, it is possible to specify constraints in derive declarations. The constraints so given must contain the set of constraints $(C\ t_1, \cdots, C\ t_k)$ where C is the derived class and the t_i are the type variables of the instantiated type. If, however, no constraints at all are given, the constraints needed will be silently supplied by the compiler.

4.3.4 Ambiguous Types

A problem inherent with overloading is the possibility of an *ambiguous type*. For example, using the *from* and *ord* functions from the *Enum* class introduced in the last section, then the following declaration

```
amb x = if ord (from 1) == 1 then x else x+x
```

causes the type inference algorithm to attribute the type

```
(Enum b,Num a) => a → a
```

to *amb*. If such type were allowed, it would never be possible to decide at which type to instantiate type variable b which is totally unrelated to type variable a which stands for the argument's type. Therefore, such types are considered ill-typed and programs containing them are rejected.

We say that an expression has an ambiguous type if, in its type

`forall` u_1`,` $\cdots$`,` u_k `.` cx `=>` t

there is a type variable u_j that occurs in cx but not in t.

Ambiguous types can only be circumvented by input from the user. One way is through the use of expression type-signatures as described in section 3.10. Our example could be rewritten as follows:

```
amb x = if ord (from 1 :: Bool) == 1 then x else x+x
```

4.4 Nested Declarations

The declarations described here can appear at the top level or in the scope of a class declaration, instance declaration or datatype declaration. With the exception of the native function declaration they can also appear in a `let` expression. In fact, this are the only declarations allowed in `let` expressions and `where` clauses that are part of expressions. (Such `where` clauses will be transformed to `let` expressions ulitmatively.)

decl:

annotation	(Type Annotation)
| *binding*	(Function or Pattern Binding)
| *nativefun*	(Native Function Declaration)

4.4.1 Type Annotations

A *type annotation* specifies the type of a variable.

annotation:
    annoitem { , *annoitem* } :: *sigma*
annoitem:
    varid | *symop* | *unop*

Translation: An annotation with more than one item is translated to k annotations, where k is the number of annotated items.

$$
\begin{array}{l}
a_1, \cdots, a_k :: s \\
\quad = \\
a_1 :: s \\
\quad \cdots \\
a_k :: s
\end{array}
$$

A type annotation has the form

$$v :: t$$

where v is the annotated item which may be a variable or an unary or binary operator symbol. (To simplify matters, we use the term *variable* in place of *annotated item* in the discussion that follows.)

Except for class methods, annotated variables v must have also a value binding, and the binding must appear in the same declaration list that contains the type signature; i.e. it is invalid to give a type signature for a variable bound in an outer scope. Moreover, it is invalid to give more than one type signature for one variable, even if the signatures are identical.

As mentioned in subsection 4.1.2, every type variable appearing in a signature is universally quantified over that signature, and hence the scope of a type variable is limited to the type expression that contains it. For example, in the following declarations

```
f :: a -> a
f x = x :: a -- invalid
```

the `a`'s in the two type expressions are quite distinct. Indeed, these declarations contain a static error, since `x` does not have type $\forall\, a.a$ but is dependent on the function type.

If a given program includes a signature for a variable f , then each use of f is treated as having the declared type. It is a static error if the same type cannot also be inferred for the defining occurrence of f.

If a variable f is defined without providing a corresponding type signature declaration, then each use of f outside its own declaration group (see section 4.4.3) is treated as having the corresponding inferred, or *principal* type. However, to ensure that type inference is still possible, the defining occurrence, and all uses of f within its declaration group must have the same monomorphic type (from which the principal type is obtained by generalization, as described in section 4.4.3).

For example, if we define

```
sqr x = x*x
```

then the principal type is `sqr` :: **forall** a.`Num` a `=>` $a \rightarrow a$, which allows applications such as `sqr 5` or `sqr 0.1`. It is also valid to declare a more specific type, such as

```
sqr :: Int → Int
```

but now appications such as `sqr 0.1` are invalid. Type signatures such as

```
sqr :: (Num a, Num b) => a → b
sqr :: a → a
```

are invalid, as they are more general than what is warranted by the definition.

However, there are certain cases where the type checker infers a type that is not the most general one possible for the definition given. In such cases, an annotation can be used to specify a type more general than the one that would be inferred. Consider this rather pathological example:

```
data T a = K (T Int) (T a)
f :: T a -> a
f (K x y) = if f x == 1 then f y else undefined
```

If we remove the annotation, the type of `f` will be inferred as `T Int -> Int` due to the first recursive call for which the argument to `f` is `T Int`.

To sum it up, there are the following possible uses of type annotations:

1. Declaration of a new class method, as described in subsection 4.3.1.
2. Declaration of a more restricted type than the principal type.
3. Declaration of a more general type than the inferred type. Please observe that, even if the type inference algorithm is not able to infer the most general type from a definition, it is still able to check whether the type signatur supplied is valid. Therefore, type annotations cannot be used to lie about the type of a variable.
4. Declaration of a type that is identical to the type that would have been inferred. This may be useful for documentation purposes.
5. Declaration of a polymorphic type for let bound functions.
6. Declaration of a higher ranked type.

4.4.2 Function and Pattern Bindings

binding:
    lhs rhs
lhs:

funlhs | *pattern*

funlhs:

varid pterm { *pterm* }

| *pconapp lexop pconapp*

rhs:

= *expr* [`where` { *decls* }]

| *guarded-exs* [`where` { *decls* }] (see section 3.9 for syntax of *guarded-exs*)

We distinguish two cases of value bindings: If the left hand side is neither a constructor application nor an application of the unary operator `!` or the pattern name binding operator `@`[4], and if it can be interpreted as a variable applied to one or more patterns, or as a binary operator except `@` applied to two patterns in infix notation, we call it a function binding, otherwise it is a pattern binding. Thus, a left hand side like $m\tilde{}\#re\#$, though making a valid pattern, will be treated as function binding (i.e. definition of the operator `~`). On the other hand, x:xs, `[`a, b`]` and (a, b) are pattern bindings, since all three just employ special syntax for constructor applications, `!`x is a (strict) pattern binding and qs`@`q:_ is a pattern binding for a list qs with head q and an unnamed tail. Finally, a simple variable or an operator enclosed in parentheses is a pattern binding for that name.

Function bindings

A function binding binds a variable (or operator) to a function value. The general form of a function binding for variable x is:

$$
\begin{aligned}
& x\ p_{11} \cdots p_{1k}\ match_1 \\
& \cdots \\
& x\ p_{n1} \cdots p_{nk}\ match_n
\end{aligned}
$$

where $k \geq 1$, $n \geq 1$ and each p_{ij} is a pattern and the matches $match_i$ are just like the matches in case expressions.

All clauses defining a function must be contiguous, and the number of patterns in each clause must be the same. The set of patterns corresponding to each match must be linear, that is, no variable is allowed to appear more than once in the set.

[4]These exceptions in an already complicated rule can only be justified by the fact that one seldom wants to redefine the `!` or `@` functions.

Translation: The general binding form for functions is semantically equivalent to the equation

$$x = \backslash x_1 \cdots \backslash x_k \rightarrow \textbf{case } (x_1, \cdots, x_k) \textbf{ of}$$
$$(p_{11}, \cdots, p_{1k})\ match_1$$
$$\cdots$$
$$(p_{n1}, \cdots, p_{nk})\ match_n$$

where the x_i are new identifiers.

Note that several clauses defining a function count as a single declaration. While definitions of different functions may appear in any order without changing the meaning of the program, this is not true for the clauses of a function definition. On the contrary, because of the translation given above and the semantics of `case` expressions, their order is quite important and cannot usually be changed without changing also the meaning of the program.

Pattern bindings

A *pattern binding* binds all variables contained in the pattern on the left hand side to values. It is a static error if the pattern does not contain any variables. The pattern is matched against the expression on the right hand side only when one of the bound variables needs to be evaluated. In any case, the expression on the right hand side will be evaluated at most once, and for each bound variable the match is performed at most once.

This default lazy semantics can be overridden by using strict patterns (see page 43). A strict variable will be evaluated as soon as it is bound to a value. This may cause other variables on which the strict variable depends to be evaluated, too.

A *simple* pattern binding has the form $v = e$, where v is just a variable. No actual pattern matching is needed in this case. The evaluation of v will cause evaluation of e and the resulting value is the value of v. If v is not evaluated, e will also not be evaluated.

The *general* form of a pattern binding is $p\ match$, where *match* is the same structure as for function bindings above, which in turn is the one used in case expressions; in other words, a pattern binding is:

$$p \mid g_1 = e_1 \mid g_2 = e_2 \mid \cdots \mid g_m = e_m$$
$$\textbf{where } \{\ decls\ \}$$

Translation: The general pattern binding above is semantically equivalent to the following:

x = **let** { $decls$ } **in**
  case () **of** { () | $g_1 = e_1$ | $g_2 = e_2$ | $\cdots$ | $g_m = e_m$ }
v_1 = **case** x **of** { $p \to v_1$ }
$\cdots$
v_k = **case** x **of** { $p \to v_k$ }

where $k \geq 1$ and the v_i are the variables occurring in the pattern p and x is a variable not used elsewhere.

4.4.3 Static Semantics of Function and Pattern Bindings

The static semantics of the function and pattern bindings of a `let` expression or `where` clause are discussed in this section.

Dependency Analysis

In general the static semantics are given by the normal Hindley-Milner inference rules. A *dependency analysis* transformation is first performed to simplify further processing. Two variables bound by value declarations are in the same *declaration group* if their bindings are mutually recursive (perhaps via some other declarations that are also part of the group).

Application of the following rules causes each `let` or `where` construct (including the (implicit) `where` defining the top level bindings in a module) to bind only the variables of a single declaration group, thus capturing the required dependency analysis:

1. The order of function and pattern bindings in `where`/`let` constructs is irrelevant.
2. **let** { d_1; d_2 } **in** e transforms to **let** { d_1 } **in let** { d_2 } **in** e when no identifier bound in d_2 appears free in d_1.

Generalization

The type system assigns types to a `let`-expression in two stages. First, the right-hand sides of the declarations are typed, giving types with no universal quantification. Second, if and only if the local declaration had a type annotation, the annotation is checked and a generalized type is attributed to the binding. Finally, the body of the `let`-expression is typed.

Difference to Haskell 98/2010: In Frege, un-annotated let bound declarations are not generalized. If one wants polymorphic local functions, one needs to provide a type signature.

For example, consider the declaration

```
f = let g y = (y,y) in (g true, g 0)
```

that would be valid in other languages with Hindley-Milner type system. The type of `g`'s right-hand side is $a \to (a, a)$. A generalization step would attribute to `g` the polymorphic type $\forall a.a \to (a, a)$, after which the typing of the body part can proceed. This makes `g` independent of any other types and allows its usage in the body at *different* types.

In FREGE, however, this generalization step has been deliberatly ommitted. The rationale for this is:

- As Simon Peyton Jones argues in [5]:

  > ... generalisation for local let bindings is seldom used; that is; if they are never generalised, few programs fail to typecheck ... We base this claim on compiling hundreds of public domain Haskell packages, containing hundreds of thousands lines of code. Furthermore, those programs that do fail are readily fixed by adding a type signature.

- Generalisation affects type constraints that arise from the use of type class operations. This means that type class constraints must be passed to local functions at runtime, and the associated operations can only be performed indirectly. Consider the following example:

  ```
  f = (dbl 7, dbl 13) where dbl x = x+x
  ```

  Here, the `dbl` function is only used at type `Int`, and without generalisation we can simply generate JAVA code like:

  ```
  public int dbl(final int x) { return x+x; }
  ```

  But with generalisation, `dbl` gets the type: $\forall a.$ Num a => $a \to a$ and we need something like

  ```
  public Object dbl(CNum ctx, Object x) {
  return ctx.plus().apply(x).apply(x).result(); }
  ```

  which results in at least 4 object creations.

- Type directed name resolution also benefits, as the type of some data may be exposed in some let bound function or list comprehension (which is desugared to a series of let definitions), but this knowledge will not be usable in other let bound functions when their type is generalized.

Sometimes it is not possible to generalize over all the type variables used in the type of the definition. For example, consider the declaration

```
f x = let g z y = ([x,z],y) in ...
```

In an environment where `x` has type a, the type of `g`'s defining expression is $a \to b \to ([a], b)$. In this case, only b could be universally quantified because a occurs in the type environment. We say that the type of `g` is *monomorphic in the type variable a.*

The effect of such monomorphism is that the first argument of all applications of `g` must be of a single type. For example, it would be valid for the "..." to be

```
(g true 0, g false 1)
```

(which would, incidentally, force `x` to have type `Bool`) but invalid for it to be

```
(g true 0, g 'c' 1)
```

It is worth noting that the explicit type signatures provided by FREGE are not powerful enough to express types that include monomorphic type variables. For example, we cannot write

```
f x = let
        g :: a -> b -> ([a],b)
        g y z = ([x,y], z)
    in ...
```

because that would claim that `g` was polymorphic in both `a` and `b`. In this program, `g` can only be given a type signature if its first argument is restricted to a type not involving type variables; for example

```
g :: Int -> b -> ([Int],b)
```

This signature would also cause the variable `x` to have type `Int`.

Higher Rank Polymorphism

In the Hindley-Milner type system, the types of lambda bound variables are always monomorphic. This restriction keeps type inference decidable, but excludes *higher rank polymorphism*, that is, the ability to write functions that take polymorphic functions as arguments. For an in depth discussion of these matters see [4]. The conservative extension of the type system proposed in that paper is implemented in the FREGE type system.

To exemplify the problem, consider the following program:

```
foo :: ([Bool], [Char])
foo = let
        f x = (x [true,false], x ['a','b','c'])
    in f reverse
```

In the body of `f`, the function `x` is applied both to a list of booleans and a list of characters - but that should be fine because the function passed to `f`, namely `reverse` (a library function with type $\forall a.[a] \to [a]$) works equally well on lists of any type. Nevertheless, the expression is rejected as it stands. With the restriction on lambda bound arguments, the type checker can assign to `x` the type `[Bool] -> [Bool]` or `[Char] -> [Char]` but not $\forall a.[a] \to [a]$.

The FREGE type checker can overcome this restriction of the Hindley-Milner type system with the help of user supplied type annotations. In our case, there are two opportunities to suggest the correct type for the argument x:

1. One could annotate the pattern x with the type

   `forall` a`.[`a`] ->  [`a`]`

2. One could annotate f itself with

   `(forall` a`.[`a`] ->  [`a`]) -> ([Bool], [Char])`

If one chooses the former, the example above looks then like

```
foo :: ([Bool], [Char])
foo = let
        -- f will take a polymorphic function as argument
        f (x::[a]->[a]) = (x [true,false], x ['a','b','c'])
    in f reverse
```

Note that we can save the `forall`, because the type variable a scopes over the whole type of x. The type checker would now be able to infer the following type for f:

`f :: (forall` a`.[`a`] -> [`a`]) -> ([Bool], [Char])`

Note that this is the same type one could have annotated f with and that in this case the `forall` is crucial, as it restricts the scope of type variable a to the type in parentheses. Without it, one would get

`f :: ([`a`] -> [`a`]) -> ([Bool], [Char])`

which is identical with

`f :: forall` a `. ([`a`] -> [`a`]) -> ([Bool], [Char])`

and this would mean something like: "f is a function that takes as first argument a function of *some unknown, albeit fixed* list type, ..." and of course neither a list of Bool nor a list of Char could be passed to this function. Whereas the former type signature with the correct placement of `forall` says: "f is a function that takes as first argument a function that can handle *any* list type, ...".

Difference to HASKELL 98/2010: Haskell 98 does not allow higher rank polymporphism, while extensions like GHC do.

4.4.4 Native Declaration

nativefun:
[`pure`] `native` *annoitem* [*javaitem*] `::` *sigma* [*throws*]
javaitem:
nativename | *unop* | *symop* | *stringliteral*
throws:
`throws` *type*{ `,` *type*}

The basic form of the *native declaration*

native *v j* **::** *t*

introduces a new variable *v* with type *t*, that will behave like any other FREGE variable of that type but is implemented in JAVA. *j* is a string value that contains information regarding *v*'s JAVA implementation. For convenience, *j* can be written without quotes as long as the names or operators specified would also be valid in FREGE. Often, it is the case that *v* and *j* are the same, as in

```
data String = pure native java.lang.String
pure native length length :: String -> Int
```

In such cases, *j* can simply be omitted.

The declarations above introduce the type `String`, which is based on the JAVA class `java.lang.String` and a function `length` that takes a value of type `String` and returns an `Int` [5]. This function is implemented with an instance method `length`. Because a JAVA instance method needs a receiver, which is by definition the first argument, and the first argument is a `String`, which is implemented by `java.lang.String`, we can assume that an application

```
length "foo"
```

will be compiled to something like

```
"foo".length()
```

The FREGE compiler cannot check the validity of the native declaration, only very basic sanity checks are currently possible. Errors not detectable (for example incorrect type information, spelling errors in identifiers, etc.) will normally cause failures during compilation of the generated JAVA code. The presence of invalid native declarations should be the only reason for rejection of the generated code by the JAVA compiler[6]; thus whenever `javac` complains, one should first check the native declarations.

[5] The declarations shown here for demonstration are actually part of the standard library and hence available in every frege program.

[6] Another reason would be bugs in the JAVA or FREGE compilers

During code generation, expressions that contain native values are mapped to certain JAVA constructs such as

- field access expressions (see [6, section 15.11])
- method invocation expressions (see [6, section 15.12]) for both static and non-static methods
- class instance creation expressions (see [6, section 15.9])
- unary expressions (see [6, section 15.15])
- cast expressions (see [6, section 15.16])
- binary expressions (see [6, section 15.17 to 15.24])

in a transparent and well defined way. This is explained in detail in chapter 8.

Overloaded native methods

The JAVA language allows *overloading* of methods [6, section 8.4.9]. Thus, in JAVA one can have different methods with the same name. To facilitate easy (perhaps semi-automatic) adoption of JAVA APIs in FREGE, native functions can also be overloaded.

nativefun:
[`pure`] `native` *annoitem* [*javaitem*] `::` *sigma* [*throws*] { `|` *sigma* [*throws*] }

To do this, one simply specifies the different types separated by a vertical bar. The typechecker will check at every call site if one of the given types fits in, and it will complain if the overload cannot be resolved unanimously.

Example:

```
pure native foo x.y.Z.foo ::  Int -> Int | Int -> Int -> Int
bad = foo 3
good = foo 3 + 42
```

In the example above, the expression for `bad` is ambiguous: `bad` could have type `Int` or `Int->Int`, therefore this would be rejected. By annotating `bad` with one of the possible types, the binding could be made valid.

In the expression given for `good`, however, the type of the application `foo 3` is certainly `Int`, as it appears as the operand of an integer addition. Hence, this overloaded use of `foo` is resolved to be at type `Int->Int`.

Chapter 5

Packages

FREGE code is organized in packages. One source file contains exactly one package. Compilation of a package creates an intermediary JAVA source file that contains the definition of a public class whose fully qualified name equals the FREGE package name. Finally, the JAVA compiler is invoked to compile the intermediate file, which results in at least one class file.

For example:

```
package pack.A where  -- frege code
-- declarations and bindings
```

compiles to

```
package pack;        // java code

public class A {
    // compiled declarations and bindings
}
```

Each source file must start with a package declaration. The syntax of package names is given in chapter 4.

All components of the package name except the last one should be in lowercase. This is because the JAVA package name is derived from these components and JAVA has the convention that the package name should be all lowercase.

Neither file name of the FREGE source code file nor the path where that file is stored have to reflect or match the package name. However, the intermediate JAVA source file and the final class files are subject the the customs of JAVA. That means that the files reside in a path that resembles the package name and the file name is the class name plus a suffix.

5.1 Execution of Frege Packages

The generated JAVA code for a FREGE package contains static fields, methods and inner classes that correspond to FREGE values, functions and types defined in that package. If the source code contained a top level binding for the name *main*, then the compiler creates a wrapper method with the signature

```
static void main(java.lang.String[] args)
```

which converts the argument array to a list, evaluates the *main* function from the FREGE code and runs the resulting IO action.

This makes it possible to run any class file that is the result of compiling a FREGE package that contained a definition of *main* with the `java` command. Note that, because the generated JAVA code will contain references to FREGE runtime code, the standard libraries and possibly other imported FREGE packages, execution of a FREGE package requires a class path that allows the JVM to find the required classes.

Special handling of the main function

The only allowed types for *main* are

```
main :: [String] -> IO X
main :: IO X
```

where X is some arbitrary type. To ensure this, the compiler will provide one of the following type annotations prior to type checking:

```
main :: [String] -> IO ()
main :: IO ()
```

The decision which annotation is supplied depends on whether `main` *looks like* a function or not. For example, the following will result in a type error:

```
main = return . const ()
```

because it doesn't look like a function, when in fact it is one. The type checker will expect an `IO ()` value, and this will not unify with the inferred type.

To avoid such errors, one of the following is suggested:

- Do not write a `main` function in point-free style.
- Give an explicit annotation for the `main` function.

A type annotation is also required whenever the value returned from the `IO` monad is not the unit type `()`.

Interpretation of the value returned from `main`

Whenever the ultimate return type of the `main` function is one of

```
IO Int
IO Bool
```

the wrapper method will retrieve the value and will pass it to `java.lang.System.exit` when it is an `Int`. If it is `Bool` instead, the wrapper method passes 0 for `true` and 1 for `false` to the exit method.

Here is a program that implements the Unix utiliy `false` in FREGE:

```
main :: IO Bool
main = return false
```

If the return type is something else, no explicit return is performed.

Example: This example assumes that the FREGE runtime and compiler is contained in a file `frege.jar` that resides in the `lib/` subdirectory of the users home directory and that the PATH environment variable is set so that the `java` and `javac` commands can be accessed.

```
$ cat Hello.fr
package test.World where
main (g:_) = println ("Hello " ++ g ++ "!")
main args = println "Hello World!"
$ java -jar ~/lib/frege.jar Hello.fr # compile the package
runnning: javac -cp ~/lib/frege.jar:. ./test/World.java
$ java -cp ~/lib/frege.jar:. test.World # run it by passing package name
Hello World!
$ java -cp ~/lib/frege.jar:. test.World again # pass an argument
Hello again!
$
```

Difference to HASKELL 98/2010: The *main* value is either itself an IO action, or a function that takes a list of strings and returns an IO action. There is no library function to access the command line arguments. A *main* function can appear in any package. In fact, a program can consist of multiple packages each defining its own *main* function. To run a specific *main* one passes the package name to an appropriate JVM launcher. When the *main* function has an ultimate return type of `IO Int` or `IO Bool`, the returned value will determine the exit status when run through the `static void main()` method.

5.2 Packages and Namespaces

The package declaration creates the namespace wherein all top level declarations of the package live. The namespace has itself a name, it is the last component of the package name. Namespace names must start with an uppercase letter. They will never conflict with names of other kinds like type names or value constructors. However, having a namespace and a type with the same name N can lead to confusions, as explained earlier in the section that deals with qualified names.

All items defined at the top level of the current namespace can be accessed with unqualified names or with a name qualified by the namespace.

5.3 Importing Packages

A package may import any other previously compiled package with an import declaration. A compiled package contains information about all non private type aliases, type classes, instances, data types, functions and values defined in it in a form that allows for quick reconstruction of the packages symbol table. Hence, during compilation of the importing package the compiler will know the names, kinds and types of the imported items and can generate appropriate code to access them.

importdcl:
import *packagename* [[`as`]*namespace*] [`public`] *importlist*
importlist:
[`hiding`] `(` [*importspec*{ `,` *importspec*}] `)`
importspec:
[`public`] *importitem* [*alias*]
importitem:
varid | *conid*[*members*] | *symop* | *conid* . *varid* | *conid* . *conid* | *conid* . *symop*
alias:
varid | *conid* | *symop* (must match the aliased item)
members:
`(` [*memspec*{ `,` *memspec*}] `)`
memspec:
[`public`] *member* [*alias*]
member:
varid | *symop* | *conid*
namespace:
conid

Difference to Haskell 98/2010: There is no special syntax for so-called "qualified" imports. In Frege, all imports are basically qualified, as is explained in the following pages.

Import declarations are processed in the order they occur in the program text. However, their placement relative to other declarations is immaterial. Even so, it is considered good style to write all import declarations somewhere near the top of the program.

An import encloses the following steps:

- The class file corresponding to the imported package is loaded. The class file must exist in the current class path or in a path given with compiler command line flags.
- A new namespace is created for the import. If no name for the name space was specified, the last component of the given package name is used.
- Items defined in the imported package are extracted from the class file data and their definition is stored in the new name space.
- The import list will be processed to give certain imported items names in the current namespace, so that they may be accessed with unqualified names. This process of linking is laid out in the following sections.

It is not possible to import a JAVA class file that does not result from the compilation of a FREGE source file.

Different packages must be imported into different name spaces.

It is possible to make use of types or classes that have no valid name in the current package. This is the case, for instance, when a package A declares a data type, package B imports this type and defines functions that use this type and package C imports and uses that function from B, but does not import A. Apart from the fact that some type signatures can't be written in C this is no problem.

5.3.1 Import lists

All non private definitions (i.e. functions, values, types, constructors, type classes and operators) of the imported package will be accessible in the importing package through identifiers qualified with the associated name space name.

Instances play a special role insofar as they cannot be named and hence cannot appear in import lists. Therefore they will be automatically known, no matter how the import list looks like.

Often one wishes to name some imported items without qualifiers. This can be accomplished by listing those items in the import list:

Example: import math.Rational as R (gcd, Rat)

This causes the function `Rational.gcd` to be known under the name `gcd` in the current package. Also, if there is a constructor or type `Rat` in `math.Rational`, it will be known in the current package. Both items can still be named as `R.gcd` or `R.Rat`.

A value constructor may have the same name as a type constructor, type alias or class. It follows that mentioning an unqualified constructor name in an import list can be ambiguous. This is resolved in favor of type names, i.e. if a type, alias or class with that name exists in the imported package, it will be linked to the current namespace. To import a value constructor with an ambiguous name, it must be qualified with its type name in the import list.

Example:
```
package Imported where
class C where  ...
data Notes = C | D | E | F | G | H | A | B

package Importing where
import Imported (C) – class C
import Imported (Notes) – data type Notes and its public constructors
import Imported (Notes.C) – just the data constructor
```

Controlling import of data constructors and members A class name in an import list can be followed by a list of class operations that should be available in the namespace of the importing package. If the list is empty, only the class name itself is imported, access to the class operations are possible through names that are qualified with the class name. If the list is missing, all public class operations are available.

Likewise, a type name in an import list can be followed by a list of names declared in that type that should be linked to the namespace of the importing package in addition to the type name itself. If the list is empty, no additional names are linked. If the list is missing, all public constructor names are linked.

This feature can be used to prevent unwanted linking of constructor names to avoid name clashes, like in the following example:

Example:
```
package A where data T1 = T ....

package B where data T2 = T ...

package C where
import A( T1() )
import B( T2 ) – hence T means T2.T
```

Like with items in the import list, data members can be renamed or re-exported.

An alternative solution for the constructor name clash problem would be:

Example:
```
package A where data T1 = T ....

package B where data T2 = T ...

package C where
import A( T1( T TA) ) – TA means A.T1.T
import B( T2( T TB) ) – TB means B.T2.T
```

Empty import list If the import list is empty, no named items are linked to the current namespace. Nevertheless, as before, all items of the imported package can be accessed with qualified names.

5.3.2 Importing all public items

In addition, it is possible to link all public definitions of a package into the namespace of the importing package by not giving an import list at all.

This will not make available the items declared `protected`. Protected items are linked only when explicitly mentioned in an import list.

Example:
```
import math.Rational
```

5.3.3 Renaming on import

An item can be put in the current namespace with a different name. However, *conid*s can only be aliased to *conid*s while *varid*s and *symop*s can not be aliased to *conid*s.

This is useful to avoid name clashes or to adapt naming conventions[1].

Multiple aliases are possible for a single item.

Example:
```
import math.Rational (gcd greatestCommonDivisor, gcd ggT)
```

It is important to understand that the qualified name of an imported item remains unaffected by renaming. All names created by imports in the current namespace are merely symbolic links to the original item. Hence, after the import of the example, the names `Rational.gcd`, `greatestCommonDivisor` and `ggT` reference the same item.

[1] Be aware, though, that error messages may report the original qualified name.

5.3.4 Re-exporting imported items

It is possible to have a packages re-export all or some of its imported items along with the items declared in the package itself, if any[2], so that importing packages can import many related items from possibly many different packages with one import.

A library designer can use this feature, especially in combination with item renaming, to hide implementation details of his library, such as package names, where and under what name exactly an item is defined etc., by providing a single package defining the "official" interface of the library. If the library is redesigned later, the package defining the interface can be adapted accordingly so that backwards source compatibility can be maintained[3].

To re-export a single item, one writes `public` before its name in the import list. Note that it is the alias name that will be exported for renamed items. To re-export all items in the import list or just all public items when there is no import list, one writes `public` after the package name.

Example:
```
package x.y.API where
import x.y.z.Sub1 (public creat create, public fold reduce)
import x.y.z.Sub2 public hiding (deprecated1, deprecated2, ...)
import x.y.contributed.byXY.Utilities public

package XYClient where
import x.y.API
– makes available create, reduce, non deprecated items from Sub2
– and public utiliities from contributor XY, but the client need not
– know all those details.
```

5.3.5 Importing all but some items

If the import list is preceded by the word "`hiding`", all public items but the named ones are made available for unqualified access in the current namespace.

Example: `import math.Rational hiding(gcd)`

Note that for the names listed in a hiding list, the re-export and the renaming syntax do not make sense. Anyway, it is still possible to re-export the items effectively linked this way by placing a "`public`" before the "`hiding`".

[2] It is perfectly legal for a package to contain nothing but import declarations.

[3] However, binary backwards compatibility can not be achieved this way, because on the JAVA and JVM level any class, method, etc. has only one unambiguous name.

5.3.6 Name clashes

It is permitted to overwrite a previously imported name through a declaration or binding. All unqualified uses of that name are resolved to refer to the declaration from the current package, but the imported item is still available under its qualified name. The compiler shall emit a warning in such cases.

It is also possible that an import overwrites an item in the current namespace that was introduced through a previous import. The compiler must emit a warning in such cases. The unqualified name will link to the last imported item.

It is advised to use the import features described above to avoid such clashes.

5.3.7 Aliasing the package name

One can specify the name of the namespace into which items will be imported. This makes it possible to disambiguate the namespaces for imported packages.

Consider the case that one needs two packages whose namespace names would come out equal, as in

```
Example:
import binary.Tree
import balanced.Tree – will fail, Tree is already a name space
```

Here, the second import would fail, because namespace `Tree` already exists. The situation can be remedied like so:

```
Example:
import binary.Tree
import balanced.Tree as AVL – all items go in namespace AVL
```

5.3.8 Multiple import

One can import any package more than once, with different import lists or different namespaces.

```
Example:
import frege.List public(map, take)
import frege.List(sortBy)
import frege.List() L       – by the way: "as" is optional
```

This would be equivalent to

Example:
import frege.List(public map, public take, sortBy)
import frege.List() as L

All public items of package `frege.List` can be named with qualifier `L.` or `List.` and the names of the functions `List.map`, `List.sortBy` and `List.take` don't need to be qualified. In addition, `List.map` and `List.take` are re-exported, so that they will be known in every package that imports this one unrestricted.

Name clashes may occur if the effective import lists have elements in common.

5.3.9 Implicit `Prelude` import

The compiler behaves as if the top declarations in the source code contained the import declaration

```
import frege.Prelude()
```

before all other import declarations. This ensures that all prelude definitions can be accessed through namespace `Prelude` and, at the same time, that no other package my be imported as `Prelude`.

In addition, unless a user provided import clause explicitly imports package `frege.Prelude`, the compiler automatically provides a

```
import frege.Prelude
```

before all other import declarations. This results in the desired behaviour, namely that all common operators, types and so on can be used with unqualified names.

5.3.10 Rules for Namespaces

When a package is imported, all symbols defined in that package are made available and can then be used in the importing package. However, this does not apply to namespace names themselves. Namespaces exist only during compilation and are a means to organize the symbol table.

Therefore, namespaces used in the source code of the imported package cannot be referenced in importing packages.

5.3.11 Importing packages with instances

Ambiguities arise when an imported package contains an instance for the same type class and data constructor as another imported package.

In such cases, the compiler will use the instance that was imported last and for all conflicting instances a warning message like

> Example: S.fr:3: data type A.A is already an instance of class Show (B.Show_A introduced on line 4)

will be emitted. The message informs about the position of the import that contains a conflicting instance (here line 3 in source file `S.fr`), the data type that was instantiated (`A.A`), the type class this instance was for (`Show`) and the internal name of the already existing instance (`B.Show_A`) and where it was introduced (on line 4). Hence, in this package, the `instance Show A` from the package that is associated with namespace `B` whose import declaration can be found at line 4 is used.

Chapter 6

Predefined Types and Classes

The FREGE Prelude contains predefined classes, types and functions that are implicitly imported into every FREGE program. In this chapter, we describe the types and classes found in the Prelude. Most functions are not described in detail here.

6.1 Standard Frege Types

These types are defined by the FREGE Prelude. Numeric types are described in section 6.3. When appropriate, the FREGE definition of the type is given. Some definitions may not be completely valid on syntactic grounds but they faithfully convey the meaning of the underlying type.

6.1.1 Booleans

```
data Bool = pure native boolean
```

The boolean type `Bool` is the primitive JAVA type `boolean`. The boolean values are represented by two keywords, `false` and `true`, see also subsection 2.7.1.

Basic boolean functions are `&&` (and), `||` (or), and `not`. The unary operator `!` is an alias for `not`. These operations are implemented in such a way that the corresponding JAVA operator is employed. This could promote unwanted strictness from the second operands of `&&` and `||`. For such cases, functions `und` and `oder` are provided that are lazy in their second operands.

The name `otherwise` is defined as `true` to make guarded expressions more readable.

`Bool` has instances for `Show`, `Eq`, `Ord`, `Enum` and `Bounded`.

Difference to HASKELL 98/2010: The constructors `True` and `False` do not exist.

6.1.2 Characters

```
data Char = pure native char
```

The character type `Char` is the primitive JAVA type `char`, whose values, according to the Java Language Specification are *16-bit unsigned integers representing UTF-16 code units*. The lexical syntax for characters is defined in section 2.7.3; character literals denote values of type `Char`. Type `Char` is an instance of the classes `Show`, `Eq`, `Ord`, `Enum` and `Bounded`.

6.1.3 Strings

```
data String = pure native java.lang.String
```

The type `String` is the JAVA type `java.lang.String`. For most `java` methods that work on strings, there is a corresponding native function binding.

`String` is an instance of the classes `Show`, `Eq`, `Ord`, `Empty`, `ListLike` and `ListSource`. The latter one allows string values to stand on the right side of the arrow in list comprehension generators, thus providing silent conversion to list of characters.

Further operations are explicit conversion to and from list of characters with `unpacked` and `packed`, conversion to various numeric types and access to individual characters of a string with integer indexes.

> Difference to HASKELL 98/2010: Strings are not list of characters, though conversion functions to and from lists exist.

6.1.4 Predefined Algebraic Data Types

Lists

```
data [a] = [] | a : [a]    -- this syntax is not really allowed
```

Lists are an algebraic data type of two constructors, although with special syntactic support, as described in subsection 3.1.4. The first constructor is the null list, written [] ("nil"), and the second is `:` ("cons"). The module `frege.prelude.PreludeList`, whose definitions are automatically imported in every FREGE program unless prevented by the user, defines many standard list functions.

Lists are an instance of classes `Show`, `Eq`, `Ord`, `Empty`, `ListLike`, `ListSource`, `Monad` and `Functor`.

Tuples

The tuple types are convenient product types for grouping a fixed number of values that can have different types. Tuple types with 2 to 26 components are predefined. Special syntactic support for tuples is described in subsection 3.1.3.

The following functions are defined for pairs (2-tuples): `fst`, `snd`, `curry`, and `uncurry`. Similar functions are not predefined for larger tuple.

The Prelude provides instances for classes `Show`, `Eq`, `Ord` and `Bounded` for pairs and 3-tuples. Instances of the same classes for tuple sizes 4 to 7 are predefined in package `frege.data.Tuples`.

The `zip`/`unzip` family of functions is also available by default for tuples of size 2 and 3. The package `frege.data.List` makes them available for tuples up to size 7.

The Unit Datatype

```
data () = ()    -- pseudo syntax
```

The unit type () is an enumeration with just one constant, which is also named (). The unit type is often the result type of impure functions that exist for their side effects.

6.1.5 Function Types

Functions are an abstract type: no constructors directly create functional values. The following simple functions are found in the Prelude: `id`, `const`, `(.)`, `flip`, `($)`.

6.1.6 ST, IO and RealWorld

```
abstract data ST s a = ST (s -> a)
    where  run :: forall a. (forall s.ST s a) -> a
abstract data RealWorld = RealWorld
type IO = ST RealWorld
```

The abstract `ST` type encapsulates impure operations, for an in depth discussion see [10].

In short, it is possible to model program actions that are impure only locally, but can be regarded as overall pure. For example, an array could be built from a list, and then computations that need fast indexing could be performed. Once the result is computed, the array would be gone and garbage collected.

Such computations can be performed with the help of the `ST.run` function. Note the higher rank polymorphic type of that function, which not only prevents execution for `ST` actions that are not polymorphic in the type argument s, but also escape of impure values that have the s in their type.

The very same mechanism is used for input/output. Functions with return type `IO a` produce `ST RealWorld a` values that can **not** be run with `ST.run`.

6.1.7 Exceptions

FREGE uses the JAVA exception facilities to model undefined or erroneous behavior.

The function `error`, when applied to a string and evaluated, constructs and throws such exceptions. Pattern matches and pattern guards can fail to match, the compiler adds code that raises exceptions in such cases.

Likewise, exceptions can be thrown in pure native code. It is possible, and, in the case of so called *checked exceptions*[1] unavoidable to catch such exceptions and return them instead of the expected result.

Last but not least, the JAVA virtual machine can throw exceptions.

Exceptions can be catched, raised and acted upon in ST actions, including native impure code.

There is support for creating custom frege exception types. not yet implemented

6.1.8 Other Types

```
data Maybe a = Nothing | Just a
derive Eq (Maybe a); derive Ord (Maybe a); derive Show (Maybe a)

data Either a b = Left a | Right b
derive Eq (Either a b); derive Ord (Either a b);
derive Show (Either a b)

data Ordering = Lt | Eq | Gt
derive Enum Ordering
derive Show Ordering
```

The type `Ordering` is used by `(<=>)` in class `Ord`. The Prelude provides functions `maybe` and `either`.

An alternate syntax is supported for nested `Either` types.

```
type (a|b) = Either a b
type (a|b|c) = Either (Either a b) c
type (a|b|c|....|z) = Either (Either (Either (Either ...))) z
```

The `|` is left associative, and acts like an infix `Either` type constructor. Hence the type $(t_1|t_2|t_3)$ is the same as `Either (Either` t_1 t_2`)` t_3.

[1] see [6, 11.2]

Note that a nested Either type must appear in parantheses, but inside the outer pair of parentheses arbitrary many `Either` alternatives can be written.

A value v of the above type could be deconstructed most easily with the `either` function like

Example: $(f_a$ `` `either` `` f_b `` `either` `` $f_c)$ v

where the infix `either` operators correspond to the `|` type constructors, and the positions of the f_i correspond to the values of type t_i.

6.2 Standard Frege Classes

TODO: To be written later.

6.3 Numbers

TODO: To be written later.

Chapter 7

Input/Output

TODO: write it

Chapter 8

Native Interface

TODO: This chapter is not yet complete.

In this chapter, we describe how native data types and native functions work, establish some conventions for the work with mutable data types and give a receipt for creating correct native function declarations.

8.1 Purpose of the Native Interface

The language constructs introduced so far make it possible to write *pure* functional programs. Pure functional programs consist of pure functions that work on immutable data. For the purpose of this discussion, we define these terms as follows:

pure function A function f is pure, if the following holds:

- f computes the same result when given the same argument values during execution of the program that contains f.
- The result must only depend on the values of the arguments, immutable data and other pure functions. Specifically, it may not depend on mutable state, on time or the current state of the input/output system.
- Evaluation of f must not cause any side effect that could be observed in the program that evaluates f. It must not change the state of the real world (such as magnetic properties of particles on the surface of a rotating disc).

This definition is not as rigid as others that can be found in the literature or in the Internet. For example, we may regard a function

```
getenv :: String → String
```

as pure, even if it depends on some hidden data structure that maps string values to other string values (the environment), provided that it is guaranteed that this mapping remains constant during program execution.

Regarding side effects, we exclude only effects in the real world (input/output, but not physical effects in the CPU or memory chips) and effects that are observable *by the program that caused it.* For example, evaluation of the expression

```
s1 ++ s2
```

where `s1` and `s2` are string values, will cause creation of a new JAVA `String` object and mutation of some memory where the new string's data is stored, this could even trigger a garbage collector run with mutation of huge portions of the program's memory. Yet, all this happens behind the scene and is observable only by *another* program such as a debugger or run time monitor, if at all.

We also do not insist that application of a pure function with the same argument values must return the same value in different executions of the program[1].

immutable values A value is immutable if there are no means to *observably* change it or any values it contains or references.

[1]If we did insist on equal return values for application of a pure function with equal arguments *in different executions of a program*, we could not use any functionality provided by the underlying platform, which is in our case JAVA and the JAVA Virtual Machine. For we could not guarantee that certain constants or pure methods we use will be unchanged in the next update or version of that component.

Alternatively, one could of course define the term *program* in such a way that it encloses a specific version of the JAVA runtime system (and in turn specific versions of the libraries used by JAVA, and the libraries used by those libraries down to the operating system and even the hardware). But then, the term *program* would become almost meaningless. Suppose, for example, that some chip producer finds a bug in one of the floating point units that he produces, which causes incorrect results to be returned by certain floating point divisions. Suppose further, that the faulty CPU chip is replaced by a fixed one in the computer used by an extremely rigid functional purist, who insists that functions must produce the same value across different program executions. Then, this person must either admit that some function he wrote was not pure (because it suddenly produces different results for the same argument values) or he must regard his program as having changed. He could, for instance, talk about how much more exact results "*this new version of my program*" produces, despite nobody hasn't changed a single bit on the hard disk!

This argumentation is not invalidated by pointing out that the faulty CPU did not behave according to their specification. It remains the fact that results of computer functions depend on the computing environment they are executed in, no matter if one likes it or not.

It is probably more rational to acknowledge that the idea of a function that *depends on nothing but its arguments* is a nice, but utterly idealistic one that must necessarily abstract away many aspects of reality. In practice, the result of a function $\backslash x \rightarrow x/3145727.0$, when defined in some computer language and executed in some computing environment depends not only on x, but also on how the floating point unit works, how exact the divisor can be represented in floating point, in short, it depends on the computing environment where it is run.

We do not understand the concept of functional purity so narrowly that we require the same result of a pure function in all thinkable computing environments. Rather, we admit realistically that results may be different in different computing environments. The session environment (the mapping of values printed by the command `env` in a UNIX session; similar features exist in other operating systems) is a part of the computing environment that is constant during program execution (in JAVA programs, that is). Hence, the result of a pure function in a program may depend on environment variables, according to our definition. It may depend on the arguments passed on the command line. Yet, it may not depend on the current directory, for obtaining the name of it or using it presupposes input/output to be performed. It may also not depend on JAVAs system properties, for those can be changed during program execution.

This is deliberately vague in view of the difficulties in JAVA when it comes to enforcing immutability. [2]

The native interface allows to call JAVA methods and use JAVA data in FREGE programs. Because JAVA reference values may be mutable and JAVA methods may not be pure functions, it provides means to differentiate between pure vs. impure methods and mutable vs. immutable values. Unfortunately, there exists no reliable general way to establish purity of JAVA methods or immutability of JAVA values. Therefore, the FREGE compiler must rely on the truthfulness of the annotations the programmer supplies.

Difference to HASKELL 98/2010: The native interface in FREGE corresponds to the *Foreign Function Interface* in Haskell 2010 ([3, Chapter 8]), but differs in the following points:

- There are no foreign exports, because all items defined in a FREGE module that are visible in other FREGE modules are also visible to any JAVA program anyway.
- Foreign imports (i.e. native declarations) always refer to items known in the JAVA virtual machine. The source language that was used to create those classes, interfaces or methods is immaterial.
- There are no calling conventions to choose from.
- Some simple marshaling between FREGE data and JVM data is supported.

8.2 Terminology and Definitions

Let's recall the general forms of native data and function declarations:

data $T\ t_1 \cdots t_i = \textbf{native}\ J$
native $v\ j :: t$

We call T a *native type* and J the *java type* associated with it. We also say that the FREGE type $(T\ t_1 \cdots t_i)$ denotes the JAVA type J. In this chapter, we'll use the abbreviations $jt\big(T\big)$ for *the* JAVA *type associated with or denoted by* T and $ft\big(J\big)$ for *the* FREGE *type that denotes* J.

If T is associated with one of `byte`, `boolean`, `char`, `short`, `int`, `long`, `float` or `double`, then T denotes a *primitive type*, otherwise it denotes a *reference type*.

We call v a *native value* and j the *java item* associated with it. If t is of the form $t_1 \to \cdots \to t_k \to t_R$, where t_R is not itself a function type, we call v a *native function*

[2] It is, for example.possible to break intended immutability with the help of the reflection API.

with *arity* k ($k \geq 1$) and *return type* t_R. The t_i are called *argument types*. For v's that are not native functions, the arity is 0 and the return type is t.

J and j are snippets of JAVA code and can be specified as identifiers, qualified identifiers or operator symbols as long as this does not violate the FREGE lexical syntax. In all other cases the code snippets can be given in the form of string literals. In the following sections, we will frequently use the value of j or just j. This is to be understood as the string that is described by the string literal, not the string literal itself.

8.3 Mapping between Frege and Java Types

Since all FREGE types must be mapped to JAVA types eventually, it is the case that every FREGE type of kind $*$ denotes exactly one JAVA type. The converse, however, is not true, since multiple FREGE types may map to one and the same JAVA type[3] and in addition, $ft(J)$ is only defined for those JAVA types that are made available to FREGE with a native data definition.

The Figure 8.1 shows a recommended mapping.

Type T	$jt(T)$	Comment
a	`java.lang.Object`	nothing is known about values of type a
$a \to b$	`frege.runtime.Lambda`	
`Bool`	`boolean`	
`Int`	`int`	and similarly for all other primitive types
`String`	`java.lang.String`	
`Integer`	`java.math.BigInteger`	
`[`a`]`	`frege.prelude.PreludeBase.TList`	
(a, b)	`frege.prelude.PreludeBase.TTuple2`	and similarly for tuples with more components
`JArray` a	`java.lang.Object`	generic array
`JArray` T	$jt(T)$`[]`	array of T
enumerations	`short`	
algebraic	$M.TT$	where T is defined in module M

Figure 8.1: Recommended type mapping

[3]Implementations are free to erase type arguments, so that, for example `Maybe String` and `Maybe Int` may denote the same JAVA type. In fact, since JAVA's type system is not powerful enough to deal with higher kinded types, it is very likely that type arguments will be erased.

8.4 Types with Special Meaning

The following types have a special meaning in the type signatures of native values and functions. They are used to require special handling of argument or return values (marshalling).

As far as FREGE is concerned, the corresponding values have exactly the declared types. The JAVA type, however, can be different.

`()` The unit type as argument type indicates an empty argument list for the JAVA method that implements the native function. The unit type is only allowed in argument position if the type is of the form $() \to t_R$. i.e. when the unit value is the only argument[4].

The unit type as return type indicates that the native function is implemented by a JAVA method that is declared `void`. The compiler supplies a wrapper that invokes the method and returns the value ().

`Maybe` a A `Maybe` type in argument position indicates that the JAVA method that implements a native function takes `null` values for the corresponding argument. The generated code will pass `null` for arguments with value `Nothing` and x for arguments of the form (`Just` x).

A `Maybe` type as return type indicates that the implementing method may return `null` values. The return value `null` will be mapped to `Nothing` and any other return value j to (`Just` j).

It therefore holds that $jt\big($`Maybe` $a\big)$ in native method's arguments or return values is $jt\big(a\big)$.

JAVA provides classes for boxed primitive values, like for instance `java.lang.Float`. If one needs to use a method that has an argument of a boxed type, one can use any FREGE type that is associated with the corresponding primitive type (i.e. `Float`). This works because JAVA performs *autoboxing*. However, if one ever needs to pass `null`, the corresponding argument type must be wrapped in `Maybe` (i.e. `Maybe Int`). For return types, the autoboxing works in a similar way. Yet, whenever it is not provably impossible that the method ever returns `null`, one must declare the return type as a `Maybe` type. Failure to do so may cause null pointer exceptions to occur at runtime.

The type wrapped by `Maybe` must not be any of the special types described here.

`Either` x t This type is to be used as return type instead of t when the implementing method is declared to throw checked exceptions or if it is known that it throws other exceptions that one needs to catch.

The x must be an *exception descriptor* whose definition follows along with two other concepts we will be using:

[4]Note that *instance methods* always have at least one argument, the so-called receiver. Hence the FREGE type of an instance method may never be $() \to t_R$

- If *jt*(x) is some JAVA type that implements `java.lang.Throwable`, then and only then is x a *throwable type*.
- If x is a throwable type, it is a valid *exception descriptor*.
- Let y be an exception descriptor and t a throwable type. Then `Either` y t is a valid exception descriptor.
- All other types are not exception descriptors.
- A type of the form `Either` x t where x is an exception descriptor and t is not an exception descriptor is called a *catching type*.

The nested `Either` syntax comes in quite handy here:

Example:

```
data BadCharset = pure native
          java.nio.charset.IllegalCharsetNameException
data UnsopportedCharset = pure native
          java.nio.charset.UnsupportedCharsetException
data CharSet = pure native java.nio.charset.Charset where
     pure native csForName java.nio.charset.Charset.forName
          ::  String -> (BadCharset|UnsupportedCharset|CharSet)
```

Code generation will create a wrapper method containing a `try` statement with `catch` clauses that catch the exceptions declared in x in left to right order[5].

Note that the JAVA language enforces certain rules for exception handling (see [6, 11]). The following points should be observed to avoid JAVA compiler errors:

- The catching type must list all checked exceptions that the method may throw.
- The catching type must not list any exception the method may not throw.
- In the catching type, more specific exceptions (in terms of the JAVA exception class hierarchy, see [6, 11.5]) must occur further left than less specific ones.

If the wrapper indeed catches one of the interesting exceptions, it constructs an appropriate `Left` value. Otherwise, if a value v is returned from the native method, the wrapper returns (`Right` v_m), where v_m is the value after marshalling of v, which takes place in cases where t is one of the types with special meaning as explained before.

A catching type is not valid as argument type for a native function.

t may not be another catching type nor a `ST` or `IO` type.

[5] The order is the same, regardless of notation of the type as `(Either (Either E1 E2) R)` or `(E1|E2|R)`. It could appear to be different only if one used type aliases like `type Rehtie a b = Either b a`, however, it is the order in the expanded form that counts.

`ST` *s t* This type must be used when the implementing method uses or produces mutable data. `ST` must be the outermost type constructor in the result type.

The compiler creates an appropriate wrapper function that constructs a `ST` action, which, when executed, runs the native method and returns its value in the `ST` monad. Native functions declared this way can also be used in the `IO` monad.

`IO` *t* This type must be used when the implementing method has any side effects. `IO` must be the outermost type constructor in the result type. The compiler creates an appropriate wrapper function that constructs an `IO` action, which, when executed, runs the native method and returns its value in the `IO` monad.

For an overview of possible return values of native functions see Figure 8.2.

8.5 Exception Handling

There are two mechanisms that deal with exceptions thrown from native functions. The first mechanism uses catching types as described in the previous section.

The idea behind the catching types is to encode the actual return value and the catched exceptions in a value of type `Either`. This makes it in effect impossible to overlook that the function may throw exceptions. The user has many choices to get at the actual return value: case expressions, the `either` function, or higher order functions like `fmap`. But he cannot pretend that the function just returns a value without taking the possibly of thrown exceptions into account.

This approach is fine in many cases, but may be a bit laborious in others. For example, when doing input/output using JAVA APIs, almost every function will potentially throw some incarnation of `java.io.IOException`. It is neither desirable nor even possible to ignore such exceptions, and this could lead to a programming style where every function call is followed by long winded error handling. In the end, even small sequences of I/O actions could result in deeply nested case expressions. Or the code could get extremely fragmented along the lines of:

Example:

```
processFile path = do
  openFile path >>= either handleOpenException continueRead
continueRead f = do
  readLine f >>= either handleNoLine (processLine f)
processLine f ln = do
  result1 <- do something with the line
  readLine f >>= either handleNoOtherLine (processRest f result1)
processRest f res1 ln2 = do ....
handleOpenException ex = do ...
handleNoLine ex = do ....
handleNoOtherLine ex = do ....
```

Should a function throw different exceptions that require separate handling, it will get even more complex.

Therefore, there is another approach, that allows to catch and handle exceptions explicitly in the `ST` or `IO` monads[6].

The explicit approach rests on the idea that the return types of native functions are left as they are, except for the `ST` or `IO` wrapper. This requires, however, that checked exceptions, which are part of JAVA method signatures must be eliminated, since FREGE function types carry no information about them.

Put differently, a JAVA method that invokes a method whose signature states that checked exceptions may be thrown must do so either inside of a throw/catch statement where the checked exception is catched, or the calling method must itself state that it throws the exception.

Neither are there statements in FREGE, let alone try/catch statements, nor are exceptions part of the contract of FREGE functions. Hence, the only possibility is to catch the exceptions where they arise, wrap them in unchecked exceptions and re-throw the latter ones, which can be done without changing the method's signature.

To achieve this, the following form of a native function declaration must be used:

native v j:: t **throws** $x_1, \cdots, x_k$

Here, t is the function type whose return type must be a `ST` or `IO` type, and the x_i ($i \geq 1$) are throwable types that denote JAVA exceptions to re-throw. The generated wrapper will create a try/catch statement that catches, wraps and re-throws all exceptions denoted by the x_i.

[6]The details of how to use the key functions `catch` and `finally` and how to write exception handlers can be found in the API documentation.

Declaration of Throwable Types

Throwable types are to be declared like in the following example:

Example:

```
data IOException = pure native java.io.IOException
derive Exceptional IOException
```

The declarations shown are imported from package `frege.java.IO` and are thus available in every FREGE program. However, not all exceptions from the JRE may be predeclared, and exceptions from 3rd party libraries will certainly not be predeclared.

The convention is to view throwable types as abstract, immutable tokens, hence the `pure native` declaration. In addition, if one wants to catch exceptions explicitly with the `catch` function, one needs to make the throwable type an instance of `Exceptional`. This is because the type of `catch` is

```
catch :: Exceptional e => IO a -> (e -> IO a) -> IO a
```

where the second argument is the exception handler. Hence, to use an exception handler that handles exception e with `catch`, e must be an instance of type class `Exceptional`.

Rules for Native Functions that Throw Exceptions

- The set of checked exceptions declared in the signature of the JAVA method must be a subset of the throwable types in the native declaration, otherwise the generated JAVA code will not compile. Put differently, all possible checked exceptions must be considered in one way or the other by the FREGE declaration.
- Catching types and the `throws` clause can be combined, as long as the set of exceptions handled with catching types is disjunct from the set of exceptions named in the `throws` clause. The exceptions occurring in the catching type will never be thrown, but wrapped in the `Either` value.
- Catching types as well as throws clauses can contain additional unchecked exceptions.
- Native functions with `throws` clauses allow one to deliberately ignore exceptions, even checked ones. Needless to say, except for toy programs, this is not acceptable, as any unhandled re-thrown exception will terminate the thread where it was raised.

8.6 Mutable and immutable Java data

Most JAVA objects are potentially mutable. However, from the FREGE point of view the question is whether objects of a certain class can be mutated with the operations that are available. If one restricts oneself to non-mutating native functions, a JAVA class can be considered immutable for the purpose at hand.

An example for such a border case is `java.math.BigInteger`, which is technically not immutable at all, yet offers a quite functional interface. With the understanding that malicious JAVA code could manipulate big integer values, this class is regarded immutable in FREGE, and serves as implementation for type `Integer`. Of course, none of the operations provided for `Integer` does manipulate the underlying JAVA object in any observable way.

It follows a discussion of the three main categories of native data that are recognized by the FREGE system.

8.6.1 Immutable Only Data

Native data types declared with the `pure native` keywords are regarded as truly immutable. For such data, it doesn't matter, if a reference to one and the same object is shared between JAVA and FREGE code or even between different threads.

There are no restrictions on the use of values of such types.

> Note: The behavior and result of programs that incorrectly declare mutable JAVA types as immutable is undefined. It is a programming error the FREGE compiler unfortunately cannot detect.

8.6.2 Mutable/Immutable Data and the ST Monad

The normal case are native data that are (by the very nature of the JAVA language) mutable, but can nevertheless be used to model overall pure functions. The corresponding `data` declaration for such types uses the keyword `native` without additional keywords.

A characteristic property of JAVA classes that belong to this category is that they offer a mix of pure and impure methods.

Common examples are arrays, collections and types like `java.lang.StringBuilder`, a mechanism for efficient construction of character strings.

The problem in FREGE is to distinguish between values (i.e. JAVA objects) whose references could be aliased and held in parts of the code that is not written in FREGE (like library code) or even in different threads and, on the other side, "safe" values that are only known in the FREGE realm.

The solution is to have a different compile time type for values that are actually mutable and values that are safe. Specifically, we will wrap mutable values of type M in the following type:

```
abstract data Mutable s m = Mutable m
```

Observe that `Mutable` itself is declared `abstract`, which makes it impossible to actually create or deconstruct such a value in FREGE code, as the value constructor is inaccessible.

Furthermore, the following rules will be enforced for every native type M that is not `pure`:

1. The only way M can appear in the return type of a native function is `Mutable` s M
2. A pure native function may not return `Mutable` s M
3. The only way M can appear in the argument type of an impure native function is `Mutable` s M
4. The phantom types of type constructors `ST` and `Mutable` must all be the same in the type of a native method. The phantom type is either a type variable or the constant `RealWorld`.
5. A pure function must not normally have arguments of type `Mutable` s M.
6. A type `Mutable` s a where a is not a native type or was declared `pure native` or `mutable native` (see subsection 8.6.3) is illegal in the type of native functions.

Rule 1, 2 and 4 make sure that M values can be created only in the `ST` or `IO` monad, and that their type appears to be `Mutable` s m, where s is the appropriate phantom type. This means also that such values cannot escape from the corresponding monad. For the `IO` monad, this is guaranteed because nothing can escape it. For the `ST` monad, the phantom type prevents execution for any `ST` action that has a type like

```
ST s (Mutable s M)
```

from pure code (yet, there is nothing wrong *per se* with such a ST action: it could be used in another ST/IO action, for example).

The motivation for rule 5 is the following: From what has been said in the paragraphs above it should be clear that mutable values can only appear in the `ST` monad. If we allowed passing mutable data to pure functions (in a let clause, say), their result would depend on whether the monadic code around it actually mutates the value or not. Consider the following slightly contrived example:

Example:

```
data Foo = native some.Foo where
  native new ::  () -> ST s (Mutable s Foo)
  native setBar ::  Mutable s Foo -> Int -> ST s ()
  pure native getBar ::  Mutable s Foo -> Int -- WRONG!!!
bad ::  ST s Int
bad = do
  foo <- Foo.new () -- create a value
  foo.setBar 42 -- set the property
  let r = foo.getBar -- seemingly pure getter
  foo.setBar 43 -- reset the property again
  return r
```

There is no way to tell what the value of `(ST.run bad)` should be. The `let`, judging by its data dependencies, could get floated left, so that its code in the compiled output appears before the first `setBar`. Or the compiler could in-line the definition of `r` in the argument of `return`. Or it could leave it as is. In any case, there is no need to evaluate `foo.getBar` right away. The only thing that can be said is that the result, should it be evaluated at all, would depend on the *then* current state of the object denoted by `foo`.

The crucial point here is the declaration of the getter method that wants to make us believe that a getter method applied to a mutable object could be pure. In the general case, this is not so. Hence rule 5, which forbids it.

There are, however, cases where a mutable object does have invariant properties. An easy example is getting the length of an array.

Making Safe Copies of Mutable Values

Often, mutable values are used only temporary, as is frequently the case with string buffers. In other cases, it are precisely the methods that are pure on the condition that the (technically) mutable value actually is not mutated any more, that interest us. In the case of arrays, for example, it is common to initialize them (from a list, say), and then enjoy the fast random read-only access in a pure computation.

The rules established above make sure that mutable/immutable values are properly treated. But they do too much insofar as they do not even allow escape of a safe copy of a mutable value yet.

For this purpose, the `Mutable` type has the following two functions:

```
freeze :: Freezable m => Mutable s m -> ST s m
thaw   :: Freezable m => m -> ST s (Mutable s m)
```

that allow for controlled construction of safe M-values from mutable ones and vice versa. m must have an instance for type class `Freezable`, whose subclasses `Cloneable` and

`Serializable` are appropriate for JAVA types that implement the JAVA interfaces with the same names. For instance, making a type T, that is associated with a JAVA type that implements `java.lang.Cloneable`, an instance of `Cloneable` (the type class) is as easy as stating:

```
instance Cloneable T
```

After having properly[7] frozen an object, there should be only one reference through which the copy could be mutated. But because we get this reference as M instead of `Mutable` s M, we cannot pass it anymore to impure methods due to rule 3. Should one ever need to do this, the frozen value must be thawed first using `Mutable.thaw`, which is supposed to make another copy that can be mutated without affecting the still existing frozen value.

Cheating with Rule 5

As said before, there are often invariants (like array length) on mutable data that can be expressed as pure native functions, but rule 5 forbids it. Because the whole apparatus described here is to help get things right, not to prevent people that know what they are doing from doing it, there is the following additional function for `Mutable`s:

```
readonly :: (m -> b) -> (Mutable s m) -> ST s b
```

This way, we can "lift" pure native functions to work on mutable values. The understanding should be that the result must not depend on the mutable state of the object.

Note that `(readonly id)` can be used to coerce a mutable value to an immutable one. This is justified in cases where the value was created locally, and one knows that no aliases exist, and the value is henceforth to be regarded as immutable without actually copying something.

8.6.3 Mutable Only Data

A native type M whose values would, by the rules above, appear *everywhere* with type `Mutable RealWorld` M, is called a *mutable only* type. It can be declared like

```
data M = mutable native ...
```

and can appear in type signatures as just M. The compiler will then apply the rules above *as if* the type was `Mutable RealWorld` M , with the following consequences:

- A value of type M can only be created in native functions whose return type is `IO` M (due to rules 2 and 4).

[7] Unfortunately, however, properly freezing is not easy in the general case. Most often, merely cloning is not enough. Actually implementing `freeze` (and the counterpart `thaw`) requires great care.

- If M occurs as argument of a native function, that function must have type `IO` a for some type a (due to rule 3).
- `Mutable.freeze` is not applicable to M. The type checker expects a `Mutable` s m, but the *as if* above concerns only sanity checks for types of native functions, the normal typing rules will never actually unify `Mutable` s m with M.
- It would be possible to `Mutable.thaw` a value of type M, yet no native function type can contain the type `Mutable` s M by rule 6, and hence nothing could be done with the supposedly safe copy (except `Mutable.freeze` it again).

8.7 Pure Java methods

A *pure* JAVA method is a pure function, i.e. it has the following properties:

- Its return value depends only on its arguments, on constant data and on nothing else.
- It has no side effects.

Dually, a function is not pure if at least one of the following holds:

1. The method performs any input or output operations.
2. The method changes data that is either visible outside of the method or influences the outcome of subsequent invocations of any other method.
3. It matters, when or how often a method is invoked.
4. It can return different values on different invocations with identical arguments. This can be the case when the result of the function depends on mutable state.

In JAVA, like in most imperative languages, the use of impure functions is widespread. Examples for methods that are impure

1. creation or removal of files, open a file, read from or write to a file
2. any so called *setter*-method, that changes state of an object. Also, random number generators that employ a hidden *seed*.
3. methods that depend on the time of day
4. methods that depend on default locale settings like number or date formatting that could be changed during the runtime of a program, methods that read so called system properties, registry settings or configuration files.

Nevertheless, JAVA provides many methods and operations that are pure. Most methods of `java.lang.String` are, as well as the methods of `java.util.BigInteger` and the operations on primitive data types. Many object constructors and getter methods are also pure when they create or operate on immutable values.

A pure JAVA method must be declared as such by starting the native declaration with the `pure` keyword. It cannot have a return type `ST` *s a* or `IO` *a*.

8.8 Deriving a Frege `native` declaration from a Java method signature

For every JAVA method signature

$$t\ name(t_1\ a1,\ t_2\ a2,\ \cdots,\ t_n\ ai)\ \text{throws}\ e_1,\ e_2\ \cdots\ e_k$$

where t is the return type, n is the fixed number of arguments [8], t_1, t_1, $\cdots$, t_n are the types of the arguments, k is the number of exceptions thrown and the e_i are the respective exceptions, the FREGE type must be

$$() \rightarrow\ f_r \text{ when } n \text{ is } 0$$
$$f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_n \rightarrow f_r \text{ when } n > 0 \text{ and for all } i\ jt\big(f_i\big) \text{ is } t_i$$

Finding the return type If t is `void`, the principal return type is `()`. In all other cases, it is the FREGE type that is associated with t.

TODO: continue me

8.9 Java Constructs Supported by Native Declarations

Depending on the type and the form of the JAVA item, different JAVA constructs are supported. Except for static field access expressions, native declarations introduce a FREGE item with a function type. The only special things about such native functions are

1. that the compiler assumes that they are *strict* in all arguments[9] so that lazy values are never passed to native functions

[8] Argument lists with a variable number of arguments are not supported.

[9] A function is said to be strict in an argument if it with necessity evaluates to undefined if the undefined value is passed for that argument.

2. that their type must not have type class constraints.

In all other respects, native functions are indistinguishable from all other functions with the same type.

8.9.1 Static Field Access Expression

Native values with arity 0 can be used to access static fields of a JAVA class. The corresponding FREGE value is computed once upon beginning of the program.

Translation: Let v be declared as

native $v\ j :: t$

where t is not a function type. Then the expression v will be compiled to the following JAVA code: j

Example: Consider the following definition

```
native pi java.lang.Math.PI :: Double
```

Then the JAVA expression generated for `pi` will be

```
java.lang.Math.PI
```

The java item should be given as a fully qualified name.

8.9.2 Instance Field Access Expression

Public instance fields can be accessed with an instance field access expression.

Translation: Let v be declared as

native v " $.j$" $:: t_o \to t_r$

where neither t_o nor t_r are function types, t_o is associated with a JAVA reference type and j is an identifier that is valid in JAVA. Then the application $v\ x$ will be compiled to a field access expression on the JAVA object that is denoted by expression x.

8.9.3 Method Invocation Expression

Translation: Let v be declared as

native $v\ j :: t$

where t is a function type with arity k.
An application of k arguments to v will be compiled to a method invocation, where special types are marshalled as documented before.

If j is a simple name, the type of the first argument must be a native reference type, and an instance method invocation with the first argument as receiver is generated.

If j is a qualified name, a static method invocation will be generated.

Like with all FREGE functions, extra code will be generated to arrange for curried applications. In certain cases, depending on special return types, a wrapper method will be created.

8.9.4 Class Instance Creation Expression

Class instances (JAVA objects) can be created with class instance creation expressions.

Translation: Let v be declared as

native v `new` $:: t$

where t is a function type with arity k whose return type is associated with a JAVA reference type, possibly wrapped in a catching type or a `ST` type, but not in `Maybe`, as a JAVA constructor cannot return null. Applications of v to k arguments will eventually invoke the appropriate constructor of the JAVA class that is associated with the return type.

8.9.5 Binary expressions

Translation: Let v be declared as

native $v\ j :: t$

where t is a function type with arity 2 and j is an operator.
An application of 2 arguments a and b to v will be compiled to a binary expression, where the compiled a stands left of j and the compiled b stands right.

The FREGE compiler cannot check that the operator is valid in JAVA nor that it can be applied to arguments of the given types.

Example:

```
pure native ++ + ::  String -> String -> String
foo = "foo" ++ "bar"
```

defines the FREGE operator ++. The top level binding `foo` is compiled to:

```
public final static java.lang.String foo = "foo" + "bar"
```

8.9.6 Unary expressions

Translation: Let v be declared as

native $v\ j :: t$

where t is a function type with arity 1 and j is an operator.
An application of an arguments a to v will be compiled to an unary expression, where j is applied to the compiled a.

The FREGE compiler cannot check that the operator is valid in JAVA nor that it can be applied to an argument of the given type.

Example:

```
pure native not !  ::  Bool -> Bool
foo x = not x
```

defines the FREGE function `not`. The top level binding `foo` is compiled to:

```
public final static boolean foo(boolean arg) {
  return !(arg);
}
```

8.9.7 Cast expressions

Translation: Let v be declared as

native v `"(`j`)"` :: t

where t is a function type with arity 1 and j is a JAVA type.
This works like an unary operator with the funny name `(`j`)`

The FREGE compiler cannot check that the cast is valid in JAVA.

Example:

```
pure native long2int "(int)" ::  Long -> Int
foo x = long2int x
```

defines the FREGE function `long2int`. The top level binding `foo` is compiled to:

```
public final static int foo(long arg) {
  return (int)(arg);
}
```

declared return type	**expected java signature**	**example java code or comment**
`()`	`void meth(...)`	`System.exit()`[1]
`(Ex|())`	`void meth(...) throws` *jt*`(Ex)`[2]	`System.arraycopy(...)`[1]
`IO ()`	`void meth(...)`	`System.gc()`
`IO (Ex|())`	`void meth(...) throws`[2] `...`	`(Thread)t.start()`
`Int`	`int meth(...)`	`(String)s.length()`
`String`	`java.lang.String meth(...)`	`(String)s.concat(...)`
a[3]	*jt*(a) `meth(...)`	general rule, note that previous 2 lines are no exceptions
`Maybe Int`[4]	`java.lang.Integer meth(...)`	`Integer.getInteger(...)`
`Maybe` a[3]	*jt*(a) `meth(...)`	general rule for any a that is not a primitive type
(`Ex|`a)[5]	same as for a + `throws`[2] `...`	`Float.parseFloat(...)`[7]
`IO` a[6]	same as for a	`System.nanoTime()`[8]
`[`a`]`[3]	*jt*(a)`[]` meth(...)	`(String)s.split(...)`[9]
`[`a`]`[3]	`Iterator<`*jt*`(`a`)>` `meth(...)`	`List<String>l.iterator(...)`[9]

Figure 8.2: Well formed native return types

[1]However, the compiler can not be fooled into thinking that such a method is actually pure. Therefore, despite the return type is well-formed, it's still invalid. If you need a function that maps any argument to `()`, consider `const ()`

[2]If the JAVA method actually declares checked exceptions, the return type must be a catching type or the `throws` clause on the native declaration must be used.

[3]where a is no type with special meaning

[4]This works in a similar way for all other primitive types. The code generated by the compiler expects a value of the corresponding boxed type or `null`. Note that, because JAVA does autoboxing of primitive values, methods that return the corresponding primitive value are also allowed.

[5]where a is not another catching type and not an `IO` type

[6]where a is not another `IO` type

[7]in this case, a would be `Float`

[8]in this case, a would be `Long`

[9]in this case, a would be `String`

Chapter 9

Specification of Derived Instances

A *derived instance* is an instance that results from a `derive` declaration. The body of a derived instance is derived syntactically from the definition of the associated type. Derived instances are possible only for certain classes known to the compiler.

9.1 Derived Instances for Eq

Instances of `Eq` are types whose values can be compared for equality. `Eq` instances can be derived for all algebraic data types.

Translation: Let P be a n-ary $(n \geq 0)$ type constructor for a product type with k-ary $(k \geq 0)$ data constructor C:

data $P\ u_1 \cdots u_n = C\ ct_1 \cdots ct_k$
Then
derive `Eq` $(P\ t_1 \cdots t_n)$
is equivalent to:
instance `Eq` $(Eq\ t_1, \cdots, Eq\ t_n)$ => $(P\ t_1 \cdots t_n)$ **where**
$C\ a_1 \cdots a_k$ == $C\ b_1 \cdots b_k$ =
true && a_1 .== b_1 && $\cdots$ && a_k .== b_k
hashCode x = $\cdots$

The generated expression for the == operator returns **true** if all subcomponents of the left operand are pairwise equal with the corresponding subcomponents of the right operand, otherwise the result is **false**.

Note that the special case $k = 0$ is trivial: such a type has only one value C and the derived == returns always **true**.

The generated expression for `hashCode` computes a value of type `Int` suitable for use in hash tables and similar data structures. In the process of doing so, all sub-components of the value will be evaluated recursively. The result is undefined for infinite values.

The case gets only marginally more complex with sum types.

Translation: Let S be a n-ary ($n \geq 0$) type constructor for a sum type with m ($m \geq 2$) data constructors $C_1, \cdots, C_m$ and arities $k_1, \cdots, k_m$:

data $S\ u_1 \cdots u_n = C_1\ ct_{1_1} \cdots ct_{k_1} \mid \cdots \mid C_m\ ct_{m_1} \cdots ct_{k_m}$
Then
derive `Eq` $(S\ t_1 \cdots t_n)$
is equivalent to:
instance `Eq` $(Eq\ t_1, \cdots, Eq\ t_n)$ => $(S\ t_1 \cdots t_n)$ **where**
a == b = **case** (a, b) **of**
$(C_1 a_1 \cdots a_{k_1}, C_1 b_1 \cdots b_{k_1})$
$\rightarrow$ **true** && a_1 .== b_1 && $\cdots$ && a_{k_1} .== b_{k_1}
$\cdots$
$(C_m a_1 \cdots a_{k_m}, C_m b_1 \cdots b_{k_m})$
$\rightarrow$ **true** && a_1 .== b_1 && $\cdots$ && a_{k_m} .== b_{k_m}
_ $\rightarrow$ **false**
hashCode x = $\cdots$

The expression a == b evaluates to **true** if both a and b were constrcuted with the same data constructor and their corresponding subcomponents are pairwise equal.

Derived Instances for Ord

The `Ord` class is used for totally ordered datatypes. It is a subclass of `Eq` and inherits the operations `==` and `!=`. It defines one new operation `<=>` that must be implemented by all instances, and operations `<`, `<=`, `>`, `>=`, `max` and `min` in terms of `<=>`.

The compare function `<=>` [1] compares two values and returns a result of type `Ordering`, which is defined as [2]

```
data Ordering = Lt | Eq | Gt
```

Instances of `Ord` can be derived for all algebraic data types that are either already instances of `Eq` or have an implementation for `hashCode`.

The translation shown here does not handle the case of the trivial product type. Such a type will have an implementation of `<=>` that always returns `Ordering.Eq`.

For product types, the generated expression compares the components a_i, b_i from 1 to $k-1$; the first result r_i that does not signify equality is the result of the overall comparison. Otherwise, if all component pairs up to $k-1$ compare equal, the result is the ordering of the last component pair, a_k .<=> b_k.

[1] The alias `compare` is provided for HASKELL compatibility.
[2] Aliases `LT`, `EQ` and `GT` are provided for HASKELL compatibility.

Translation: Let P be a n-ary $(n \geq 0)$ type constructor for a product type with k-ary $(k \geq 1)$ data constructor C:

data $P\ u_1 \cdots u_n = C\ ct_1 \cdots ct_k$
Then
derive `Ord` $(P\ t_1 \cdots t_n)$
is equivalent to:

```
instance Ord (Ord t_1, ..., Ord t_n) => (P t_1 ... t_n) where
  (C a_1 ... a_k) <=> (C b_1 ... b_k) = case a_1 <=> b_1 of
    Eq ->
      ...
        case a_{k-1} <=> a_{k-1} of
          Eq -> a_k <=> b_k
          r_{k-1} -> r_{k-1}
      ...
    r_1 -> r_1
```

Derived instances for sum types make use of the Prelude function

```
constructor :: any -> Int
```

which returns the index of the constructor for algebraic data types.[3]

The code for sum types first sorts out the cases where the constructors are not the same; the result in such a case is the ordering of the constructors. The remaining m cases compare nullary constructors equal to themselves, values with unary constructors compare just like the components compare and values with constructors of higher arity compare like the tuples constructed from their components would compare when k_i-ary tuples had a derived instance of `Ord`.

[3]The constructors are numbered starting from 0 in the order they appear in the data declaration.

Translation: Let S be a n-ary $(n \geq 0)$ type constructor for a sum type with m $(m \geq 2)$ data constructors $C_1, \cdots, C_m$ and arities $k_1, \cdots, k_m$:

data $S\ u_1 \cdots u_n = C_1\ ct_{1_1} \cdots ct_{k_1} \mid \cdots \mid C_m\ ct_{m_1} \cdots ct_{k_m}$
Then
derive `Ord` $(S\ t_1 \cdots t_n)$
is equivalent to:
instance `Ord` $(Ord\ t_1, \cdots, Ord\ t_n)$ => $(S\ t_1 \cdots t_n)$ **where**
a <=> b = **case** (`constructor` a)<=> (`constructor` b) **of**
`Eq` $\rightarrow$ **case** (a, b) **of**
alt_1
$\cdots$
alt_m
$r_0 \rightarrow r_0$

where each of the alternatives alt_i has a form that depends on the arity of the constructor C_i:

$(C_i, C_i) \rightarrow$ `Eq`	for nullary C_i
$(C_i a_1, C_i b_1) \rightarrow a_1$ `.<=>` b_1	for unary C_i
$(C_i a_1 \cdots a_{k_i}, C_i b_1 \cdots b_{k_i}) \rightarrow$ $(a_1, \cdots, a_{k_i})$ `.<=>` $(b_1, \cdots, b_{k_i})$	for C_i with arity $k_i \geq 2$

9.2 Derived Instances for Enum

The `Enum` class can be derived for algebraic datatypes that have only nullary constructors. It provides conversion from and to `Int` values, successor and predecessor functions and the operations *enumFrom*, *enumFromTo*, *enumFromThen* and *enumFromThenTo* to construct arithmetic sequences. `Enum` is a subclass of `Ord` and hence of `Eq`.

In addition to the above, derived instances also provide the operation *hashCode* required for instances of `Eq`. In derived instances, *hashCode* is always the same as *ord* and not shown in the translations.

In all derived instances, the following holds for the successor and predecessor functions:

Translation:

```
succ e = from (ord e + 1)
pred e = from (ord e - 1)
```

This implies that the successor of the last enumeration value as well as the predecessor of the first enumeration value are undefined.

Difference to Haskell 98/2010: The functions *toEnum* and *fromEnum* are know as *from* and *ord* in Frege. Aliases are provided for compatibility.

A trivial type can be an instance of `Enum`.

Translation: Let T be a trivial type:

data $T = C$
  Then
derive `Enum` T
  is equivalent to:
instance `Enum` T **where**
  ord C = 0; from 0 = C; succ _ = undefined; pred _ = undefined;
  enumFrom _ = [C]; enumFromTo _ _ = [C];
  enumFromThen _ _ = [C]; enumFromThenTo _ _ _ = [C];

Note that predecessor and successor are undefined, and all arithmetic sequences result in a list with just one element, C.

Product types with arity $k > 0$ cannot be derived instances of `Enum`. It remains to show the translation for those sum types that can be instances of `Enum`.

Translation: Let S be a sum type with m ($m \geq 2$) nullary constructors C_1, $\cdots$, C_{m-1}:

data $S = C_1 | \cdots | C_m$
  Then
derive `Enum` S
  is equivalent to:
instance `Enum` S **where**
  ord e = **case** e **of**
    $C_1 \rightarrow 0$
    $\cdots$
    $C_m \rightarrow m - 1$
  from i = **case** i **of**
    $0 \rightarrow C_1$
    $\cdots$
    $m - 1 \rightarrow C_m$
  enumFromTo a b = **if** $a \leq b$ **then** a:enumFromTo (succ a) b **else** []
  enumFrom a = enumFromTo a C_m
  enumFromThen a b = enumFromThenTo a b (**if** $a \leq b$ **then** C_m **else** C_1)
  enumFromThenTo a b c = map from
                  (Int.enumFromThenTo (ord a) (ord b) (ord c))

Note that the construct $m - 1$ will be substituted by the appropriate integer constant. The application (S`.from` i) is undefined for ($i < 0$) or ($i \geq m$). For all C_i it is the case that S`.from` C_i`.ord == ` C_i

9.3 Derived instances for Bounded

This type class defines two per type constants *minBound* and *maxBound* and can be derived for enumeration types.

Translation: Let S be a sum type with m ($m \geq 2$) nullary constructors:

data $S = C_1 | \cdots | C_m$
Then
derive `Bounded` S
is equivalent to:
instance `Bounded` S **where**
minBound $= C_1$
maxBound $= C_m$

9.4 Derived instances for Show

The type class `Show` is for types whose values can be represented as character strings. It can be derived for any algebraic data type.

Translation: Let S be a n-ary ($n \geq 0$) type constructor for a type with m ($m \geq 1$) data constructors $C_1, \cdots, C_m$ and arities $k_1, \cdots, k_m$:

data $S\ u_1 \cdots u_n = C_1\ ct_{1_1} \cdots ct_{k_1} \mid \cdots \mid C_m\ ct_{m_1} \cdots ct_{k_m}$
Then
derive `Show` $(S\ t_1 \cdots t_n)$
is equivalent to:
instance `Show` $(Show\ t_1, \cdots, Show\ t_n)$ => $(S\ t_1 \cdots t_n)$ **where**
`show v =` **case** `v` **of**
$C_1 a_1 \cdots a_{k_1} \rightarrow$ `"`C_i`" ++ " "`
`++` a_1`.showsub ++` $\cdots$ `" " ++` a_k `.showsub`
$\cdots$
$C_m a_1 \cdots a_{k_m} \rightarrow \cdots$
`showsub` C_i `= "`C_i`"` for each i where $k_i = 0$
`showsub` $C_i a_1 \cdots a_{k_i}$ `=` for each i where $k_i > 0$
`"(" ++ show (` $C_i a_1 \cdots a_{k_i}$ `) ++ ")"`

The derived `show` functions create a textual representation of a value that will be syntactically reminiscent of a constructor application if the `Show` instances of the subcomponents behave likewise. The `showsub` function shows the value enclosed in parenthesis if it is more complex than just a nullary constructor.

The translation above is equally valid for product and sum types. Types that enjoy special syntactic support (list types, tuples, and the unit type) have also special `Show`

instances whose translation is omitted for brevity. Suffice it to say that these instances will reproduce the familiar textual representations, i.e. the expression `show (1,2)` will produce `"(1, 2)"` and not `"(,) 1 2"`.

Bibliography

[1] Gottlob Frege *Funktion und Begriff* Vortrag, gehalten in der Sitzung vom 9.1.1891 der Jenaischen Gesellschaft fr Medizin und Naturwissenschaft

[2] Simon Peyton Jones, John Hughes et al. *Report on the Programming Language Haskell 98*

[3] Simon Marlow (editor) *Haskell 2010 Language Report*

[4] Simon Peyton Jones *Practical type inference for arbitrary-rank types*

[5] Simon Peyton Jones, et al. *Let Should Not Be Generalised*

[6] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification Third Edition*

[7] Simon Peyton Jones. *The Implementation of Functional Programming Languages*

[8] Martin Erwig, Simon Peyton Jones. *Pattern Guards and Transformational Patterns*

[9] *Java 2 Platform API Specification*

[10] John Launchbury, Simon Peyton Jones *Lazy Functional State Threads*

Index