Reimplementing Data Field Haskell
Omimplementering av Data Field Haskell

Jesper Simos

Department of Computer Science and Electronics
Mälardalen University

Supervisor and Examiner: Prof. Björn Lisper

April 17, 2007

Abstract

Indexed data structures, such as arrays and matrices, can be found in many programming languages. The data field model is a semantical framework which seeks to capture the essence of indexed data structures and make them more generalised. The first implementation of the data field model, Data Field Haskell, was done in Haskell 1.3 as an extension to an existing Haskell compiler. This compiler, dfhc, was ported to a newer compiler version, dfhc98, and updated to be Haskell98 compatible. Data Fields enables a collection-oriented programming style on Haskell.

In this thesis we present a new implementation of the data field model in Haskell. This implementation is not built as a extension to an existing compiler. The presented solution is comprised of a library and preprocessor that offers functionality equivalent to that of the old dfhc98 compiler. The proposed implementation is small, portable and modular, leading to a solution that is easier to maintain and extend as need arises.

Referat

Indexerade datastrukturer, som arrayer och matriser, existerar i många programmeringsspråk. Datafältsmodellen är ett semantiskt ramverk som försöker generalisera indexerade datastrukturer. Första implementationen av datafältsmodellen, Data Field Haskell, gjordes i Haskell 1.3 som en utökning av en befintlig Haskell kompilator. Denna kompilator, dfhc, blev senare porterad till en nyare version, dfhc98, och uppdaterad till att vara Haskell98 kompatibel. Datafält medgör s.k. ”collection-oriented programming“ i Haskell.

I detta examensarbete presenterar vi en ny implementering av datafältsmodellen i Haskell. Denna implementering bygger inte på en utökad kompilator utan består av ett bibliotek och en preprocessor vars funktioner är ekvivalenta med de funktioner dfhc98 erbjuder. Den föreslagna nya implementationen är kompakt, portabel och modulär, vilket ger en lösning som borde vara enklare att underhålla och utöka vid behov.

Preface

Everything has beauty, but not everyone sees it.[1]

This has thesis has been an interesting and exciting endeavour. To see the solution grow from an completely blank virtual[2] page to a full library. Much appreciation and thanks goes to my supervisor and examiner Björn Lisper for providing the underlying model of Data Field Haskell and for suggesting this area in which I now have done two theses. I'm also thankful for the help and pointers that my supervisor provided during the course of this work. Since I needed to know the details of Data Field Haskell and the interface provided by the old implementation, I studied the two theses that had been done regarding the implementation. For this I would also like to thank J. Holmerin, the implementor of the first Data Field Haskell compiler, and J.A.Sjögren, who ported the first compiler to make it Haskell98 compliant. All actual work was done on a Linux platform, so I'll give my thanks to the whole Linux community as well.

Finally I would like to express my deepest gratitude towards my wonderful parents, for always supporting and helping me in every project that I undertake.

Jesper Simos
Arboga
April, 2007

[1] Accredited to Confucius (551 B.C. - 479 B.C.)

[2] Even virtual pages feel more tangible than simply stating that one has x lines of code on the monitor.

Contents

Chapter 1

Introduction

The ability to handle a set of items in an uniform manner enhances the readability of code and can often make that same code more compact. When one can abstract away traversal details, manipulating a group of items becomes a very easy task. Languages such as *APL* [8], *Fortran 90* [17], *Sisal* [9], *NESL* [7] are able to operate directly on collections. This style of programming is often called *collection-oriented programming* [19]. *Higher-order functional languages* also have collection-oriented functions, but these are often restricted to operations on lists.

To exemplify the difference between a collection-oriented approach and that of an imperative one, we present a very small example in pseudocode. The purpose of this example is to give a more concrete demonstration of the benefits of collection-oriented programming. Assume that we want to apply a function f on each element in a list l. An imperative solution would need to traverse the list element by element, using some kind of loop-construct, and then applying f on the current element. An in-place update(destructive) solution would then look something like:

```
index:=0
while (index < length of list)
     do
          l[index] := f(l[index])
          index := index + 1
     od
return l
```

The problem is easy to code but the majority of the code is actually related to traversal details. The collection-oriented solution is much more compact and as such much easier to comprehend:

```
map f l
```

This line of code achieves the same effect as the imperative one, i.e it is semantically identical to the imperative solution. Here f is applied to all elements in l and due to the collection-oriented nature traversal details are handled transparently. In essence, the code looks exactly like we would expect from the problem definition, which was a function f applied to all elements in a list l.

Other benefits of the collection-oriented paradigm is that it offers a convenient model for parallel programming. Given the explosion of multicore processors, even in the consumer markets, demand of better utilisation of equipment and quality of code, it is vital that models and tools exists to meet the challenge potential programmers will ultimately face.

The collection we have been using in our example, the list, is an indexed data structure. Indexed data structures are found in many programming languages and are important in the field of high performance computing. The purpose of the data field model [16] is to provide a semantical framework for very general indexing structures. The implementation of the data field model in the higher-order, purely functional language Haskell, Data Field Haskell [1], was done as an extension of an existing Haskell-compiler, NHC [5].

In this thesis we present a new and reworked implementation of Data Field Haskell. The implementation is platform-independent, written in Haskell98 [11] (current standard) with extensions and transformed from a compiler to a library with preprocessor. The goal has been to create a compact and highly portable Data Field Haskell version that will be easier to maintain and update in the future.

1.1 Background

The first implementation of the data field model was done by Jonas Holmerin [12]. It was based on the NHC13 compiler, which supported Haskell 1.3. A port to the current standard, Haskell98, was done in 2001 by Andreas Sjögren [20]. The source for the port was the old dfhc compiler and the target was the newer NHC98, which resulted in the dfhc98-1.0beta compiler. After the completion of dfhc98-1.0beta, the project was dormant for several years. In 2006 an investigation was made by the author [18] to see if dfhc98-1.0beta could be updated to run on comtemporary systems and also to investigate possibilities to make a more portable solution. The findings of this investigation were that too much had changed (Haskell-compilers, libraries and related tools) for a smooth updating of the dfhc98-1.0beta compiler. The second part of the investigation, however, led to a possible solution for a more portable implementation.

The purpose of this thesis is to implement the suggested solution in [18] and thus get a new and working Data Field Haskell implementation with equivalent features to the former dfhc98-1.0beta compiler.

1.2 Delimitations

The implementation of the Data Field Haskell library went quite well without any major problems. There were some issues that hindered the construction of a full and working preprocessor. The first problem that occurred was during the syntax translation part of the preprocessor. One syntax conversion required a transformation from one type to another type. Unfortunately some “exotic” patterns, that needed to be translated into another format, either required much effort for it to work or it simply could not be done(no translation possible). The second and most important delimitation was the timelimit. Producing code to correctly calculate data field bounds from the large abstract syntax tree proved to take much longer than expected.

1.3 Expected & Actual results

The expected result from this thesis was to have a compact and portable Data Field Haskell library and preprocessor, that would have features equivalent to that of the Data Field Haskell compiler dfhc98. The features the new implementation needed was:

1. Library to implement all data field related functions found in dfhc98
2. Preprocessor to handle and translate Data Field Haskell specific syntax constructs.
3. Automatically deriving the bounds from Data Field Haskell specific syntax constructs.

Due to the problems discussed in *Delimitations*, the bounds derivation part is not yet functional and the preprocessor might be more stricter about input than necessary. Due to the recursive nature of the solution of the bounds derivation part, all rules need to be implemented for the derivation to work accurately. The implementation of bounds derivation is not complete yet so, in essence, deriving bounds does not work yet on any constructs. To summarize the actual result, point *1* and *2* are completed but point *3* requires more work.

The contribution of this thesis is a new implementation of Data Field Haskell, comprising of a library and a preprocessor. The implementation is compact, portable, documented and should hopefully be easy to maintain and extend, should such a need arise.

Chapter 2

Haskell

To be able to understand Data Field Haskell, we need to turn our attention towards the programming language that we both extend and use to implement our Data Fields.

Haskell is a lazy and purely functional general purpose programming language with a lot of interesting features. Haskell is also called a declarative language, since programming is mainly done by specifying desired actions but not the order of the actions. It is quite different from other programming languages, this is due to the nature of the design and to the fact that much research is performed on the Haskell language. The first version of the language was defined in 1990. The current standard is known as Haskell98, but there is ongoing efforts to produce a new standard. This new proposed standard is informally known as Haskell' or “Haskell Prime”.

Haskell brings benefits such as shorter, clearer and more maintainable code. The theoretical foundation of Haskell makes it easier to reason about programs. As an example, there are no side-effects in Haskell therefore one does not need to worry about some hidden state affecting computations. Functions in Haskell are equivalent to mathematical functions, that is if a function is called with a value a the result will always be b. It doesn't matter how many times the function is called, that result will always be the same. To demonstrate the difference, consider a function `getchar()` that reads input from a user. Here each call to `getchar()` might differ depending on what the user inputs.

Haskell is a very flexible language, where many constructs can be user defined. This property has led to Haskell being a suitable language for interpreters and compilers. In fact one of the biggest Haskell compiler is written in Haskell itself. It also has automatic memory management, which relieves the programmer of the burden to handle memory allocation and deallocation.

Drawbacks do exist like in Haskell like in any other programming language. Laziness makes it harder to reason about the performance of written code. Haskell is not so common outside the academic sphere. Finally some find Haskell to complex to understand.

The following sections will present features found in Haskell, although it should not be seen as a substitute for a good book. A good start to learn Haskell is found in [13].

Naturally we can not give a comprehensive coverage of all things in Haskell, therefore our presentation will contain only a select set of features found in

Haskell. We first explain some general concepts found in Haskell and then we introduce executable Haskell snippets with explanations. This will provide an insight in how Haskell is used, it is assumed though that the reader is familiar with some programming concepts.

2.1 Pure

Destructive updating of values are not allowed in a pure language. In other words there are no side-effects. Absence of side-effects also make it easier to reason about and prove properties of a program. The reason for this is that all function values only depend or their input and have no hidden state.

2.2 Lazy evaluation

Evaluation of expressions only happens when the result is needed. Lazy evaluation enables infinite data structures, which can not be expressed easily in languages that uses an eager evaluation strategy. Infinite data structures can sometimes provide very compact solutions for problems that can be modelled as streams.

2.3 Strongly typed

In Haskell all types are checked at compile-time. Conversions between different types are in general not allowed. In those cases conversions are allowed, it must be done by explicitly calling the proper conversion function.

2.4 Polymorphism

This property allows defined functions to work on several different types. Polymorphic data structures, such as a list of any numeric type, are also possible.

2.5 Type inference

With type inference the type of an expression can be automatically inferred by the compiler. The type inferred will be the most general type an expression can have. This can alleviate the programmer from having to make type declarations, although this is often done as a means to describe and document functions. Documentation generators also make use of type declarations when automatically generating documentation from source code.

2.6 Type classes

One of Haskell's unique features is that of type classes. The purpose of type classes is to be able to restrict polymorphic types. Type classes consists of two parts:

1. Class declaration with operations: This defines the class and operations that is part of the class. The operations consists of type signatures. Sometimes these type signatures are followed by implementations which are called *default methods*.

2. Instance declaration with methods: In order for a type to be a member of a class, an instance declaration has to be made where the operations of the class are implemented, called *methods*, for that particular type.

Using type classes it is possible to partition the polymorphic types into sets, types that are members of a class and types that are not part of the class. Type classes also presents a convenient way of handling many different types using one single function name, so called *ad-hoc polymorphism*. Depending of the type of the parameter the resulting call of the function will be that which is specified by the instance declaration.

We now present a concrete example of the two parts that form a type class:

```
class Check a where   -- (1)
  (===) :: (Eq a) => a -> a -> Bool
```

The class declaration with operation(s) in `(1)` and

```
instance Check String where   -- (2)
   str1 === str2 = str1 == str2
```

the instance declaration with method definition `(2)`. The class declarations specifies which operations a data type needs to implement to be part of the class. The instance declaration makes a data type part of the given class. In our example we simply have a class which compares items. The class restriction `(Eq a) =>` specifies that the type `a` must be a member in class `Eq`, in essence meaning that the operator `==` must be defined on that data type. In the instance declaration, we can see that we are using exactly `==` to check for equivalence, thus the reason for the constraint. Also by making another instance declaration for some other type we get the overloading effects, since comparison can be done on different types with the same functions `===`.

2.7 Currying

This technique refers to the situation where, instead of taking multiple arguments, a function is converted to a function that takes one argument but returns a new function as a result. This new function can then be applied to another argument giving another function until all arguments have been used and the final result calculated.

2.8 Algebraic data types

These data types wrap data from other data types in its constructors. Unlike ordinary data types the algebraic data type can not be executed only unwrapped. Pattern matching is used to traverse or deconstruct these structures. Since Haskell is polymorphic, it also applies to the algebraic data types. A very useful construct, algebraic data types can look like example `(3)`:

```
data Enum = One |Two |Three   -- (3)
```

in the case with nullary constructors. Algebraic data types with constructors that can take several polymorphic arguments might look like example `(4)`:

```
data Container a b = OneA a
                   | OneB b   -- (4)
                   | BothAB a b
```

One could also choose to have an algebraic data type where the types are fully specified as in `(5)`:

```
data Thing = ThingConstructor Int   -- (5)
```

To use these data types the constructor is used together with possible arguments:

```
c = BothAB 5 "five"   -- (6)
```

the expression type of example `(6)` is `c :: Container Integer [Char]`.

2.9 Anonymous functions

Using λ-abstraction one can define anonymous functions. These are used in instances where small "onetime" functions are needed, such as arguments to higher-order functions. They are constructed in Haskell using the form of `\pat_0 ... pat_n -> expr`, as in:

```
\x y -> sqrt (x^2 + y^2)   -- (7)
```

`(7)` is an anonymous function that gives the length of the hypotenuse in a right triangle. Binding this equation to a name results in a "normal" function.

2.10 Binding

There are a couple of ways of binding expressions to a name. The equal sign, `=`, can be used. There are also the `let` and `where` bindings, as showed in `(8)`. There is one difference between `let` and `where`, it is that `where` can only be used at the top level of a function definition. We construct the function in the previous example `(7)`, this time using local bindings.

```
g x y = let xsq = x^2
            ysq = y^2
            sqsum = xsq + ysq
        in sqrt sqsum
                                  -- (8)
h x y = sqrt sqsum
         where sqsum = xsq + ysq
```

```
xsq = x^2
ysq = y^2
```

Note that although we have reused the name `xsq`,`ysq` and `sqsum` there is no nameclash due to the effect of the bindings.

2.11 Comments

There are two ways to comment code in Haskell. The first type is a lineoriented comment which starts with `--`. This states that the comment stretches from the mark until the end of the line. The second format is the pair formed by `{-` and `-}`. Here everything between these markers are considered comments.

```
--This is a linecomment
 {-
   These are more comments with    -- (9)
   {- some nesting -} demonstrated
 -}
```

Comments can, as shown in `(9)`, be nested.

2.12 Monads

To accommodate side-effects and have the ability to order calculations, monads were introduced. Input and output is done in the `IO`-monad. Calculations that keep track of state also need to use monads. There is a special notation coupled with monads, it's called the *do*-notation. This notation is just syntactic sugar[1] for the operations in the monad class.

2.13 Do expression

Do expressions provide a nicer syntax for monadic programming. An informal description of monadic programming is that monads are used to structure computations that need to happen in a certain order or when side-effects are wanted. As an example, input and output operations in Haskell is done in the `IO`-monad. The term monad comes from a branch of mathematics called category theory, however there is nothing different between monads or monadic programming and Haskell. They are, simply put, constructs with rules that determine the function of a monad. Do expressions provide an alternative way of programming with monads. Statements in a do-block are executed in sequential order. This is also the only place where one can find statements in Haskell. Example `(10)` shows the difference between the monadic style and do:

```
monadprint = print "Input:" >> getLine >>=
                \str -> print str
                                        -- (10)
```

[1] Easier syntax that doesn't provide anything extra.

```
doprint = do
            print "Input:"
            str <- getLine
            print str
```

Both functions do the same thing. After printing “Input:” on the screen, it waits for some input which it then echoes back to the screen.

2.14 Exceptions

There are two ways to raise an exception. The first one is to explicitly call the function `error`, with an informative string. The other is to use the constant `undefined` which also raises an exception. With descriptive error strings, the time finding flaws in the code can be reduced. One thing to note is that general exceptions are hard to integrate into pure and lazy languages.

```
error "Something is wrong!"   -- (11)
errorlist = [1, undefined]
```

Evaluating `errorlist` in example `(11)` results in an exception being raised.

2.15 Functions

The foundation of functional programming languages, functions can be found almost everywhere in Haskell code. Functions can be of specific types or be made polymorphic. In Haskell, function definitions are expressed as an equation or a set of equations.

```
addint :: Int -> Int -> Int
addint x y = x + y

addnum :: (Num a) => a -> a -> a   -- (12)
addnum x y = x + y

ident :: forall a. a -> a
ident x = x
```

The line just above every function definition in `(12)` is called a type signature, these are discussed in 2.30. The first function in example `(12)` demonstrates a function with two integer[2] arguments that returns an integer. The second function is a polymorphic function with a class constraint, class constraints are explained in point 2.6, that can add types that are members of the `Num` class and return a value of the same type as the arguments. It is easy to see that `addnum` is a generalisation if `addint`. The identity function `ident` is fully

[2] Haskell have two integer values, `Int` and `Integer`. The difference is that the size of the former is limited to the word size on the given machine, whereas the size of the latter is unrestricted.

polymorphic as it has no class constraints. The `ident` function can take, as argument, any type and return it. If no type signature is given, the compiler will infer the most general type of a function. This means that removing the `addint :: Int -> Int -> Int` line above `addint` will result in `addint` becoming equal to `addnum`. They will have the same function type signature. Important to note is that, due to Haskell's strong typing, the types of the arguments must match that of the signature. Calling `addint` with a type other than `Int` will fail. Removing the type signature, as mentioned earlier, will allow `addint` to work with any numeric type. But in this case there is a class constraint that must be fulfilled.

2.16 Guards

Guards are used together with patterns. They are used to specify further conditions that have to be met in order for the right-hand side of the equation to be evaluated.

```
signstring 0 = "Zero"
signstring n | n > 0 = "Pos"   -- (13)
             | otherwise = "Neg"
```

In example `(13)`, the `signstring` function takes a number and returns a string, which gives the sign of the number. The guard consists of `|` *`predicate(s)`* or `| otherwise`. If the predicates succeeds, the right-hand side of the equation is evaluated. If it fails the next predicate is tested. `otherwise` always succeeds and can be seen as a default case.

2.17 Higher-order functions

These functions work with other functions as arguments or results. Higher-order functions are useful, as they allow one to capture common patterns in one single function. Consider the `map` function. It takes a function and a list as arguments. The result is a new list, where each element is formed by the application of the function to the same element in the old list. In other words, the function is applied to every element in the old list and this new list is the result.

```
map (\x -> 1 + x) [1,2,3,4]   -- (14)
```

Example `(14)` also shows the use of anonymous functions as arguments. The result of this expression is the new list `[2,3,4,5]`. We also give one example of a higher-order function that returns a function as a result.

```
newfunc x = \y -> y^x   -- (15)
```

The result of the function in `(15)` is a new function with one argument, whose result is argument y raised to the power of x. `newfunc 2`, for instance, gives a function that computes the squares of its arguments.

2.18 Infinite data structures

Due to lazy evaluation, where evaluation of an expression only happens when it is needed, these kinds of structures are possible. They offer, as mentioned earlier, a convenient way to solve certain problems.

```
inflist = [1..]                  -- (16)
squaredlist = map (\x -> x^2) inflist
```

In (16), `inflist` is an infinite list of positive numbers. `squaredlist` is generated by using the higher-order function `map` with the squaring function on `inflist` to get an infinite list of squared natural numbers. A more useful example is that of the infinite list of primes. This will be demonstrated in point *List comprehension*.

2.19 Layout

Haskell has two styles to delimit code blocks, layout sensitive and layout insensitive style. Layout insensitive style adds the use of semicolons and braces. Layout sensitive style is the style that is used normally when programming. Having two styles means that Haskell code can be both easily generated by other programs and programmed. We will only informally describe the details of the layout sensitive style. The first character of each equation must line up. If an equation spans several lines, the following lines must be to the right of the first character. The first character following some keywords such as `let`, `where` and others are what determines the starting column of the layout.

```
let a = ...
    b = ...
in ...
               -- (17)
let a = ...
  b = ...
in ...
```

The first `let`-expression in example (17) has a legal layout, whereas the second has not.

2.20 Lists

List are very useful data structures. All elements have to have the same type, but as many other data structures in Haskell, the list can be polymorphic. Lists can be finite or infinite. There are also many functions, such as `map`, `foldl`, `head` and `filter`, that work on lists.

```
listone = 1:2:3:4:[]
listtwo = [1,2,3,4]   -- (18)
listthree = [1..4]
aritmlist = [2,4..]
```

The first three lists in example (18) all represent the list [1,2,3,4]. The last two lists uses a special syntax to form a arithmetic sequence. Again one can see that aritmlist is an infinite list of even numbers.

2.21 List comprehensions

Another very useful construct to handle lists is list comprehensions. They mimic set comprehensions, that have the form of $\{f(x_1, ..., x_n)| \; p_1(x_1) \wedge ... \wedge p_n(x_n)\}$, and offer a compact way of building lists using a style very similar to that of set comprehensions. These construct lists where elements are taken from other lists, often together with one or more predicates. List comprehensions are explained with some list examples.

```
orglist = [1..10]
l1 = [x^2 | x <- orglist]                -- (19)
l2 = [(y,y)| y <- l1, (y ‘mod‘ 2) == 0]
```

In example (19), orglist is as we have seen earlier a finite list containing all numbers from 1 to 10. l1 constructs a new list where each element comes from orglist but is squared. l2 constructs a list of pairs from the even squares found in l1. The backquotes ‘...‘ are used to make a function work like an infix operator. We finally present an infinite list of primes using the recursive function sieve and a list comprehension in (20).

```
primes = sieve [2..] where
          sieve (h:t) = h :                    -- (20)
           sieve [y | y <- t, (y ‘mod‘ h) /= 0]
```

2.22 List reductions

Haskell offers several functions that reduce a list given a binary operator. The reason for several functions is that a list can be reduced either from left to right or from right to left. If the function used is commutative then the result will be the same regardless of choice. Non commutative functions require more care when selecting the proper function. The reduction functions in Haskell are foldl, foldr, foldl1 and foldr1. The l and r stands for left and right respectively and the 1 signifies that no initial value needs to be supplied. The functions ending with 1 must therefore be applied only to non-empty lists. All folds require finite lists in order for them to work. There are also several scan functions. They work in the same way as *folds* but with the difference that the final result is a list of intermediary results from the reduction. The scan functions are scanl, scanr, scanl1 and scanr1. The following two folds both yields the same result.

```
foldr (+) 0 [1,2,3] -- => 6
foldl (+) 0 [1,2,3] -- => 6
```

In this case the result differs since (-) is not commutative.

```
foldr (-) 0 [1,2,3] -- => 2
foldl (-) 0 [1,2,3] -- => -6
```

This time we use the same setup but with scans instead.

```
scanr (+) 0 [1,2,3] -- => [6,5,3,0]
scanl (+) 0 [1,2,3] -- => [0,1,3,6]
scanr (-) 0 [1,2,3] -- => [2,-1,3,0]
scanl (-) 0 [1,2,3] -- => [0,-1,-3,-6]
```

Note that in the `r`-variants the result of the folds matches the first element in the scans and in the `l`-variants the result of the folds matches the last element in the scans.

2.23 Modules

Haskell code is packaged in modules. Modules introduce a namespace and can be used to create an abstract data type. They have the form found in example `(21)`:

```
module "Name" where
DATATYPES     -- (21)
DECLARATIONS
ETC...
```

The first letter in a module name must be capitalized.

2.24 Operators

Operators have infix syntax. Operators can be user defined, the only requirement is that the operators do not contain any numbers, alphabetical characters or the symbol `'`.

```
(<^>) a b = sqrt (a^2 + b^2)   -- (22)
```

The operator in `(22)` calculates the hypotenuse of a right triangle. An example of the usage of the operator is `side1 <^> side2 => hypotenuse`. By using `(...)` around an operator it can be used as a normal function. Compare this with the use of backquotes, which converts a function into an operator.

2.25 Pattern matching

Using pattern matching it is easy to deconstruct data structures. Function definitions can also be made very succinct with pattern matching. Some of the earlier examples have already shown use of pattern matching. Pattern matching can work on data constructors with both user defined and predefined types.

```
match 'a' = 1
match 'b' = 2   -- (23)
match _ = 3
```

The `match` argument in `(23)` is matched first with the character 'a'. If there is a match the result is the number '1', otherwise the pattern matching continues with the second equation and so on. The underscore is known as a *wildcard*, it matches everything. The pattern `h:t` matches a list with a *head* and *tail*. Matching on a data constructor `C Int String` in a function `func` can look like `func (C num str)`. In this case, a successful match will bind the values in the data constructor to `num` and `str`.

2.26 Recursion

Often used in functional programming, but also found in other languages, it is formed by base case and recursion step. Recursion can be used to solve a problem by dividing it into smaller parts or as a way to iterate.

```
quicksort [] = []                            -- (24)
quicksort (h:t) = quicksort
                   [lower| lower <- t, lower < h]
                    ++ [h] ++
                     quicksort
                      [higher| higher <- t, higher >= h]
```

Example `(24)` demonstrates the quicksort function, where both recursion and list comprehensions are used.

2.27 Standard Haskell types & classes

Haskell provides a number of predefined type classes and types. Basic types are *booleans*, *characters*, *strings*, *lists* and *tuples*. Some of the classes are *Eq*, *Show* and *Num*. For a full list of types and classes, please refer to [11].

2.28 Standard Prelude

The standard prelude forms the basic library with functions and classes. Different compilers often might include extra libraries, but the prelude should always be the same in all implementations.

2.29 Tuples

Contrary to lists, tuples are non growable data structures supporting heterogeneous types. They are often used to pack together values of different types. They are static in the sense that a 2-tuple can not be expanded to a 3-tuple. Tuples of different dimensions have different types, that is n-tuples are distinct from m-tuples if $n \neq m$.

```
(1, "string")   -- (25)
('a', 5, 'r')
```

Example (25) demonstrates 2- and 3-tuples.

2.30 Type signature

The purpose of type signatures is to provide a way to specify which types a function can take and return. Using type signatures one can restrict a function to work with only specific types. Excluding the type signatures results in a function with the most general type, class constraints are of course taken in consideration. It also serves as a documentation of the function, as the types of the arguments and results is clearly visible.

```
func1 :: String -> String
func2 :: (Eq a, Show b) => a -> a -> b -> String  -- (26)
func3 :: (a -> b) -> a -> b
```

The first type signature in (26) is for a function whose argument is of type string and whose result is also of type string. The second signature shows some class constraints. This function has three arguments, of which the first two must have the same type and be part of class Eq and the third must be part of class Show. The result is of type string. The final signature specifies that the first argument is a function from type a to typeb. The second argument is of type a and the result of the functions is of type b. func3 is a higher-order function and in this case the purpose of it is to apply its first argument to its second argument.

Chapter 3

Extensions

The library utilises two extensions to Haskell98, currently only found in the *Glasgow Haskell Compiler* [10]. The use of these extensions are necessary in order to implement product bounds[1] and general functions that can handle them. The two extensions used are *Generalised Algebraic Data Types*(GADT) [15] and *Functional Dependencies* [14].

GADT:s are a generalisation of ordinary algebraic data types. The difference between the two data types is that, in GADT, type signatures of constructors can be given explicitly. Normally the result type of a data type has to be the same as the data type. Often this has the form of a type constructor T applied to all type parameters used. This simply means that if we have a type variable `a` we want to use in the data type, then it must also present in the data type declaration such as `data T a = ...`. If we would like to use two different types in our data type then both types need to be declared as in `data T a b = ...`. The problem we face with this approach is that we either have to expose the internals of the data type or not being able to form our product bounds in the first place. The desired solution for the representation of product bounds is a data type which has a single type variable that is "flexible". With this we mean a data type where type variables need not be declared even though they might be used. An example of what we want is to express `T a -> T b -> T (a,b)`. With standard data type constructors this would not be valid since `T a` is not compatible with `T (a,b)`. Furthermore the type variable `b` in `T (a,b)` does not even exist in `T a`. Fortunately this problem is solved with the use of GADT. In a GADT, the result type can differ from the data type, that is the result can be an application of a type constructor T to arbitrary argument types. This means that code using GADT can express `T a -> T b -> T (a,b)`, something that would be impossible with standard data types as the resulting type `T (a, b)` would have to be `T a` to be legal. To make it more concrete we show a real GADT declaration in Haskell:

```
data T a where
     T1 :: a -> T a
     T2 :: T a -> T b -> T (a,b)
```

[1]Product bounds are multidimensional bounds, found in Data Field Haskell, that are formed from simpler bounds. The concept will be further explained in 7.

Notice how the keyword `where` is used together with the keyword `data` to specify a GADT. Also note that data constructors are given something that looks very similar to type signatures.

Functional dependencies allows a programmer to specify relationships between parameters in a multiparameter class. This provides more flexibility when defining functions that must be able to handle parameters of different types, as one is given more control on how those type parameters should be handled. Without these annotations, the compiler can not always deduce the intention of the programmer even if the programmer knows what should be achieved. With functional dependencies, relationships between parameters can be expressed and enforced. To specify a functional dependency, such as type `a` determines type `b`, one uses a `| a -> b` construct in a class declaration. To present a more concrete view we give a contrived example of two identical classes, where the first one uses functional dependencies and the second one is an ordinary Haskell class. The class `C` has one single operation `f`, which takes values of type `a` and returns values of type `b`. The actual types of `a` and `b` are given by the instance declarations.

```
class C a b| a -> b where                       -- (1)
      f :: a -> b

instance C Int Bool where                       -- (2)
      f n = if n == 0 then True else False

instance C Char Bool where                      -- (3)
      f c = if c == 'a' then True else False

-- instance C Char Int where                    -- (4)
--      f c = if c == 'a' then 1 else 0
```

The reason `(4)` is commented is that the instance is illegal when using functional dependencies. This is because our declared functional dependency is specified as `a` uniquely determines `b`. In `(3)` we specified `a` as `Char` and `b` as `Bool`, this means that we can not specify another instance where `a` is `Char` and `b` is something other than `Bool`. Doing so would violate the functional dependency.

```
class C a b where                               -- (5)
      f :: a -> b

instance C Int Bool where                       -- (6)
      f n = if n == 0 then True else False

instance C Char Bool where                      -- (7)
      f c = if c == 'a' then True else False

instance C Char Int where                       -- (8)
      f c = if c == 'a' then 1 else 0
```

In this case `(8)` works since we have no functional dependency specified. The problem here is that we need to supply a type annotation to the compiler when using function `f` with type `a` as `Char`, this is so the compiler knows which of `(7)` or `(8)` was intended. This is one benefit with functional dependencies in classes, if one knows what effect is desired one can specify it as an functional dependency instead of having to supply type annotations. In this trivial case it might be a bit hard to grasp the usefulness of functional dependencies, but when dealing with more complex code they can be quite powerful and sometimes even necessary.

Chapter 4

Template Haskell

Template Haskell(TH) [6] provides a type-safe compile-time meta-programming framework for Haskell. The purpose of TH is to give a programmer the ability to manipulate Haskell code in Haskell at compile time. Functions to create, handle and translate between concrete- and abstract syntax are provided. TH also provides a less verbose abstract syntax tree that is easier to calculate on compared with the parser for Haskell98 found in the GHC library.

Besides the functions provided there are two syntactical extensions that make it easy to program with TH. These are splice- and quasi-quote notation. $(..), called a splice, evaluates the enclosed code at compile-time and replaces the $(..) with what was just evaluated. [|..|], the quasi-quote notation, converts the enclosed Haskell code into an abstract representation. This makes transformations reasonably transparent and efficient. When using this extension, invoking the ghc-specific flag `-fth` is required.

TH is useful for macro-like expansions, where one can make certain transformations on given code snippets. These transformations would then be inserted back into the code. Since the transformations and calculations are done at compile-time, this work does not need to be done during runtime. The use of these features is the main idea behind bounds deriving module found in the library.

Chapter 5

Tools

This section briefly describes the tools used for the implementation.

5.1 Haddock

Haddock [2] is a tool for automatically generating documentation from annotated source code. The resulting documentation is a fully hyperlinked document. This tool was used on the library to get a nice looking HTML documentation.

5.2 Happy

Happy [3] is a parser generator for Haskell. It is similar to "yacc", that produces code for the generated parser in C. Happy takes a file containing an annotated BNF specification of a grammar. It then creates a Haskell module containing the parser for the given input grammar.

Happy is a bottom-up parser in contrast to the monadic parser combinators used in dfhc98. Parser combinators take another approach when parsing code, instead of specifying a grammar one builds the whole parser by combining smaller parsers. For example on can have a parser *pc* that only recognizes characters and another parser *pn* that can recognize numbers, then create a parser that can recognize valid identifiers by combining *pc* and *pn*.

5.3 Glasgow Haskell Compiler

The Glasgow Haskell Compiler(GHC) [10] is an optimising compiler for Haskell done in Haskell. It comes with an interactive environment, *ghci*, and extensive libraries. GHC fully implements Haskell 98 and it also features a number of extensions. It works on several platforms and is considered, together with *Hugs*[1] [4], de facto standard for the language. GHC features a liberal license and source code for the compiler is available.

[1] An interactive Haskell interpreter.

Chapter 6

The Data Field Model

The model presents a simple and elegant view of indexed data structures. The idea is to model indexed data structures as a partial function with explicit information of the domain. In the data field model, the explicit information of the domain is called *bounds*. The data field thus consists of a pair (f, b), the function f and the bounds b. Since we are dealing with partial functions, they do not necessarily have to be total. So in order for these functions to have a conventional function type, a specific *error value* $*$ is introduced. The algebraic properties of $*$ is similar to $\perp$ and is "returned" when a data field is called with an argument outside its domain. This model enables a collection-oriented programming style because most types of collection-oriented operations can be defined as higher order functions operating on partial functions.

Bounds are important in the data field model. They provide an abstract set that defines where the data field is valid, that is where the data field does not result in $*$. The basic defined and required operations for bounds are:

1. Each bound has an interpretation as a predicate or set.
2. A predicate classifying each bound as either *finite* or *infinite*. This depends on whether its set is surely finite or possibly infinite.
3. For every bound b defining a finite set, a function *size(b)* that yields the size of the set and *enum(b)* that is a function enumerating its elements.
4. Binary operations $\sqcap$("intersection") and $\sqcup$("union") on bounds.
5. The bounds *all* and *nothing* representing the universal and empty set, respectively.

These operations support calculations on partial functions without revealing the inner structure of bounds. Beside these operations there are several more defined for both data fields and bounds. They will be discussed in the next section that details Data Field Haskell, since these are more related to the implementation of Data Field Haskell.

The model also defines *φ-abstraction*. It is a syntax for convenient definition of data fields similar to λ-abstraction for functions. As an example, $\varphi x.t$ is a data field $(\lambda x.t, b)$ where b provides an upper approximation to the domain of $\lambda x.t$. φ-abstraction provides a formal semantics for collection-oriented operations where the bound of the result is implicitly given by the bounds of the operands.

The next section will describe the implementation of the model, the rules related to φ-*abstraction* and automatic derivation of bounds.

For a more detailed view of the data field model, please refer to [16].

Chapter 7

Data Field Haskell

This section will detail and explain functions, rules and building blocks found in the implementation. The list is not meant to be complete, as implementation specific technicalities will not be addressed. Functions found in the library will be briefly described with their function types.

Before we proceed with the details we first present an overview of Data Field Haskell. The purpose of Data Field Haskell is to concretize the Data Field model. This means that the concepts, rules and operations found in the Data Field model form the core of Data Field Haskell. This core has then been enhanced with further functions and constructs that enable users to work efficiently with data fields. There is a direct correspondence between the data fields found in Data Field Haskell and those in the Data Field model. This is also true for bounds, where the core functions that work on bounds are equivalent to the operations defined on bounds in the model. Another example is the φ-abstraction, that in Data Field Haskell is called `foreach`. The reason for the difference in name is related to the fact that it would be quite difficult to type φ on a normal keyboard when coding.

The bounds are quite central in the library so we will give a short presentation of them. Bounds can be simple bounds or product bounds. Product bounds are formed from several simple bounds. For instance, creating a product bound from two simple bounds yield a two dimensional product bound. The benefit of this becomes apparent when one realizes that the two simple bounds that form the product bound can be different. The bound in the first dimension could be finite, whereas the bound in the second dimension could be infinite. This makes product bounds extremely flexible entities. In this sense infinite bounds are quite similar to 2.18 where we presented infinite data structures and finite bounds would then be similar to a list with finite elements. It is important to note that this description is done merely to help understanding bounds as they are fundamentally different from lists.

Simple bounds are divided into the five categories: *dense*, *sparse*, *predicate*, *universe* and *empty*. Dense bounds are contiguous, in other words given two points the bound is defined on those two points and everyting between them. For now it will suffice to say that the points are indexes, the details of these indexes will be given shortly. Sparse bounds are formed from a set of points. In contrast to dense bounds, sparse bounds can be both contiguous and non contiguous. From a performance view it is better to use dense bounds if one knows that the points will be contiguous. Predicate bounds are infinite bounds formed from a predicate function, they only tell if a point is within the given bound or not. The bounds *universe* and *empty* are special as they signify that a bound is defined everywhere or nowhere.

We now turn to the points we mentioned when we described the bounds. As said earlier these points are indexes, so from now on we will only talk about indexes. These indexes are used when indexing a data field. This is similar to indexing any indexed data structure such as an array. Data fields can be indexed with any type given following constraints on the type:

- Type must be a member of class `Ix`. This ensures that normal indexing operations are defined for the used type.
- Type must be a member of class `Show`. Bounds are members of `Show` to make them easier to program with (visual inspection of bounds), so types used for indexing must also support this.
- Type must be a member of class `Pord`. Defines efficient operations for partial orders.
- Type must be a member of class `Bounds`. This constraint is found primarily on functions that handle data fields. The purpose of this contraint is to make sure that the index used on the data field is compatible with the bounds in the data field.

We also present the class `DeepSeq` that can appear as a class constraint. This class provides operations for deep evaluation. It is used to force an evaluation of arguments that would be unevaluated otherwise, due to the lazy nature of Haskell. In other words, we can turn a lazy function into a strict function using `DeepSeq`. The arguments of this new function would then be fully evaluated before being used by the function itself.

Finally we have the `datafield` function which is the Data Field Haskell equivalent of data fields found in the model. It takes a function and one of the earlier mentioned bounds to form the data field.

A note regarding the code examples in the following sections, all output that starts with `<Bounds>:` is presented for pedagogical reasons. For instance, when extracting a bound from a data field the related bound is returned. However if we just specify that a bound is returned without presenting more details, it will be hard to understand how primitives are affected by different operations.

7.1 Datafields

This section presents the main operations on data fields. Together with the operations that deal with bounds, these form the bulk of Data Field Haskell. Note that the data type `Dfval a` is used to express the fact that the result of a data field can be out of bounds. Further information of `Dfval a` is found in 7.3.

datafield Creates the data field from a function and bounds. The function used must be a `Dfval`-value function. Functions to convert normal functions to `Dfval`-value functions are provided. The type is

```
datafield :: (Ix a, Show a, Pord a, Bounds c a) =>
             (a -> Dfval b) -> c -> Datafield a b c
```

A oneline example constructing a data field:

```
df = datafield (dfvalfun (\x-> x)) (0 <:> 10)
```

assoctoDf Creates a data field from an associative list and has type

```
assoctoDf :: Bounds (Bound a) a =>
             [(a, b)] -> Datafield a b (Bound a)
```

Constructing a data field from list:

```
tdf = assoctoDf [(1,10),(2,20),(3,30),(4,40),(5,50),(6,60)]
```

dftoAssoc Is the opposite of the above function. Creates an associative list from a data field. It has type

```
dftoAssoc :: forall b a c. (Bounds c a, DeepSeq a,
                                        DeepSeq b) =>
                            Datafield a b c ->
                            [(a, Dfval b)]
```

Another oneline example:

```
assoclist = dftoAssoc tdf
```

This results in `assoclist` getting bound to the list

```
[(1,Dfval 10),(2,Dfval 20),(3,Dfval 30),
 (4,Dfval 40),(5,Dfval 50),(6,Dfval 60)]
```

(!) Is the data field application operator. It applies a data field to an index. The type is

```
(!) :: forall a b c d . (Bounds c a, DeepSeq a) =>
                         Datafield a b c -> a -> Dfval b
```

Applying the data field `df` to two values, one in bounds and the other one outside the defined bounds:

```
df!2 -- => Dfval 2
df!50 -- => OutOfBounds
```

Note that the operator is equivalent to the `(!)` in Haskell for indexing arrays.

(<\>) The restriction operator restricts a given data field with a specified bound. The type is

```
(<\>) :: (Ix a, Show a, Pord a, Bounds c a) =>
          Datafield a b c -> c -> Datafield a b c
```

Restricting `df` with a predicate bound, in this case the bound that is created by `evenbound = predicate even`.

```
bounds (df <\> evenbound)
-- => <Bounds>: Sparse [0,2,4,6,8,10]
```

bounds Extracts the bounds part of a given data field. Has type

```
bounds :: (Ix a, Show a, Pord a, Bounds c a) =>
           Datafield a b c -> c
```

Extracting the bounds from `df`:

```
bounds df -- => <Bounds>: Dense 0 to 10
```

translate Translates a given data field with respect to a given value. The type of `translate` is

```
translate :: forall c b a . (TransBound c a, Bounds c a) =>
             a -> Datafield a b c -> Datafield a b c
```

Translating the data field `df`:

```
bounds (translate 5 df) -- => <Bounds>: Dense 5 to 15
```

domain This function yields the domain of the data field. The type is

```
domain :: (Ix a, Show a, Pord a, Bounds c a, DeepSeq a) =>
           Datafield a b c -> [a]
```

Using it on `df` we get:

```
domain df -- => [0,1,2,3,4,5,6,7,8,9,10]
```

tab One of several tabulator functions. This one tabulates in a lazy fashion and has type

```
tab :: (DeepSeq a, Bounds c a) =>
        Datafield a b c -> Datafield a b c
```

stricttab Tabulates a data field and evaluates the cell to *weak head normal form*. The type is

```
stricttab :: (DeepSeq a, Bounds c a) =>
              Datafield a b c -> Datafield a b c
```

hstricttab Tabulates a data field in a hyperstrict fashion, this evaluates to the inner most constructor. Has type

```
hstricttab :: (DeepSeq a, DeepSeq b, Bounds c a) =>
               Datafield a b c -> Datafield a b c
```

Beside these functions there are also several folds and scans for data fields provided. They are similar to the ones presented in 2.22, but works on data fields instead. We only present one of each family:

foldlDf A left fold for data fields. Works similar to the Haskell `foldl`. The type is

```
foldlDf :: (Bounds c a, DeepSeq a) =>
           (r -> a2 -> r) -> Dfval r ->
            Datafield a a2 c -> Dfval r
```

Using this fold on `df` we get:

```
foldlDf (+) (dfval 0) df -- => Dfval 55
```

scanlDf A left scan for data fields. Works similar to the Haskell `scanl`. The type is

```
scanlDf :: (Bounds c a, DeepSeq a) =>
           (r -> a2 -> r) -> Dfval r ->
            Datafield a a2 c -> [Dfval r]
```

The scanl for data fields applied to `df` yields:

```
scanlDf (+) (dfval 0) df
-- => [Dfval 0, Dfval 0, Dfval 1, Dfval 3, Dfval 6,
--     Dfval 10, Dfval 15, Dfval 21, Dfval 28, Dfval 36,
--     Dfval 45, Dfval 55]
```

foldrDf A right fold for data fields. Works similar to the Haskell `foldr`. The type is

```
foldrDf :: (Bounds c a, DeepSeq a) =>
           (a1 -> r -> r) -> Dfval r ->
            Datafield a a1 c -> Dfval r
```

scanrDf A right scan for data fields. Works similar to the Haskell `scanr`. The type is

```
scanrDf :: (Bounds c a, DeepSeq a) =>
           (a1 -> r -> r) -> Dfval r ->
            Datafield a a1 c -> [Dfval r]
```

7.2 Bounds

Functions and classes related to bounds are given in this section. Most of the important operations in bounds are done in classes to hide complexity and enable user defined bounds.

Bound This is the abstract data type that handles bounds. The type is `Bound a`.

(<:>) Operator to construct dense bounds. Member of class `DenseBound`.

```
class (Ix a, Show a, Pord a) =>
   DenseBound b a | a -> b where
      (<:>) :: a -> a -> b
```

This time using characters as indexes we create two dense bounds:

```
cb1 = 'a' <:> 'd' -- => <Bounds>: Dense 'a' to 'd'
cb2 = 'e' <:> 'h' -- => <Bounds>: Dense 'e' to 'h'
```

(<*>) Operator to construct product bounds. Member of class `ProdBound`.

```
class ProdBound a b c | a b -> c where
      (<*>) :: a -> b -> c
```

Using the two earlier constructed dense bounds we form a product bound:

```
cb1 <*> cb2
-- => <Product Bounds>: [<Bounds>: Dense 'a' to 'd',
--                       <Bounds>: Dense 'e' to 'h']
```

sparse Creates a sparse bound from a list of index values.

```
sparse :: (Ix a, Show a, Pord a) => [a] -> Bound a
```

Creating a sparse bound from three values:

```
sparse [1, 100, 1000] -- => <Bounds>: Sparse [1,100,1000]
```

predicate Creates a predicate bound from a predicate function.

```
predicate :: (Ix a, Show a, Pord a) =>
             (a -> Bool) -> Bound a
```

A predicate bound that checks if the given index is character `z`:

```
predicate (\x -> x=='z') -- => <Bounds>: Predicate
```

prod_n These functions creates product bounds. n should be substituted with numbers between 2 and 5 inclusive.

```
prod_n :: Bound a -> Bound b -> ... ->
          PnBounds (Bound a, Bound b, ...)
```

transBound Translates a given bound an amount given by an index. This is used for translation of data fields.

```
class Num c => TransBound b c | b -> c where
      transBound :: c -> b -> b
```

Simple projections These functions projects the specified dimension from a multidimensional bound. n can vary between 1 to 5. If n is 1, then it means that we want to project the first dimension. If a dimension does not exist, such as a fifth dimension in a two dimensional product bound, `Nothing` is returned.

```
class ProjSimple_m_n b c | b -> c where
      projSm_n :: b -> Maybe c
```

Restriction projections These are a variation on the simple projections. The operations are provided for calculating a new bound (which may be an approximation) from non-product multidimensional bounds or product bounds. Useful in cases where an index variable need to be fixed, such as if we have index (x, y) with some two dimensional bound, $b2$. Assume that we want to set $x = 1$, we would then get $(1, y)$. The new dimensional bound is then calculated from $b2$ by checking if 1 is within the first dimension of the bound. If this is the case then the projection function returns the second dimension, that for y in this example.

```
class RestrictProj_m_1 b c d | b -> c d where
      bprojpm_1 :: b -> c -> d
```

```
class RestrictProj_m_2 b c d | b -> c d where
      bprojpm_2 :: b -> c -> d
```

compactPBounds Convenient function to flatten a product bound into a sparse bound. Works only on finite bounds.

```
compactPBounds :: forall a b . Bounds b a => b -> Bound a
```

The following functions are all members of the bounds class which form the core of the module. This class is declared as

```
class (Ix a, Show a, Pord a) =>
       Bounds b a | b -> a where
```

universe Represents the universal bound(*all*) with type `universe :: b`.

empty Represents the empty bound(*nothing*) with type `empty :: b`.

finite Checks if a given bound is finite or not. Type is `finite :: b -> Bool`.

enum Returns an ordered list of indexes from the set defined by the finite bound.

```
enum :: b -> [a]
```

size Returns the size of a finite bound. Has type `size :: b -> Int`.

lowerBound Returns an index representing the lower bound of a finite bound.

```
lowerBound :: b -> a
```

upperBound Returns an index representing the upper bound of a finite bound.

```
upperBound :: b -> a
```

join Calculates the "union" of two bounds. The type is `join :: b -> b -> b`.

meet Calculates the "intersection" of two bounds. The type is

```
meet :: b -> b -> b
```

inBounds Binary function that checks if a given index is within bounds. Type is `inBounds :: a -> b -> Bool`.

7.3 Other Functions & Operations

This section presents functions and data types that aid programming with data fields.

Dfval The purpose of *Dfval* is to be able to express if the result from a datafield application is within the given bounds. The data type is declared as `data Dfval a = Dfval a | OutOfBounds`. The value `OutOfBounds` is the implementation equivalent of $*$ in the data field model. The data field returns a `Dfval` if the given index was in bounds and `OutOfBounds` otherwise. It is semantically identical to the `Maybe` data type and also member of the `Monad`-class. The only difference is that the constructors of `Dfval` are private and thus not directly accessible. The visible type is `data Dfval a`.

dfvalfun *dfvalfun* provides a convenient converter function that transforms any function into one that return a `Dfval`-value. The type of the function is `dfvalfun :: (a -> b) -> a -> Dfval b`.

dfval Is a simple wrapper function to wrap values in a `Dfval`. However, since the constructors of `Dfval a` are private, this is one of few ways to insert values in a `Dfval`-value. The type of dfval is `dfval :: a -> Dfval a`.

isoutOfBounds A predicate to check if a value is `OutOfBounds`. Type is `isoutOfBounds :: Dfval a -> Bool`.

outOfBounds A function that returns the value `OutOfBounds`, it has type `outOfBounds :: Dfval a`.

7.4 Syntactical Constructs & Translations

Together with the functions provided by the library, two syntactical constructs are introduced to simplify programming with Data Field Haskell and make it more expressive. Rules for automatically deriving the bounds for these syntactical constructs also exist. These three parts makes handling of data fields more convenient than just using library functions and explicitly specifying bounds.

The two constructs that Data Field Haskell introduces are:

- `foreach` is the Data Field Haskell implementation of the *φ-abstraction* in the model. The syntax for `foreach` is:

$$\texttt{foreach}\ apat_1 ... apat_n \rightarrow exp$$

  The bounds is then automatically derived from *exp*.

- `for` is a syntax for defining data fields by cases. The cases consists of a pair formed by a bounds expression(b_i) and a Haskell expression(e_i). This is very similar to the Haskell case-expression. The syntax is:

$$\texttt{for}\ pat\ \texttt{in \{}\ b_1 \rightarrow e_1\ \texttt{;}\ \ldots\ \texttt{;}\ b_n \rightarrow e_n\ \texttt{\}}$$

Since these syntactical constructs need to be translated to standard Haskell before they can be compiled, translation rules are given for each of the constructs.

FOREACH1

$$\texttt{foreach}\ x_1 \ldots x_n \rightarrow exp = \texttt{foreach}\ x_1 \rightarrow \ldots \rightarrow \texttt{foreach}\ x_n \rightarrow exp$$

This describes that a `foreach` with multiple arguments are translated into nested `foreach`-abstractions, each with a single argument.

FOREACH2

$$\texttt{foreach}\ x \rightarrow exp = \texttt{datafield}\ (\lambda x \rightarrow exp)\ \beta(exp, (x), \emptyset)$$

where β, explained in the next section, is the function for deriving bounds. `foreach`-abstractions with single arguments are translated into an application of the `datafield` function to a λ-abstraction and bound derived from the expression as explained earlier.

FOR The construct:

$$\texttt{for}\ pat\ \texttt{in \{}\ b_1 \rightarrow e_1\ \texttt{;}\ \ldots\ \texttt{;}\ b_n \rightarrow e_n\ \texttt{\}}$$

translates into:

```
(foreach pat -> if inBounds pat (b1) then e1 else if ...
                else if inBounds pat (bn) then en
                else outOfBounds)
                <\> (b1) `join` ... `join` (bn)
```

After this conversion is done, the remaining translation is handled by the `FOREACH2` rule.

In our case translations are purely syntactical, so types are not checked during the translations. This must be handled by the compiler after all translations are done.

We present some examples of actual code in a Haskell module before translation:

```
module Test where

df1 = foreach x -> Dfval (2*x)

df2 = foreach x y -> Dfval (x + y)
```

```
 df3 = for x in (sparse [1,3,5,7,9]) -> Dfval True

 df4 = for y in (0 <:> 5) -> Dfval True
               (sparse [50,80,100]) -> Dfval True
               (55 <:> 75) -> Dfval False
```

and after translation:

```
module Test where
df1
  = datafield (\ x -> Dfval (2 * x))
      ($(calcBound [| (Dfval (2 * x)) |] ["x"]))
df2
  = datafield
      (\ x ->
         datafield (\ y -> Dfval (x + y))
           ($(calcBound [| (Dfval (x + y)) |] ["x", "y"])))
      ($(calcBound [| (Dfval (x + y)) |] ["x", "y"]))
df3
  = (datafield
       (\ x ->
          if inBounds x (sparse [1, 3, 5, 7, 9])
           then Dfval True else
            outOfBounds)
       ($(calcBound [| (Dfval True) |] ["x"])))
      <\> ((sparse [1, 3, 5, 7, 9]))
df4
  = (datafield
       (\ y ->
          if inBounds y (0 <:> 5) then Dfval True else
            if inBounds y (sparse [50, 80, 100])
             then Dfval True else
              if inBounds y (55 <:> 75) then Dfval False
              else outOfBounds)
       (join
          (join ($(calcBound [| (Dfval True) |] ["y"]))
             ($(calcBound [| (Dfval True) |] ["y"])))
          ($(calcBound [| (Dfval False) |] ["y"]))))
      <\>
      (join (join ((0 <:> 5)) ((sparse [50, 80, 100])))
            ((55 <:> 75)))
```

Note that some of the translated code has had to be modified by hand in order for the lines to fit the report.

7.5 Deriving Bounds

Bounds are derived automatically from expressions. We first give an informal presentation of how bounds are derived and then we present the formal rules.

The bounds for `foreach x -> e` are derived from `e`. If `e` consists of `a!x`, such that we have `foreach x -> a!x`, then the derived bounds would be `bounds a`. In the case of `foreach x -> a!x + b!x`, the bounds derived from this expressions would be `(bounds a) ‘meet‘ (bounds b)`. This is because the derived bounds depends on both `a` and `b`. Furthermore the `+`-operator is strict in both its arguments, thus the new derived bound is valid only where both `a` and `b` are valid. If the expression consists of a conditional, the bounds from the branches should be joined as any branch could be taken. Since the conditional is strict in the condition the expressions would look like:

```
foreach x -> if a!x then b!x else c!x
```

with derived bound:

```
(bounds a) ‘meet‘ ((bounds b) ‘join‘ (bounds c))
```

The rules for deriving bounds are given by the β-scheme. We have taken the rules and their explanations from Holmerin [12], where they first appeared. They have been edited slightly to fit into our context. As these rules were meant for the Data Field Haskell compiler, some of them might not be applicable to the current implementation of Data Field Haskell. This is especially true, since the module that handles the calculation of bounds was not fully completed. This means that there might be possible conflicts between rules and implementation in the bounds calculating module that has not yet been discovered. However, we present the β-scheme for completeness of the model. We first present definitions and help functions used:

> Below, and in the rules following, x and v stand for variables, while e and t stand for Haskell core-expressions.
>
> Some notes on the translation: The parameters in $\beta(e, \vec{x}, Y)$ are an Haskell core expression e, a tuple $\vec{x}$, alternatively written $(x_1, ..., x_n)$ (where n might be 1), and a set Y. e is the expression being analyzed. $\vec{x}$ is the argument which we analyze e as a data field over. At the beginning, this is the argument of the `foreach`-abstraction, and is thus a single variable, but since case-expressions may bind new variables to the components of a tuple, we also need to find the applications of data fields to those variables. This is done by analyzing the sub expression where the binding has effect with respect to the tuple which contains the new variables. The set Y is used to keep track of variables which are bound after the variable being abstracted over. These are needed since we can consider variables which are bound earlier as constants (i.e they can occur in the derived bound).
>
> By abuse of notation, we will write $Y \cup \vec{x}$ for $Y \cup x_1, ..., x_n$.
>
> To keep the description more readable the function `meet` is denoted by $\sqcap$, `join` by $\sqcup$, and `prod_n`$e_1 ... e_n$ is written either as $e_1 \times \ldots \times e_n$, as $\times_{i=1}^{n} e_i$, or, if all factors are identical, as e^n . We assume that all bound variables are distinct.
>
> We also define a family of projection functions on bounds, pr_k^m . Let ρ be a (set-theoretic) partial function from [1, m] to Haskell expressions, and b be a m-dimensional bound (i.e a bound which represents a set of m-tuples). The projection $pr_k^m(\rho, b)$ is the projection

of the bound b in the k:th dimension, with additional constraints in the dimensions for which the partial function ρ is defined. We first define pr_k^m for product bounds. Let π_k^m be a family of functions with the property $\pi_k^m(b_1 \times \ldots \times b_k \times \ldots \times b_m) = b_k$, and

$$pr_k^m(\rho, b) = \texttt{if } cond \texttt{ then } \pi_k^m(b) \texttt{ else empty}$$

where

$$cond = v_{i1} \texttt{ `inBounds` } \pi_{i1}^m(b) \&\& \ldots \&\& v_{il} \texttt{ `inBounds` } \pi_{il}^m(b)$$

$$\rho = (i_1, v_{i1}), \ldots, (i_l, v_{il})$$

This definition works for dense bounds as well, if we define

$$\pi_k^m = ((l_1, ..., l_k, ..., l_m) <:> (u_1, ..., u_k, ..., u_m)) = l_k \texttt{<:>} u_k$$

For sparse bounds we can define π_k^m as

$$\pi_k^m(\texttt{sparse l}) = \texttt{sparse } (\texttt{map } (\backslash(x_1, ..., x_k, ..., x_m) \to x_k) \texttt{ l})$$

and $pr_k^m(\rho, b)$ as

$$pr_k^m(\rho, b) = \pi_k^m(b \sqcap (\texttt{predicate p}))$$

where

$$\texttt{p} = (\backslash(x_1, ..., x_m) \to x_{i1} == v_{i1} \&\& ... \&\& x_{il} == v_{il}))$$

For predicate bounds, we have

$$pr_k^m(\rho, \texttt{predicate p}) = x_k \to p(\rho(1), ..., x_k, ..., \rho(m))$$

if $\rho(i)$ is defined for $i \in 1, ..., m\ k$, and

$$pr_k^m(\rho, \texttt{predicate p}) = \texttt{universe}$$

otherwise. For `universe` and `empty` we have

$$pr_k^m(\rho, \texttt{universe}) = \texttt{universe}$$

and

$$pr_k^m(\rho, \texttt{empty}) = \texttt{empty}$$

We continue with presenting the rules for deriving the bounds:

(LAM)

$$\beta(\backslash v_1 ... v_n - > e, \vec{x}, Y)$$
$$= \beta(e, \vec{x}, Y \cup v_1, ..., v_n)$$

(CASE1)

$$\beta(\texttt{case } x_i \texttt{ of } (v_1, ..., v_n) - > e; _- > e', \vec{x}, Y)$$

$$= ((\texttt{universe}^{i-1} \times \beta(e, \vec{v}, Y \cup \vec{x}) \times \texttt{universe}^{m-i}) \sqcap \beta(e, \vec{x}, Y \cup v))$$

(CASE2)

$$\beta(\texttt{case}\ x\ \texttt{of}\ Kv_1...v_n- > e;_- > e', \vec{x}, Y)$$

$$= \beta(e, \vec{x}, Y \cup v_1, ..., v_n) \sqcup \beta(e', \vec{x}, Y)$$

(APP1)

$$\beta(\texttt{(!)}\ e\ (t_1, ..., t_m), \vec{x}, Y)$$

$$= \mathcal{T}(\texttt{bounds}\ e, (t_1, ..., t_m), \vec{x}, Y), if FV(e) \cap (Y \cup x) = \emptyset$$

(APP2)

$$\beta(e_1\ e_2, \vec{x}, Y)$$

$$= \beta(e_1, \vec{x}, Y) \sqcap \beta(e_2, \vec{x}, Y)$$

(LET)

$$\beta(\texttt{let}\ v_1 = e_1; ...; v_n = e_n\ \texttt{in}\ e, \vec{x}, Y)$$

$$= \beta(e_1, \vec{x}, Y \cup v_1, \ldots, v_n) \sqcap \ldots \sqcap \beta(e_n, \vec{x}, Y \cup v_1, \ldots, v_n)$$

$$\sqcap \beta(e, \vec{x}, Y \cup v_1, \ldots, v_n)$$

(PFAIL)

$$\beta(\texttt{caseNoMatch}, \vec{x}, Y) = \texttt{empty}$$

(AFAIL)

$$\beta(\texttt{outofBounds}, \vec{x}, Y) = \texttt{empty}$$

(DEFAULT)

$$\beta(e, \vec{x}, Y) = \texttt{universe},$$

if none of the other rules apply

(TUPLE)

$$\mathcal{T}(b, (t_1, ..., t_m), (x_1, ..., x_n), Y)$$

$$= \times_{i=1}^{n} \sqcap_{k=1}^{m} \mathcal{C}(b, k, (t_1, ..., t_m), x_i, Y \cup \vec{x})$$

(COMP)

$\mathcal{C}(b, k, (t_1, ..., t_m), x_i, Y)$
$=$ transBound$(pr_k^m(\rho, b), a)$ if $t_k \equiv x_i + a$ where FV(a) $\cap Y = \emptyset$
$= pr_k^m(\rho, b)$ if $t_k \equiv x_i$
$= \mathcal{T}(pr_k^m(\rho, b), (t'_1, ..., t'_l), x_i, Y\ x_i)$
if $t_k \equiv (t'_1, ..., t'_l)$, and $x_i \in FV(t_k)$
$=$ universe otherwise
where $\rho = (j, t_j) | FV(t_j) \cap Y = \emptyset$

Finally we present the explanations of the above stated rules:

- The (LAM)-rule simply keeps track of variables bound by λ-abstractions.

- The (CASE1)-rule handles the fact that case-expressions can be used to bind variables to the components of a tuple which is a component of the tuple $\vec{x}$. That is, we get a new representation $\vec{v} = (v_1, ..., v_m)$ of the component x_i in $\vec{x}$. This means that we need to consider data field being applied to the variables $v_1, ..., v_n$ as well as the original variables. This is handled by analyzing both over $\vec{v}$ and over $\vec{x}$ and applying $\sqcap$ to the results. Since $\vec{v}$ is a representation of a single component x_i of $\vec{x}$, the expression derived by $\beta(e, \vec{v}, Y \cup \vec{x})$ only restricts the bound in the dimension i. This is the reason for the `universe` bounds in the other dimensions. Since matching of tuples never fail, we do not need to bother with the other branch of the case-expression.
- The (CASE2)-rule handles case-expressions where the pattern is not a tuple. This means that (in general) any branch could be taken, which means that the bound derived for the case-expression should be $\sqcup$ applied to the bounds of the branches.
- The (APP1)-rule handles applications of data fields, on tuples or non-tuples (a non-tuple is simply considered a tuple of arity 1). One should note that this rule matches syntactically on the `!`-operator. Thus the rule does not hold if we replace `!` with `f`, even if `f` is defined as `f = (!)`. The details of data field application is given in the (TUPLE)-rule.
- The (APP2)-rule handles applications of other functions than `!`. Application is strict in the function being applied, so the bounds of the application will depend on the bounds of data fields occurring in the expression which we apply. The bounds of the application may or may not depend on data fields in the argument (for the corresponding rule for partial functions it depends on whether or not the function applied is strict), but for the purpose of the propagation of bounds we assume that all functions are strict, which means bounds from the argument should be propagated.
- The (LET)-rule handles `let` expressions. The (LET)-rule can be seen as a theorem following from the transformations of `let` to λ- and case-expressions given in the Haskell definition and the other rules given here. But since this is not obvious, we give the rule for (LET) here.
- The (PFAIL)-rule handles pattern-matching failure. We need to distinguish pattern-matching failure from other errors since we otherwise would get the bound universe for all case expressions.
- The (AFAIL)-rule should be self-explanatory (an expression which is out of bounds is defined nowhere).
- (DEFAULT) takes care of all cases which do not match any other rule.
- (TUPLE) defines the $\mathcal{T}$-scheme which is used to define data field application on tuples. The bound $\mathcal{T}\ (b, \vec{t}, \vec{x})$ calculated

from the bound b is a product where the i:th component is restricted by the occurrences of x_i in $\vec{t}$. Basically, if x_i occurs in t_k , then the k:th dimension of b might restrict the i:th dimension of the resulting bound. Exactly how depends in what context x_i occurs. The details are given by the (COMP)-rule.

- (COMP) defines $\mathcal{C}$, which is used to by the $\mathcal{T}$-scheme to analyze the occurrences of a variable in a component of a tuple.

Chapter 8

Data Field Haskell Library

The intent of this chapter is to introduce the concepts, design and implementation of the Data Field Haskell library. The implementation is comprised of a library and a preprocessor. We first describe the two parts, the Data Field Haskell library and the Data Field Haskell preprocessor, in turn and then we describe the differences between the new implementation and the old implementation, dfhc98. For specific details regarding the library, please refer to the appropriate section in the appendix.

8.1 Data Field Haskell Library(DFHL)

The purpose of the library is to provide the necessary data field functions found in dfhc98. The functions in the library are equivalent to those found in dfhc98 but is not necessarily identical. The library also contains features that in dfhc98 was done in the runtime part of the compiler. Due to the use of some extensions to the Haskell language, the library currently requires the *Glasgow Haskell Compiler*(GHC) [10] in order to be used.

8.1.1 Design & Goals

The design of the library was determined by a number of goals set for this project. As mentioned earlier, in [18], there was an attempt to port dfhc98 to a newer version of the base compiler. This was only partly successful as dfhc98 was able to compile standard Haskell modules but not any of the indended data field specific extensions. Compiling a Data Field Haskell module to a binary resulted in segmentation fault when these binaries were executed. The goals were derived from the experience gained from porting dfhc98. The following qualities were both desired and required from the new implementation:

- Maintainability
- Portability
- Simplicity

These requirements led to an implementation that needed to be small, modular and relying on as few language extensions as possible. It also needed to be implemented in one language and be coded in such a way that it could be easily maintained.

The choice of language for the implementation was simple as Haskell is a very potent language and was the language of choice for the previous implementation. A decision was made to have as few modules as possible while still retaining modularity, to make the whole implementation compact. Some extensions in Haskell are used in the library as they form a fundamental part of the functionality. To enhance maintainability, portability and readability of the code, it has been written as simple as possible.

The main drawback with this approach could be that efficiency has been sacrificed compared to earlier implementations. This has not been measured and further studies might be required in order to determine if there is an actual difference in performance. Regardless, performance was deemed less important than the other desired goals.

8.1.2 Implementation

To make the implementation compatible with the previous version great care was taken to ensure that function names and their intended functionality would correspond. This was done by studying the reports done on dfhc and by looking through the source code for clues. As the design of the Data Field Haskell library fundamentally differs from that of dfhc98, the source code was mostly used to check function names, types and to give more information about areas that was less detailed in the reports.

The implementation consist of a total of five files. The files and their contents are described below:

Bounds.hs All data types and functions related to bounds are found in this module. The implementation of bounds is a class based solution which means that users are able to add own bounds data types if desired.

Datafield.hs This is the main module which exports all functionality of the Data Field Haskell library. `Datafield.hs` should provide a nearly identical interface to that of dfhc98:s `Datafield.hs`.

Dfcommon.hs Functions and data types used in the whole library is found in this module. One of the more important data types, `Dfval a`, in the library is found here. This is the return type of all data fields and it is an instance of the `Monad`-class. This enforces that the handling of `OutOfBounds` values are correct.

DeepSeq.hs This module performs a deep evaluation of its argument. This file, in its entirety, was found in a mailing list in November, 2006. See the module for details.

Pord.hs This module was compiled from various `Pord` class related files in the old dfhc98 compiler. It provides *least upper bound*, `lub`, and *greatest lower bounds*, `glb`, according to a *partial order* `lt`.

The recommended way to understand the library is to take a look at the Haddock generated documentation. Going through the code while reading the comments is another nice way of getting a deeper understanding of how the library works.

8.1.3 Future Improvements

Some important improvements to make in the library is to enhance performance, add new features and add a supporting framework for the preprocessor. Profiling the library and replacing slow parts with rewritten code could be a good start to improve performance. New desired features might crop up that need to be added, these can not be suggested now as the library has yet to see actual use. Since the bounds calculation part of the preprocessor was not completed during this thesis, there might still be some functions that could be added to assist the preprocessor stage. Another improvement that needs to be done is to modify the code that is using `deepSeq.lhs` to use a possible coming module provided by the next Haskell standard.

8.2 Data Field Haskell Preprocessor(DFHP)

The preprocessor handles the syntactical constructs found in Data Field Haskell. It translates these constructs into standard Haskell98 with Template Haskell extensions. The Template Haskell part of the translation is meant to provide the automatic derivation of bounds. This step, both syntax translation and bounds derivation, was handled in the frontend of dfhc98. As with the library, the modules responsible for the derivation of bounds needs to be compiled with GHC due to the use of Template Haskell. The preprocessor itself uses no such extensions and should be Haskell98 compliant.

8.2.1 Design & Goals

The goals of DFHP are very similar to that of DFHL. We will not repeat the goals here and instead continue with a discussion of the design. dfhc98 translated the syntactical construct in the front-end of the compiler. The parser used in dfhc98 is a parser combinator using a non standard monad. We could have decided to reuse the front-end from dfhc98 to create the preprocessor, however another approach was taken. We chose to use modules found in the GHC distribution that provide a lexer and a parser for Haskell98. The difference with this parser is that it is generated from a parser generator. It takes a specification similar to Yacc and generates a parser that can parse the language specified. The parser generator used is Happy, which is also written in Haskell. The reason for this choice is that it seemed easier and more extensible to have a generated parser than to have a monadic parser combinator.

The bounds deriving part was also separated from the syntax translating phase, since the resulting abstract syntax tree was to verbose and heavy to work with. Thus the preprocessor only handles translation of syntax and a separate module using Template Haskell is employed to handle the derivation of bounds for the syntactical constructs.

8.2.2 Implementation

The two parts that form DFHP is a preprocessor which does the syntax translation of the syntactical constructs according to specified translation rules and a module that uses Template Haskell to calculate the derived bounds for the constructs.

Data field specific keywords and nodes has been added to the lexer and the parser generator description. Sources with data field extensions are fed to the preprocessor which performs the translations and then creates a new source file that consists of Haskell-only expressions with Template Haskell parts. When compiled these Template Haskell parts are then transformed using the module for bounds calculation into Haskell expressions which represents the final bound.

Since the data field related extensions were modest, not much code was needed for the preprocessor. The `foreach` was handled by making it similar to how λ-abstractions were handled in the parser. `for` was modelled after the `case`-expression.

Currently a restriction of the patterns used to the `for` construct is needed. According the rules of translating the `for` construct into Haskell, the patterns found as arguments to the `for` construct must be able to be translated into expressions. There is a problem with this as patterns and expressions are distinct and not necessarily compatible. The type of patterns in the abstract syntax tree given from the preprocessor is `HsPat`, whereas the type of expressions are `HsExp`. If p is the set of patterns and e is the set of expressions, then the preprocessor can only handle elements from the set $p \cap e$ if we assume that elements in this set are the ones that have a representation in both pattern and expression sets. Patterns not allowed to appear as parameters to the `for` construct is:

HsPIrrPat This is a irrefutable pattern, written in code as `~`.

HsPWildCard Wildcard patterns(`_`).

HsPAsPat This is the node for a @-patterns.

HsPRec These are labelled patterns.

HsPApp This represents data constructor and argument patterns.

HsPInfixApp This is a pattern with infix data constructor.

HsPNeg A negated pattern.

If any of these are encountered by the preprocessor it will abort with an error message stating which of the restricted patterns stopped the process.

`CalcBound.hs` is the module responsible for deriving bounds. It works as a recursive function that traverses the abstract syntax tree from a Template Haskell quotation. It calculates the proper bound and which is then spliced in the code at compile time. The quotation and splicing code is put in to place via the preprocessor, so all the `calcBound` function does is to extract the abstract syntax tree from the `Q` monad, do the calculations and then return the result back in the `Q` monad.

8.2.3 Future Improvements

One obvious improvement is that the module responsible for deriving bounds should be completed. Currently only a basic framework that lack almost all functionality is done. The implementation should follow the rules for deriving bounds as specified. A potential problem when solving this, is the complexity of the abstract syntax tree received. There seem to exist no real shortcut to this problem. Most likely a brute force approach must be taken, that is one must handle a node at a time taking care to always be consistant with the rules.

Another improvement that could be performed on the preprocessor itself is to see if the current restrictions on allowed patterns can be softened. This would allow a greater number of valid programs to pass through the preprocessor. How much the restrictions can be softened depends on if valid transformations exist between specified patterns and expressions.

When compiling the preprocessed sources, possible error messages that might arise are very difficult to track in the unprocessed source. Since the preprocessor introduces additional code in the preprocessed sources an error on a line a in the original file might be reported as lying on line $a + n$ where n can be any integer. A solution to this problem would be very beneficial as it would make tracking down bugs in the code much easier than it is currently.

The transformation can lead to redundant calculations of bounds from expressions. It is desirable, from a performance aspect, if these redundancies could be minimized or eliminated completely.

Finally it is important to remember that, since the preprocessor utilises a separate module to calculate bounds, any change in the interface of the bounds calculating function requires a matching change in the preprocessor. If this change is forgotten then the preprocessor will insert code that tries to call an obsolete version of the bounds deriving function.

8.3 Differences with dfhc98

The most notable difference is that the presented solution consists of a library with data field related functions and a preprocessor. The old implementation, dfhc98, was based on a full Haskell compiler and therefore able to use the underlying structure for more efficiency. We present two lists to give an overview of the differences in each solution. In the first we compare the library part with the matching parts in dfhc98:

Bounds dfhc98 had problems with a proper implementation of product bounds. The reason was that, during the time of the implementation of dfhc, there were no extensions to Haskell98 that could be used to express the types needed for product bounds. This problem is inherent to Haskell98 and could only be bypassed now due to new extensions to the language. Assume two bounds, `Bounds a` and `Bounds b`, the resulting product bound should then have type `Bounds (a, b)`. But any constructor in data type `Bounds a` would have a type `t -> Bounds a`, where free variables in `t` must be `a`. The solution to this problem was handled by using a lowlevel approach [12] with coercions to achieve the desired effect. In DFHL, this problem is solved by using a combination of Generalised Algebraic Data Types(GADT) and Functional Dependencies(fundeps). The GADT:s allows construction of datatypes, where the return types of constructors not necessarily needs to coinside with the type of the datatype. In other words, a constructor for `Bounds a` can have type `t -> Bounds (a,b)`. With fundeps greater control is given to the programmer, as one can specialize classes and their instances. DFHL thus offer a bounds implementation that consists of Haskell98 code together with two extensions to the language.

Module Changes dfhc98 provides an extension to the standard prelude in the form of roughly a dozen modules to handle the data field specific functions. DFHL provides equivalent features in five separate files. `Datafield.hs` and `Bounds.hs` contains the implementation of data field- and bounds functions respectively. `Dfcommon.hs` contains common and useful functions used in the entire library. `Pord.hs` has been compiled from various pord related files in dfhc98. `HEval.hs`, which was a module found in dfhc98, is replaced with `DeepSeq.lhs`. Their functionality is equivalent but this change was done to have a more future proof solution as a version of a deepseq module seem likely to appear in the coming revision to Haskell98. A modification of the library to use the standard deepseq module would then be less problematic.

Derivation of Pord & HEval Instances dfhc98 offers automatic derivation of Pord and HEval instances. The Pord class have operations for partial order and HEval is used for hyperstrict evaluation. As DFHL is not a compiler and only a add-on library this functionality is not offered by DFHL.

Hyperstrict Evaluation & (OutOfBounds) Efficiency This is handled in dfhc98 by two separate mechanisms. The first mechanism is the HEval module which provide the Haskell interface and the second is a modification to the compiler runtime. The runtime is extended with an exception handler. If during a hyperstrict evaluation $*$ is encountered, the exception handler will ensure that no unnecessary calculations will be executed. This makes $*$ handling very efficient. DFHL, on the other hand, has no such features. The underlying runtime is, by design, not accessible by DFHL. In order for DFHL to mimic the behaviour of dfhc98 regarding $*$, the $*$ handling is done in a monad. In this respect, dfhc98 would probably be more efficient than DFHL.

Portability Since dfhc98 was based on a Haskell compiler, in particular the NHC compiler, the implementation will be dependent on which platforms the base compiler can run on. Another issue that one must take into account is that dfhc98, being a full compiler, requires a lot of tools to build it. It also needs to be compatible with those tools. In [18] this was investigated but the conclusions drawn in the report was that, due to the long period of time without maintenance, there were serious compatibility problems with essential tools. Even the work to port the data field related extension between different versions of the same base compiler can be extensive. This is especially true if maintenance of the implementation is not regular. Since dfhc98 consist of both Haskell code and code written in C, the compiler is affected not only by changes to Haskell but also to changes in the C standard and compilers. DFHL is written completely in Haskell, although it uses some extensions not found in other compilers. It is therefore currently tied to the GHC compiler. There is reason to believe that these extensions will find their way into the next Haskell standard. In such a case, porting DFHL to other compilers would be trivial. Maintenance and portability of DFHL is simplified by the fact that the library is in Haskell only and that the whole library only consists of five files.

Prelude Modifications dfhc98 modifies and extends the `Show` class to handle out of bounds values by printing `<OUB>`. Likewise the `putChar` and `putStr` in the Prelude is modified to print `<OUB>` when `OutOfBounds` is encountered. DFHL takes another approach as the `OutOfBounds` in DFHL is a actual data type that can be printed by deriving the `Show` class.

Size The compressed source of dfhc98 is about 1.1 megabyte whereas the source for DFHL is about 0.1 megabytes. Not all of the files in the compressed dfhc98 package is part of the actual dfhc98 implementation but the comparison should give a hint at differences in size.

The second list gives the differences between the preprocessor and corresponding parts in dfhc98:

Abstract Syntax Tree(AST) The AST received from dfhc98 and DFHP differs substantially. The tree from the older version and smaller compiler dfhc98 is much more compact and easier to handle than the one received from the GHC based parser generator. This makes calculations in the AST much more complex as one needs to deal with bigger and more verbose nodes in the tree.

Derivation of Bounds dfhc98 automatically derives a bound from the syntactical constructs. Currently the module in DFHL responsible for this part is not working, so this feature is still lacking compared to dfhc98.

Forall One of the syntactical constructs in dfhc98 was the `forall`-construct. However this word is already used in some Haskell compilers [10] and even a keyword in other implementations [4], so in DFHL the `forall` has been renamed to `foreach`. Besides the superficial change of name, the `foreach`-construct is identical to the `forall`-construct

Lexer The lexer used in dfhc98 is a handwritten lexer that has been modified to include the data field extensions. The DFHP lexer was based on the `Language.Haskell.Lexer` source module provided by GHC.

Parser The parser used in dfhc98 is based on a monadic parser combinator. On the contrary DFHL uses a parser generator to generate its parser, from a modified parser description file also provided by GHC. Adding new constructs to the parser is very easily done as only one file needs to be modified for the parser to recognize new syntax. Of course the corresponding abstract syntax tree needs to be modified as well if more advanced calculations are needed.

Portability The portability of dfhc98 was already discussed earlier and so will be skipped here. The portability of DFHP depends on availability of a Haskell compiler, a parser generator (Happy) and Template Haskell for the boundsderiving module. If only the syntax translating frontend is wanted, the Template Haskell requirement can be dropped.

Syntax Translation Both dfhc98 and DFHP are unable to handle other extensions to Haskell98 other than the data field extensions. In DFHP, there should be no restrictions when handling `foreach`-constructs. When dealing with the `for`-construct, there are some restrictions imposed by

DFHP. Not all patterns can be translated by the preprocessor and use of non-supported patterns leads to an error when trying to translate the mentioned pattern.

Type Checking dfhc98 offers type checking of the data field syntactical constructs. DFHP is written purely as a syntax translator and as such has no information about types. Type errors are handled by the Haskell compiler after DFHP has been run on the source code. This ensures that type errors are caught but also leads to error messages that can be hard to track down. This is because the line number in the error message from the preprocessed source will most likely not correspond to the line number of the actual Data Field Haskell source.

Finally we conclude this part by noting a couple of things. In contrast to dfhc98, that comes as a full package, DFHL is fully modular. Each part of DFHL, the library, preprocessor and bounds calculation can be used independently of each other. Even the library implementation is fully modular with relevant and related parts confined in separate modules. dfhc98 most likely have a more efficient solution when dealing with $*$ values. Because dfhc98 is a full stand alone compiler with access to all parts, the datafield concept is more pervading in dfhc98 than it can be with a standard compiler added with DFHL. In the area of maintainability and portability, DFHL should stand as the most suitable candidate due to is small size and Haskell-only implementation.

Chapter 9

Conclusions and future work

In this thesis we have provided the background for this thesis and the reasons a new implementation was needed. We have given an overview of the data field model and its corresponding implementation in Haskell, Data Field Haskell. We also presented the design, goal, implementation and future improvements of the two constituent parts of the new implementation. Finally we gave a presentation of the differences between the old Data Field Haskell compiler, dfhc98, and the new Data Field Haskell implementation.

Once the initial obstacles had been solved, such as handling bounds in an uniform way or figuring out the proper type for functions, extending these concepts to larger dimensions was fairly easy as long as the dimensions did not get too large. Currently the limit of product dimensions are set to five. This is not a hard limit as more dimensions could be added, just as the library was extended from working with two or three dimensions to five dimensions. Five dimensions were chosen as this was the biggest tuple that the interactive environment in GHC would print as standard. Another reason was that adding dimensions also adds more type variables that need to be handled and thus complexity grows.

We have already given examples on some improvements that could be done, but we will give a short recount of some of the suggested improvements. Performance of the library could be improved as the current implementation focused on clarity and maintainability. Profiling the code should reveal opportunities to extract more performance. If modules duplicates functions that can be provided by standard Haskell libraries, then the rest of the library could be rewritten to use these functions instead and the redundant module could then be eliminated. The module that handles deep evaluation is one such module that could be replaced once alternatives arises. Since the bounds deriving module was not fully completed during this thesis, an emphasis should lie on completing it. The functionality of the module is one important part of the preprocessor, so a working module is highly desired. Finally improving the correlation of line numbers in Data Field Haskell source versus preprocessed source in error reporting from the compiler is needed. This will make tracking down bugs in code much easier.

The desired goals were reached with the library implementation. The whole library is compact and modular. Code is written with clarity in mind to help future maintainers. The library functions and preprocessor have been tested as far as possible. However, due to the bounds calculating part of the preprocessor not being completed, a full test coverage of Data Field Haskell have not been possible.

The work with the implementation of Data Field Haskell has been a rewarding experience. Although the library still needs to mature, it can be used at its current state.

Bibliography

[1] Data Field Haskell. `http://www.mrtc.mdh.se/projects/DFH/`.

[2] Haddock. `http://www.haskell.org/haddock/`.

[3] Happy. `http://www.haskell.org/happy/`.

[4] Hugs. `http://www.haskell.org/hugs/`.

[5] Nhc98. `http://www.haskell.org/nhc98/`.

[6] Template Haskell. `http://www.haskell.org/th/`.

[7] G.E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). 1993.

[8] AD Falkoff and KE Iverson. The design of APL. *ACM SIGAPL APL Quote Quad*, 6(1):5–14, 1975.

[9] J.L. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Mille. The Sisal Model of Functional Programming and its Implementation. *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms/Architecture Synthesis*, page 112, 1997.

[10] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[11] Haskell 98 Language and Libraries, The Revised Report. `http://www.haskell.org/onlinereport/`.

[12] Jonas Holmerin. Implementing Data Fields in Haskell. Master's thesis, Department of Teleinformatics, Royal Institute of Technology, November 1999.

[13] P. Hudak. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.

[14] Mark P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–??, 2000.

[15] P. Jones, S. Washburn, and G. Weirich. Wobbly types: Type inference for generalised algebraic data types, 2004.

[16] Björn Lisper and Per Hammarlund. The Data Field Model. Technical report, Mälardalen Real-Time Research Centre, Mälardalen University, June 2001.

[17] M. Metcalf, J.K. Reid, and M. Cohen. *Fortran 95 2003 explained.* Oxford Univ. Press, 2005.

[18] Jesper Simos. Porting Data Field Haskell. `http://www.mdh.se/ide/eng/msc/index.php?choice=show&id=0520`, August 2006.

[19] JM Sipelstein and GE Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, 1991.

[20] Johan Andreas Sjögren. Data Field Haskell 98. Master's thesis, Department of Computer Engineering, Mälardalen University, June 2001.

Appendix A

Data Field Haskell Modules

These are the modules that form the library part of the Data Field Haskell implementation. The module `DeepSeq.lhs` is not included here as it's not the work of the author and because it should be replaced by a Haskell standard module providing equivalent features when such a module exists.

A.1 Bounds.hs

```
{-# OPTIONS_GHC -fglasgow-exts #-}

-- Package: Datafield Haskell Library
-- Module: Bounds
-- Author: Jesper Simos
-- Copyright (c) 2007, Jesper Simos
-- License: GPLv2 (see base folder)
-- E-Mail: jss03001@student.mdh.se
-- Date: 2006-12-01
-- Last Change: 2007-02-27
--

-- | "Bounds" provide the means to construct and handle bounds and product bounds in an uniform manner.
-- There is a limit on the size of product bounds, this limit is currently set at 5.
module Bounds (Bound, single, unsingle, DenseBound((<:>)), ProdBound((<*>)),
               Bounds(universe, empty, finite, enum, size, lowerBound, upperBound, join, meet, inBounds),
               sparse, predicate, TransBound(transBound), ProjSimple_m_1(projSm_1),
               ProjSimple_m_2(projSm_2), ProjSimple_m_3(projSm_3), ProjSimple_m_4(projSm_4),
               ProjSimple_m_5(projSm_5), RestrictProj_m_1(bprojpm_1), RestrictProj_m_2(bprojpm_2
```

```haskell
                  ),
                compactPBounds, prod_2, prod_3, prod_4, prod_5 )
        where

-- Imports ---------------------------------------------------------------

import Ix
import Pord
import Dfcommon
import List

-- Constants -------------------------------------------------------------

-- | 'modulename' gives the name of the module as a string. Useful together with 'DFcommon.failwhere''.
modulename = "Bounds.hs"
-- | 'bprefix' denotes ordenary bounds. Used with "Show"-instance.
bprefix = "<Bounds>: "
-- | 'pbprefix' denotes product bounds. Used with "Show"-instance.
pbprefix = "<Product Bounds>: "

-- Precedence Declarations -----------------------------------------------

infixl 3 <*>
infix 2 <:>


-- | 'Dummy' is a dummy datatype to handle tuple types in single dimension bounds when using fundeps.
data Dummy a = Dummy a deriving (Eq, Ord, Ix, Show)

-- | Needed for technical reasons to resolve the Pord class constraint when using 'Dummy'.
instance (Pord a) => Pord (Dummy a) where
        glb (Dummy a) (Dummy b) = Dummy (glb a b)
        lub (Dummy a) (Dummy b) = Dummy (lub a b)
        lt (Dummy a) (Dummy b) = lt a b

-- | 'single' wraps a value in a 'Dummy' type.
single :: forall a. a -> Dummy a
single val = Dummy val

-- | 'unsingle' unwraps the 'Dummy' and yields the value.
unsingle :: forall a. (Dummy a) -> a
unsingle (Dummy val) = val

-- End Dummy
```

```haskell


-- Data Declarations ----------------------------------------

-- | The Bound datatype is the basic building block.
-- It can be dense, sparse(sets of points) or a predicate (infinite).
-- The special bounds Universe(infinite) and Empty(finite) forms the universal- and empty set.
data (Ix a, Show a, Pord a) => Bound a = Dense a a | Sparse [a] | Pred (a -> Bool) | Universe | Empty

-- This directive is needed to pass Haddock
-- #ifndef __HADDOCK__

-- | P(n)Bounds are composite bounds formed from products of basic bounds.
data P2Bounds a where
        P2Base :: a -> P2Bounds a
        P2Comp :: P2Bounds a -> P2Bounds b -> P2Bounds (a,b)

data P3Bounds a where
        P3Base :: a -> P3Bounds a
        P3Comp :: P3Bounds a -> P3Bounds b -> P3Bounds c -> P3Bounds (a,b,c)

data P4Bounds a where
        P4Base :: a -> P4Bounds a
        P4Comp :: P4Bounds a -> P4Bounds b -> P4Bounds c -> P4Bounds d -> P4Bounds (a,b,c,d)

data P5Bounds a where
        P5Base :: a -> P5Bounds a
        P5Comp :: P5Bounds a -> P5Bounds b -> P5Bounds c ->
                     P5Bounds d -> P5Bounds e -> P5Bounds (a,b,c,d,e)

-- #endif
-- Class Declarations ----------------------------------------


class Extract b a| b -> a where
        -- | 'extract' converts compound bounds to tuples of basic bound type.
        extract :: b -> a

class (Ix a, Show a, Pord a) => DenseBound b a| a -> b where
```

```haskell
        -- | '\<:\>' uses given values to construct bounds ranging from 1 to 5 dimensions.
        (<:>) :: a -> a -> b

class ProdBound a b c | a b -> c where
        -- | '<*>' composes product bounds from basic bounds.
        (<*>) :: a -> b -> c

-- | The methods in this class works on all bounds, both basic and product bounds.
class (Ix a, Show a, Pord a) => Bounds b a | b -> a where
        -- | 'universe' represents the universal bound.
        universe :: b
        -- | 'empty' represents the empty bound.
        empty :: b

        -- Operations
        -- | 'finite' checks if a bound is finite.
        finite :: b -> Bool
        -- | 'enum' enumerates a finite bound.
        enum :: b -> [a]
        -- | 'size' returns the size of a bound.
        size :: b -> Int
        -- | 'lowerBound' returns the lowest value in the bound.
        lowerBound :: b -> a
        -- | 'upperBound' returns the highest value in the bound.
        upperBound :: b -> a
        -- | 'join' combines two bounds. Similar to union operation of sets.
        join :: b -> b -> b
        -- | 'meet' combines two bounds. Similar to intersect operation of sets.
        meet :: b -> b -> b
        -- | 'inBounds' checks if a value is contained in the given bound.
        inBounds :: a -> b -> Bool

class (Num c) => TransBound b c | b -> c where
        -- | 'transBound' translates a given bound or product bound an amount of n where n can be a tuple.
        transBound :: c -> b -> b

-- | The classes ProjSimple_m_k (currently k can take on values from 1 to the upper limit
-- of product bounds) are a family of simple projection functions.
```

```
class ProjSimple_m_1 b c | b -> c where
        -- | 'projSm_1' returns the first dimension in a given product bound.
        -- Function works on all sizes of product bounds, it returns Nothing if dimension does not exist.
        projSm_1 :: b -> Maybe c

class ProjSimple_m_2 b c | b -> c where
        -- | 'projSm_2' returns the second dimension in a given product bound.
        -- Function works on all sizes of product bounds, it returns Nothing if dimension does not exist.
        projSm_2 :: b -> Maybe c

class ProjSimple_m_3 b c | b -> c where
        -- | 'projSm_3' returns the third dimension in a given product bound.
        -- Function works on all sizes of product bounds, it returns Nothing if dimension does not exist.
        projSm_3 :: b -> Maybe c

class ProjSimple_m_4 b c | b -> c where
        -- | 'projSm_4' returns the fourth dimension in a given product bound.
        -- Function works on all sizes of product bounds, it returns Nothing if dimension does not exist.
        projSm_4 :: b -> Maybe c

class ProjSimple_m_5 b c | b -> c where
        -- | 'projSm_5' returns the fifth dimension in a given product bound.
        -- Function works on all sizes of product bounds, it returns Nothing if dimension does not exist.
        projSm_5 :: b -> Maybe c

-- | The RestrictProj_m_k classes provide operations for calculating a new bound
-- (which may be an approximation) from non-product multidimensional bounds or product bounds
-- which is a restriction of the original bound. Currently only implemented for 2-dimensions.

class RestrictProj_m_1 b c d| b -> c d where
        -- | 'bprojpm_1' restricts the other dimensions and returns a bound approximating the first
```

```haskell
        -- dimension, given that the indices used to restrict the other dimensions are within the
        -- bounds they are going to restrict. Example(pseudocode but with correct types):
        -- @
        --    bprojpm_1 (Dense (\'a\', \'c\') (\'d\', \'f\')) (Just \'e\') => (Dense \'a\' \'d\')
        -- @
        --
        -- Since the character \'e\' is within the range of characters \'c\' - \'f\', the first dim. is returned.
        -- Another example, this time we simply restrict the first dim. and the function returns the second dim.:
        -- @
        --    bprojpm_2 (Dense ('a', 'c') ('d', 'f')) Nothing => (Dense 'c' 'f')
        -- @
        --
        bprojpm_1 ::  b -> c -> d
        bprojpm_1 _ _ = failwhere "Needs a instance declaration for this type!"

class RestrictProj_m_2 b c d| b -> c d where
        -- | 'bprojpm_2' restricts the other dimensions and returns a bound approximating the second
        -- dimension, given that the indices used to restrict the other dimensions are within the
        -- bounds they are going to restrict.
        bprojpm_2 ::  b -> c -> d
        bprojpm_2 _ _ = failwhere "Needs a instance declaration for this type!"

-- Instance Declarations -----------------------------------------------

-- This is needed for a general transBound
instance (Num a, Num b) => Num (a, b) where
        (+) (a1,b1) (a2,b2) = (a1+a2, b1+b2)
        (*) (a1,b1) (a2,b2) = (a1*a2, b1*b2)
        negate (a, b) = (negate a, negate b)
        abs (a, b) = (abs a, abs b)
        signum (a, b)= (signum a, signum b)
        fromInteger a = (fromInteger a, 0)

instance (Num a, Num b, Num c) => Num (a, b, c) where
        (+) (a1,b1,c1) (a2,b2,c2) = (a1+a2, b1+b2, c1+c2)
        (*) (a1,b1,c1) (a2,b2,c2) = (a1*a2, b1*b2, c1*c2)
        negate (a, b, c) = (negate a, negate b, negate c)
        abs (a, b, c) = (abs a, abs b, abs c)
```

```
        signum (a, b, c)= (signum a, signum b, signum c)
        fromInteger a = (fromInteger a, 0, 0)

instance (Num a, Num b, Num c, Num d) => Num (a, b, c, d) where
        (+) (a1,b1,c1,d1) (a2,b2,c2,d2) = (a1+a2, b1+b2, c1+c2, d1+d2)
        (*) (a1,b1,c1,d1) (a2,b2,c2,d2) = (a1*a2, b1*b2, c1*c2, d1*d2)
        negate (a, b, c, d) = (negate a, negate b, negate c, negate d)
        abs (a, b, c, d) = (abs a, abs b, abs c, abs d)
        signum (a, b, c, d)= (signum a, signum b, signum c, signum d)
        fromInteger a = (fromInteger a, 0, 0, 0)

instance (Num a, Num b, Num c, Num d, Num e) => Num (a, b, c, d, e) where
        (+) (a1,b1,c1,d1,e1) (a2,b2,c2,d2,e2) = (a1+a2, b1+b2, c1+c2, d1+d2, e1+e2)
        (*) (a1,b1,c1,d1,e1) (a2,b2,c2,d2,e2) = (a1*a2, b1*b2, c1*c2, d1*d2, e1*e2)
        negate (a, b, c, d, e) = (negate a, negate b, negate c, negate d, negate e)
        abs (a, b, c, d, e) = (abs a, abs b, abs c, abs d, abs e)
        signum (a, b, c, d, e)= (signum a, signum b, signum c, signum d, signum e)
        fromInteger a = (fromInteger a, 0, 0, 0, 0)


instance (Ix a, Show a, Pord a) => Show (Bound a) where
        show (Dense a b ) = bprefix ++ "Dense " ++ show a ++ " to " ++ show b
        show (Sparse l ) = bprefix ++ "Sparse " ++ show l
        show (Pred _ ) = bprefix ++ "Predicate"
        show (Universe) = bprefix ++ "Universe"
        show (Empty) = bprefix ++ "Empty"

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) => Show (P2Bounds (Bound a, Bound b)) where
        show p2b = let (a,b) = extract p2b in pbprefix ++ "[" ++ show a ++ "," ++ show b ++ "]"

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b, Ix c, Show c, Pord c) => Show (P3Bounds (Bound a, Bound b, Bound c)) where
        show p3b = let (a,b,c) = extract p3b in pbprefix ++ "[" ++ show a ++ "," ++ show b ++ "," ++ show c ++ "]"
```

```

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b, Ix c, Show c, Pord c, Ix d, Show d, Pord d) =>
        Show (P4Bounds (Bound a, Bound b, Bound c, Bound d)) where
          show p4b = let (a,b,c,d) = extract p4b
                     in pbprefix ++ "[" ++ show a ++ "," ++ show b ++ "," ++ show c ++ "," ++ show d ++ "]"

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b, Ix c, Show c, Pord c, Ix d, Show d, Pord d, Ix e, Show e, Pord e) =>
        Show (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e)) where
          show p5b = let (a,b,c,d,e) = extract p5b
                     in pbprefix ++
                        "[" ++ show a ++ "," ++ show b ++ "," ++ show c ++ "," ++ show d ++ "," ++ show e ++ "]"


instance Extract (P2Bounds a) a where
        extract (P2Base a) = a
        extract (P2Comp a b) = (extract a, extract b)

instance Extract (P3Bounds a) a where
        extract (P3Base a) = a
        extract (P3Comp a b c) = (extract a, extract b,
                                  extract c)

instance Extract (P4Bounds a) a where
        extract (P4Base a) = a
        extract (P4Comp a b c d) = (extract a, extract b,
                                    extract c, extract d)

instance Extract (P5Bounds a) a where
        extract (P5Base a) = a
        extract (P5Comp a b c d e) = (extract a, extract b,
                                      extract c, extract d,
                                      extract e)


instance DenseBound (Bound Bool) Bool where
           (<:>) a b = Dense a b

instance DenseBound (Bound Char) Char where
           (<:>) a b = Dense a b
```

```haskell

instance DenseBound (Bound Int) Int where
            (<:>) a b = Dense a b

instance DenseBound (Bound Integer) Integer where
            (<:>) a b = Dense a b

instance (Ix a, Show a, Pord a) => DenseBound (Bound a) (Dummy a) where
            (<:>) (Dummy a) (Dummy b) = Dense a b

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) =>
  DenseBound (P2Bounds (Bound a,Bound b)) (a,b) where
        (<:>) (a1, b1) (a2, b2) =
            P2Comp (P2Base (Dense a1 a2)) (P2Base (Dense b1 b2))

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
           Ix c, Show c, Pord c) =>
  DenseBound (P3Bounds (Bound a,Bound b,Bound c))
        (a,b,c) where
         (<:>) (a1, b1, c1) (a2, b2, c2) =
            P3Comp (P3Base (Dense a1 a2)) (P3Base (Dense b1 b2))
                   (P3Base (Dense c1 c2))

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
           Ix c, Show c, Pord c, Ix d, Show d, Pord d) =>
  DenseBound (P4Bounds (Bound a,Bound b,Bound c,Bound d))
        (a,b,c,d) where
         (<:>) (a1, b1, c1, d1) (a2, b2, c2, d2) =
            P4Comp (P4Base (Dense a1 a2)) (P4Base (Dense b1 b2))
                   (P4Base (Dense c1 c2)) (P4Base (Dense d1 d2))

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
           Ix c, Show c, Pord c, Ix d, Show d, Pord d,
           Ix e, Show e, Pord e) =>
  DenseBound (P5Bounds (Bound a,Bound b,Bound c,Bound d, Bound e))
        (a,b,c,d,e) where
         (<:>) (a1, b1, c1, d1, e1) (a2, b2, c2, d2, e2) =
            P5Comp (P5Base (Dense a1 a2)) (P5Base (Dense b1 b2))
                   (P5Base (Dense c1 c2)) (P5Base (Dense d1 d2))
                   (P5Base (Dense e1 e2))
```

```haskell
instance ProdBound (Bound a) (Bound b) (P2Bounds (Bound a, Bound b)) where
        (<*>) b1 b2  = P2Comp (P2Base b1) (P2Base b2)

instance ProdBound (P2Bounds (Bound a, Bound b)) (Bound c) (P3Bounds (Bound a, Bound b, Bound c)) where
        (<*>) b1 b2  = let (x,y) = extract b1
                       in P3Comp (P3Base x) (P3Base y) (P3Base b2)

instance ProdBound (P3Bounds (Bound a, Bound b, Bound c)) (Bound d) (P4Bounds (Bound a, Bound b, Bound c, Bound d))
      where
        (<*>) b1 b2  = let (x,y,z) = extract b1
                       in P4Comp (P4Base x) (P4Base y) (P4Base z) (P4Base b2)

instance ProdBound (P4Bounds (Bound a, Bound b, Bound c, Bound d)) (Bound e)
                  (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e)) where
        (<*>) b1 b2  = let (x,y,z,w) = extract b1
                       in P5Comp (P5Base x) (P5Base y) (P5Base z) (P5Base w) (P5Base b2)


instance (Ix a, Show a, Pord a) => Bounds (Bound a) a where
        universe = Universe
        empty = Empty

        -- Operations
        finite (Dense _ _ ) = True
        finite (Sparse _ ) = True
        finite (Pred _ ) = False
        finite (Universe) = False
        finite (Empty) = True

        enum (Dense a b ) = range (a,b)
        enum (Sparse l ) = nub (sort l)
        enum (Pred _ ) = failwhere "Enumeration Invalid!"
        enum (Universe) = failwhere "Enumeration Invalid!"
        enum (Empty) = []

        size b = length (enum b)

        lowerBound (Dense a b ) = a
```

```
    lowerBound (Sparse l ) = foldr1 glb l
    lowerBound (Pred _ ) = failwhere "lowerBound Invalid!"
    lowerBound (Universe) = failwhere "lowerBound Invalid!"
    lowerBound (Empty) = failwhere "lowerBound Invalid!"

    upperBound (Dense a b ) = b
    upperBound (Sparse l ) = foldr1 lub l
    upperBound (Pred _ ) = failwhere "upperBound Invalid!"
    upperBound (Universe) = failwhere "upperBound Invalid!"
    upperBound (Empty) = failwhere "upperBound Invalid!"

    join (Dense a1 b1 ) (Dense a2 b2 ) = Dense (glb a1 a2 ) (lub b1 b2)
    join (Dense a b ) (Sparse l) = Sparse (range (a,b) `union` l)
    join (Sparse l) (Dense a b ) = Sparse (range (a,b) `union` l)
    join (Dense a b ) (Pred p) = Pred (\x -> p x || inRange (a,b) x)
    join (Pred p) (Dense a b ) = Pred (\x -> p x || inRange (a,b) x)
    join (Sparse l1) (Sparse l2) = Sparse (l1 `union` l2)
    join (Sparse l) (Pred p) = Pred (\x -> p x || x `elem` l)
    join (Pred p) (Sparse l) = Pred (\x -> p x || x `elem` l)
    join (Pred p1) (Pred p2) = Pred (\x -> p1 x || p2 x)

    join Universe b2 = Universe
    join b1 Universe = Universe
    join Empty b2 = b2
    join b1 Empty = b1

    meet (Dense a1 b1 ) (Dense a2 b2 ) = Dense (lub a1 a2 ) (glb b1 b2)
    meet (Dense a b ) (Sparse l) = Sparse (range (a,b) `intersect` l)
    meet (Sparse l) (Dense a b ) = Sparse ( l `intersect` range (a,b))
    meet (Dense a b ) (Pred p) = Sparse [x | x <- range (a,b), p x]
```

```haskell
        meet (Pred p) (Dense a b ) = Sparse [x | x <- range (a,b), p x]
        meet (Sparse l1) (Sparse l2) = Sparse (l1 `intersect` l2)
        meet (Sparse l) (Pred p) = Sparse [x | x <- l, p x]
        meet (Pred p) (Sparse l) = Sparse [x | x <- l, p x]
        meet (Pred p1) (Pred p2) = Pred (\x -> p1 x && p2 x)

        meet Universe b2 = b2
        meet b1 Universe = b1
        meet Empty b2 = Empty
        meet b1 Empty = Empty

        inBounds x (Dense a b ) = inRange (a,b) x
        inBounds x (Sparse l ) = x `elem` l
        inBounds x (Pred p ) = p x
        inBounds x (Universe) = True
        inBounds x (Empty) = False

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) =>
    Bounds (P2Bounds (Bound a, Bound b)) (a,b) where

        universe = P2Comp (P2Base Universe) (P2Base Universe)
        empty = P2Comp (P2Base Empty) (P2Base Empty)

        -- Operations
        finite p2b = let (a,b) = extract p2b
                     in finite a && finite b

        enum p2b = let (a,b) = extract p2b
                   in [(x,y)|x <- enum a, y <- enum b]

        size p2b = length (enum p2b)

        lowerBound p2b = let (a,b) = extract p2b
                         in (lowerBound a, lowerBound b)

        upperBound p2b = let (a,b) = extract p2b
                         in (upperBound a, upperBound b)

        join p2b1 p2b2 =
          let (a1,b1) = extract p2b1
              (a2,b2) = extract p2b2
          in P2Comp (P2Base (join a1 a2)) (P2Base (join b1 b2))

```

```haskell
        meet p2b1 p2b2 =
          let (a1,b1) = extract p2b1
              (a2,b2) = extract p2b2
          in P2Comp (P2Base (meet a1 a2)) (P2Base (meet b1 b2))

        inBounds (a1,b1) p2b =
          let (a2, b2) = extract p2b
          in inBounds a1 a2 && inBounds b1 b2

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
          Ix c, Show c, Pord c) =>
   Bounds (P3Bounds (Bound a, Bound b, Bound c)) (a,b,c) where

        universe = P3Comp (P3Base Universe) (P3Base Universe) (P3Base Universe)
        empty = P3Comp (P3Base Empty) (P3Base Empty) (P3Base Empty)

        -- Operations
        finite p3b = let (a,b,c) = extract p3b
                     in finite a && finite b && finite c

        enum p3b = let (a,b,c) = extract p3b
                   in [(x,y,z)|x <- enum a, y <- enum b, z <- enum c]

        size p3b = length (enum p3b)

        lowerBound p3b =
          let (a,b,c) = extract p3b
          in (lowerBound a, lowerBound b, lowerBound c)

        upperBound p3b =
          let (a,b,c) = extract p3b
          in (upperBound a, upperBound b, upperBound c)

        join p3b1 p3b2 =
          let (a1,b1,c1) = extract p3b1
              (a2,b2,c2) = extract p3b2
          in P3Comp (P3Base (join a1 a2)) (P3Base (join b1 b2))
                    (P3Base (join c1 c2))

        meet p3b1 p3b2 =
          let (a1,b1,c1) = extract p3b1
              (a2,b2,c2) = extract p3b2
          in P3Comp (P3Base (meet a1 a2)) (P3Base (meet b1 b2))
```

```
                    (P3Base (meet c1 c2))

        inBounds (a1,b1,c1) p3b =
          let (a2, b2, c2) = extract p3b
          in inBounds a1 a2 && inBounds b1 b2 && inBounds c1 c2

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
          Ix c, Show c, Pord c, Ix d, Show d, Pord d) =>
     Bounds (P4Bounds (Bound a, Bound b, Bound c, Bound d))
        (a,b,c,d) where

        universe = P4Comp (P4Base Universe) (P4Base Universe)
                          (P4Base Universe) (P4Base Universe)
        empty = P4Comp (P4Base Empty) (P4Base Empty)
                       (P4Base Empty) (P4Base Empty)

        -- Operations
        finite p4b = let (a,b,c,d) = extract p4b
                     in finite a && finite b &&
                        finite c && finite d

        enum p4b = let (a,b,c,d) = extract p4b
                   in [(x,y,z,u)|x <- enum a, y <- enum b,
                                 z <- enum c, u <- enum d]

        size p4b = length (enum p4b)

        lowerBound p4b =
          let (a,b,c,d) = extract p4b
          in (lowerBound a, lowerBound b,
              lowerBound c, lowerBound d)

        upperBound p4b =
          let (a,b,c,d) = extract p4b
          in (upperBound a, upperBound b,
              upperBound c, upperBound d)

        join p4b1 p4b2 =
          let (a1,b1,c1,d1) = extract p4b1
              (a2,b2,c2,d2) = extract p4b2
          in P4Comp (P4Base (join a1 a2)) (P4Base (join b1 b2))
                    (P4Base (join c1 c2)) (P4Base (join d1 d2))
```

```haskell

          meet p4b1 p4b2 =
            let (a1,b1,c1,d1) = extract p4b1
                (a2,b2,c2,d2) = extract p4b2
            in P4Comp (P4Base (meet a1 a2)) (P4Base (meet b1 b2))
                      (P4Base (meet c1 c2)) (P4Base (meet d1 d2))

          inBounds (a1,b1,c1,d1) p4b =
              let (a2, b2, c2, d2) = extract p4b
              in inBounds a1 a2 && inBounds b1 b2 &&
                 inBounds c1 c2 &&inBounds d1 d2

instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b,
          Ix c, Show c, Pord c, Ix d, Show d, Pord d,
          Ix e, Show e, Pord e) =>
      Bounds (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e))
          (a,b,c,d,e) where

          universe = P5Comp (P5Base Universe) (P5Base Universe) (P5Base Universe)
                            (P5Base Universe) (P5Base Universe)
          empty = P5Comp (P5Base Empty) (P5Base Empty) (P5Base Empty)
                         (P5Base Empty) (P5Base Empty)

          -- Operations
          finite p5b =
            let (a,b,c,d,e) = extract p5b
            in finite a && finite b && finite c &&
               finite d && finite e

          enum p5b =
            let (a,b,c,d,e) = extract p5b
            in [(x,y,z,u,v)|x <- enum a, y <- enum b,
                            z <- enum c, u <- enum d,
                            v <- enum e]

          size p5b = length (enum p5b)

          lowerBound p5b =
              let (a,b,c,d,e) = extract p5b
              in (lowerBound a, lowerBound b, lowerBound c,
                  lowerBound d, lowerBound e)

          upperBound p5b =
              let (a,b,c,d,e) = extract p5b
```

```
            in (upperBound a, upperBound b, upperBound c,
                upperBound d, upperBound e)

        join p5b1 p5b2 =
            let (a1,b1,c1,d1,e1) = extract p5b1
                (a2,b2,c2,d2,e2) = extract p5b2
            in P5Comp (P5Base (join a1 a2)) (P5Base (join b1 b2))
                      (P5Base (join c1 c2)) (P5Base (join d1 d2))
                      (P5Base (join e1 e2))

        meet p5b1 p5b2 =
            let (a1,b1,c1,d1,e1) = extract p5b1
                (a2,b2,c2,d2,e2) = extract p5b2
            in P5Comp (P5Base (meet a1 a2)) (P5Base (meet b1 b2))
                      (P5Base (meet c1 c2)) (P5Base (meet d1 d2))
                      (P5Base (meet e1 e2))

        inBounds (a1,b1,c1,d1,e1) p5b =
             let (a2, b2, c2, d2, e2) = extract p5b
             in inBounds a1 a2 && inBounds b1 b2 && inBounds c1 c2 &&
                inBounds d1 d2 && inBounds e1 e2


instance (Ix a, Show a, Pord a, Num a) => TransBound (Bound a) a where
        transBound n (Dense a b ) = Dense (a-n) (b-n)
        transBound n (Sparse l ) = sparse (map (subtract n) l)
        transBound n (Pred p ) = predicate (p . \x -> x+n)
        transBound n (Universe) = universe
        transBound n (Empty) = empty

instance (Ix a, Show a, Pord a, Num a, Ix b, Show b,
            Pord b, Num b) => TransBound (P2Bounds (Bound a, Bound b)) (a,b) where
        transBound (n1,n2) p2b =
              let (a,b) = extract p2b
              in P2Comp (P2Base (transBound n1 a))
                        (P2Base (transBound n2 b))

instance (Ix a, Show a, Pord a, Num a, Ix b, Show b,
            Pord b, Num b, Ix c, Show c, Pord c, Num c) =>
     TransBound (P3Bounds (Bound a, Bound b, Bound c))
        (a,b,c) where
```

```
    transBound (n1,n2,n3) p3b =
        let (a,b,c) = extract p3b
        in P3Comp (P3Base (transBound n1 a))
                  (P3Base (transBound n2 b))
                  (P3Base (transBound n3 c))

instance (Ix a, Show a, Pord a, Num a, Ix b, Show b,
          Pord b, Num b, Ix c, Show c, Pord c, Num c,
          Ix d, Show d, Pord d, Num d) =>
   TransBound (P4Bounds (Bound a, Bound b, Bound c, Bound d))
      (a,b,c,d) where
    transBound (n1,n2,n3,n4) p4b =
         let (a,b,c,d) = extract p4b
         in P4Comp (P4Base (transBound n1 a))
                   (P4Base (transBound n2 b))
                   (P4Base (transBound n3 c))
                   (P4Base (transBound n4 d))

instance (Ix a, Show a, Pord a, Num a, Ix b, Show b,
          Pord b, Num b, Ix c, Show c, Pord c, Num c,
          Ix d, Show d, Pord d, Num d, Ix e, Show e,
          Pord e, Num e) =>
  TransBound (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e))
    (a,b,c,d,e) where
   transBound (n1,n2,n3,n4,n5) p5b =
         let (a,b,c,d,e) = extract p5b
         in P5Comp (P5Base (transBound n1 a))
                   (P5Base (transBound n2 b))
                   (P5Base (transBound n3 c))
                   (P5Base (transBound n4 d))
                   (P5Base (transBound n5 e))


instance ProjSimple_m_1 (P2Bounds (Bound a, Bound b)) (Bound a) where
        projSm_1 b = let (b1, b2) = extract b in Just b1

instance ProjSimple_m_1 (P3Bounds (Bound a, Bound b, Bound c)) (Bound a) where
        projSm_1 b = let (b1, b2, b3) = extract b in Just b1

instance ProjSimple_m_1 (P4Bounds (Bound a, Bound b, Bound c, Bound d)) (Bound a) where
        projSm_1 b = let (b1, b2, b3, b4) = extract b in Just b1

```

```
instance ProjSimple_m_1 (P5Bounds (Bound a, Bound b,
    Bound c, Bound d, Bound e)) (Bound a) where
        projSm_1 b = let (b1, b2, b3, b4, b5) = extract b
            in Just b1


instance ProjSimple_m_2 (P2Bounds (Bound a, Bound b)) (
    Bound b) where
        projSm_2 b = let (b1, b2) = extract b in Just b2

instance ProjSimple_m_2 (P3Bounds (Bound a, Bound b,
    Bound c)) (Bound b) where
        projSm_2 b = let (b1, b2, b3) = extract b in Just
            b2

instance ProjSimple_m_2 (P4Bounds (Bound a, Bound b,
    Bound c, Bound d)) (Bound b) where
        projSm_2 b = let (b1, b2, b3, b4) = extract b in
            Just b2

instance ProjSimple_m_2 (P5Bounds (Bound a, Bound b,
    Bound c, Bound d, Bound e)) (Bound b) where
        projSm_2 b = let (b1, b2, b3, b4, b5) = extract b
            in Just b2


instance ProjSimple_m_3 (P2Bounds (Bound a, Bound b)) (
    Bound b) where
        projSm_3 b = Nothing

instance ProjSimple_m_3 (P3Bounds (Bound a, Bound b,
    Bound c)) (Bound c) where
        projSm_3 b = let (b1, b2, b3) = extract b in Just
            b3

instance ProjSimple_m_3 (P4Bounds (Bound a, Bound b,
    Bound c, Bound d)) (Bound c) where
        projSm_3 b = let (b1, b2, b3, b4) = extract b in
            Just b3

instance ProjSimple_m_3 (P5Bounds (Bound a, Bound b,
    Bound c, Bound d, Bound e)) (Bound c) where
        projSm_3 b = let (b1, b2, b3, b4, b5) = extract b
            in Just b3


instance ProjSimple_m_4 (P2Bounds (Bound a, Bound b)) (
    Bound b) where
        projSm_4 b = Nothing
```

```
instance ProjSimple_m_4 (P3Bounds (Bound a, Bound b, Bound c)) (Bound c) where
        projSm_4 b = Nothing

instance ProjSimple_m_4 (P4Bounds (Bound a, Bound b, Bound c, Bound d)) (Bound d) where
        projSm_4 b = let (b1, b2, b3, b4) = extract b in Just b4

instance ProjSimple_m_4 (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e)) (Bound d) where
        projSm_4 b = let (b1, b2, b3, b4, b5) = extract b in Just b4


instance ProjSimple_m_5 (P2Bounds (Bound a, Bound b)) (Bound b) where
        projSm_5 b = Nothing

instance ProjSimple_m_5 (P3Bounds (Bound a, Bound b, Bound c)) (Bound c) where
        projSm_5 b = Nothing

instance ProjSimple_m_5 (P4Bounds (Bound a, Bound b, Bound c, Bound d)) (Bound d) where
        projSm_5 b = Nothing

instance ProjSimple_m_5 (P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e)) (Bound e) where
        projSm_5 b = let (b1, b2, b3, b4, b5) = extract b in Just b5


-- Instance for restriction of non-product bounds.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) => RestrictProj_m_1 (Bound (a,b)) (Maybe b) (Bound a) where
        bprojpm_1 (Dense (a1, a2) (b1, b2)) (Just i2) = if inRange (a2, b2) i2 then Dense a1 b1 else Empty
        bprojpm_1 (Dense (a1, a2) (b1, b2)) Nothing = Dense a1 b1
        bprojpm_1 (Sparse l) (Just i2) = Sparse (map fst (filter (\(_, b)-> i2==b) l))
        bprojpm_1 (Sparse l) Nothing = Sparse (map fst l)
        bprojpm_1 (Pred p) (Just i2) = Pred (\x -> p (x, i2))
        bprojpm_1 (Pred p) Nothing = Universe
        bprojpm_1 Universe _ = Universe
```

```haskell
        bprojpm_1 Empty _ = Empty

-- Instance for restriction of product bounds.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) =>
    RestrictProj_m_1 (P2Bounds (Bound a, Bound b)) (Maybe b) (Bound a) where
        bprojpm_1 p2b (Just i2) = let (b1, b2) = extract p2b
                                  in if inBounds i2 b2
                                     then b1 else Empty
        bprojpm_1 p2b Nothing = case (projSm_1 p2b) of
                                     Just b -> b
                                     Nothing -> failwhere "First dimension non existant! Something is disturbingly wrong!"

-- If one wants to extend bprojpm_k to handle more dimensions, just continue with the following
-- instance decl. Rules should be similiar to the code just above both one more dimension must be
-- handled. Likewise for a third dimension (bprojpm_3) just mimic the instance code and class for
-- RestrictProj_m_k, but take care to make sure that the code now should return the third dim. instead
-- of dim. k.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b, Ix c, Show c, Pord c) =>
    RestrictProj_m_1 (Bound (a,b,c)) (Maybe b, Maybe c) (Bound a)


-- Instance for restriction of non-product bounds.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) => RestrictProj_m_2 (Bound (a,b)) (Maybe a) (Bound b) where
        bprojpm_2 (Dense (a1, a2) (b1, b2)) (Just i1) = if inRange (a1, b1) i1 then Dense a2 b2 else Empty
        bprojpm_2 (Dense (a1, a2) (b1, b2)) Nothing = Dense a2 b2
        bprojpm_2 (Sparse l) (Just i1) = Sparse (map snd (filter (\(a, _)-> i1==a) l))
        bprojpm_2 (Sparse l) Nothing = Sparse (map snd l)
        bprojpm_2 (Pred p) (Just i1) = Pred (\y -> p (i1, y))
        bprojpm_2 (Pred p) Nothing = Universe
```

```haskell
        bprojpm_2 Universe _ = Universe
        bprojpm_2 Empty _ = Empty

-- Instance for restriction of product bounds.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b) =>
    RestrictProj_m_2 (P2Bounds (Bound a, Bound b)) (Maybe a) (Bound b) where
        bprojpm_2 p2b (Just i1) = let (b1, b2) = extract p2b
                                  in if inBounds i1 b1
                                     then b2 else Empty
        bprojpm_2 p2b Nothing = case (projSm_2 p2b) of
                                     Just b -> b
                                     Nothing -> failwhere "Second dimension non existant! Something is disturbingly wrong!"

-- Again this is an instance for 3-dim. that needs to be completed if one wants to extend the
-- restriction projections.
instance (Ix a, Show a, Pord a, Ix b, Show b, Pord b, Ix c, Show c, Pord c) =>
    RestrictProj_m_2 (Bound (a,b,c)) (Maybe a, Maybe c) (Bound b)

-- Functions ------------------------------------------------------------

-- | 'prod_2' constructs 2-dimensional datafields.
prod_2 :: Bound a -> Bound b -> P2Bounds (Bound a, Bound b)
prod_2 a b = a <*> b

-- | 'prod_3' constructs 3-dimensional datafields.
prod_3 :: Bound a -> Bound b -> Bound c -> P3Bounds (Bound a, Bound b, Bound c)
prod_3 a b c = a <*> b <*> c

-- | 'prod_4' constructs 4-dimensional datafields.
prod_4 :: Bound a -> Bound b -> Bound c -> Bound d ->
          P4Bounds (Bound a, Bound b, Bound c, Bound d)
prod_4 a b c d = a <*> b <*> c <*> d

-- | 'prod_5' constructs 5-dimensional datafields.
prod_5 :: Bound a -> Bound b -> Bound c -> Bound d -> Bound e ->
```

```haskell
           P5Bounds (Bound a, Bound b, Bound c, Bound d, Bound e)
prod_5 a b c d e = a <*> b <*> c <*> d <*> e

-- | 'sparse', given a list of points, yields a sparse bound.
sparse :: (Ix a, Show a, Pord a) => [a] -> Bound a
sparse set = Sparse set

-- | 'predicate' returns a predicate bound
predicate :: (Ix a, Show a, Pord a) => (a -> Bool) -> Bound a
predicate p = Pred p

-- | 'compactPBounds' flattens a finite product bound to a sparse tuple bound.
-- The size of tuple size is that of the dimension of the product bound.
compactPBounds :: forall a b. (Bounds b a) => b -> Bound a
compactPBounds pb = let list = enum pb in sparse list

-- | 'failwhere' gives an error with location information
failwhere :: String -> a
failwhere = failwhere' modulename


--tests


-- End of Module Bounds
```

A.2 Datafield.hs

```haskell
 {-# OPTIONS_GHC -fglasgow-exts #-}

-- Package: Datafield Haskell Library
-- Module: Datafield
-- Author: Jesper Simos
-- Copyright (c) 2007, Jesper Simos
-- License: GPLv2 (see base folder)
-- E-Mail: jss03001@student.mdh.se
-- Date: 2006-12-01
-- Last Change: 2007-02-12
--

-- | "Datafield" contains all functions that are needed to handle and calculate with datafields.
-- This module exports all needed functions from "Bounds" and "Dfcommon".
```

```haskell
-- A monadic style of programming is recommended when using datafields.
module Datafield (datafield, assoctoDf, dftoAssoc, (!), (<\>), bounds, translate, domain, tab, stricttab, hstricttab, foldlDf, foldl1Df, scanlDf, scanl1Df, foldrDf, foldr1Df, scanrDf, scanr1Df, Dfval, dfvalfun, dflookup, dfval, isoutOfBounds, outOfBounds, module Bounds) where

-- Imports ------------------------------------------------------------
import Ix
import Bounds
import Pord
import Dfcommon
import qualified Monad as M

-- Constants ----------------------------------------------------------

-- | 'modulename' gives the name of the module as a string. Useful together with 'failwhere'' in "Dfcommon".
modulename = "Datafield.hs"

-- Precedence Declarations --------------------------------------------

infixl 9 !
infixr 1 <\>

-- Data Declarations --------------------------------------------------

-- | The cornerstone of datafields. 'Datafield' a b c is a datafield of values from a to b with bounds of type c.
data (Ix a, Show a, Pord a, Bounds c a) => Datafield a b c = Datafield (a -> Dfval b) c | Tabfield [(a, Dfval b)] c

-- Class Declarations -------------------------------------------------

-- Instance Declarations ----------------------------------------------

-- Functions ----------------------------------------------------------

-- | 'datafield' constructs a datafield from a given function and bound.
-- Note that the value given must be a datafield value function. Use 'dfvalfun' from "Dfcommon" to convert a normal function.
datafield :: (Ix a, Show a, Pord a, Bounds c a) => (a -> Dfval b) -> c -> Datafield a b c
```

```
datafield f b = Datafield f b

-- | 'assoctoDf' takes an assoc list of index and values and converts it to a datafield.
assoctoDf :: (Bounds (Bound a) a) => [(a, b)] -> Datafield a b (Bound a)
assoctoDf l = Tabfield [(x, dfval y)| (x,y) <- l] (sparse [a | (a,_) <- l])

-- | 'dftoAssoc' converts a datafield to an assoc list.
dftoAssoc :: forall b a c. (Bounds c a, DeepSeq a, DeepSeq b) => Datafield a b c -> [(a, Dfval b)]
dftoAssoc df@(Datafield f b) = dftoAssoc (hstricttab df)
dftoAssoc (Tabfield l b) = l

-- | '!' applies a datafield to an index.
(!) :: forall a b c d. (Bounds c a,DeepSeq a) => Datafield a b c -> a -> Dfval b
(!) (Datafield f b) a = if inBounds a b then (f (deepS a)) else outOfBounds
(!) (Tabfield l b) a = if inBounds a b
                          then (dflookup a l)
                          else outOfBounds

-- | '<\>' restricts a given datafield with the bound given as second argument.
(<\>) :: (Ix a, Show a, Pord a, Bounds c a) => Datafield a b c -> c -> Datafield a b c
(<\>) (Datafield f b1) b2 = Datafield f (b1 `meet` b2)
(<\>) (Tabfield l b1) b2 = Tabfield l (b1 `meet` b2)

-- | 'bounds' returns the bounds of a datafield.
bounds :: (Ix a, Show a, Pord a, Bounds c a) => Datafield a b c -> c
bounds (Datafield _ b) = b
bounds (Tabfield _ b) = b

-- | 'translate' translates the given datafield an amount of n (where n can be a tuple).
translate :: forall c b a. (TransBound c a, Bounds c a) => a -> Datafield a b c -> Datafield a b c
translate n (Datafield f b) = Datafield (\x -> f (x-n)) (transBound (-n) b)
translate n (Tabfield l b) = Tabfield [(x+n, y)| (x,y) <- l] (transBound (-n) b)

-- | 'domain' gives the domain of a given datafield.
domain :: (Ix a, Show a, Pord a, Bounds c a, DeepSeq a) => Datafield a b c -> [a]
domain (Datafield f b) = deepS (enum b)
```

```haskell
domain (Tabfield l b) = deepS (enum b)


-- | 'tab' tabulates the datafield but does no evaluation of elements.
tab :: (DeepSeq a, Bounds c a) => Datafield a b c -> Datafield a b c
tab (Datafield f b) = Tabfield [(x, f x)| x <- enum b] b
tab t@ (Tabfield l b) = t

-- | 'stricttab' tabulates and evaluates each element to whnf
stricttab :: (DeepSeq a, Bounds c a) => Datafield a b c -> Datafield a b c
stricttab (Datafield f b) = Tabfield [( seqval x, seqval (f (seqval  x)))| x <- enum b] b
stricttab t@ (Tabfield l b) = (Tabfield (map seqval l) b)

-- | 'hstricttab' tabulates and does a deep evaluation each element.
hstricttab :: (DeepSeq a, DeepSeq b,Bounds c a) => Datafield a b c -> Datafield a b c
hstricttab (Datafield f b) = Tabfield [(deepS x, deepS (f (deepS x)))| x <- enum b] b
hstricttab (Tabfield l b) = (Tabfield (deepS l) b)


-- | 'foldlDf' is a foldl variant for datafields.
foldlDf :: (Bounds c a, DeepSeq a) => (r -> a2 -> r) -> Dfval r -> Datafield a a2 c -> Dfval r
foldlDf f a df = let f' z x = if isoutOfBounds (M.liftM2 f z (df!x)) then z else (M.liftM2 f z (df!x)) in foldl f' (a) (domain df)

-- | 'foldl1Df' is a foldl1 variant for datafields.
foldl1Df :: forall a2 c a. (Bounds c a, DeepSeq a) => (a2 -> a2 -> a2) -> Datafield a a2 c -> Dfval a2
foldl1Df f df = let f' z x = if isoutOfBounds (M.liftM2 f z (df!x))
                                  then z else (M.liftM2 f z (df!x))
                   in foldl f' (df!(head (domain df))) (tail (domain df))

-- | 'scanlDf' is a scanl variant for datafields.
scanlDf :: (Bounds c a, DeepSeq a) => (r -> a2 -> r) -> Dfval r -> Datafield a a2 c -> [Dfval r]
scanlDf f a df = let f' z x = if isoutOfBounds (M.liftM2 f z (df!x)) then z else (M.liftM2 f z (df!x)) in scanl f' (a) (domain df)
```

```haskell

-- | 'scanl1Df' is a scanl1 variant for datafields.
scanl1Df :: forall a2 c a. (Bounds c a, DeepSeq a) => (a2 -> a2 -> a2) -> Datafield a a2 c -> [Dfval a2]
scanl1Df f df = let f' z x = if isoutOfBounds (M.liftM2 f z (df!x))
                                then z else (M.liftM2 f z (df!x))
                  in scanl f' (df!(head (domain df))) (tail (domain df))

-- | 'foldrDf' is a foldr variant for datafields.
foldrDf :: (Bounds c a, DeepSeq a) => (a1 -> r -> r) -> Dfval r -> Datafield a a1 c -> Dfval r
foldrDf f a df = let f' x z = if isoutOfBounds (M.liftM2 f (df!x) z) then z else (M.liftM2 f (df!x) z) in foldr f' (a) (domain df)

-- | 'foldr1Df' is a foldr1 variant for datafields.
foldr1Df :: forall a1 c a. (Bounds c a, DeepSeq a) => (a1 -> a1 -> a1) -> Datafield a a1 c -> Dfval a1
foldr1Df f df = let f' x z = if isoutOfBounds (M.liftM2 f (df!x) z) then z else (M.liftM2 f (df!x) z)
                  in foldr f' (df!(last (domain df))) (init (domain df))

-- | 'scanrDf' is a scanr variant for datafields.
scanrDf :: (Bounds c a, DeepSeq a) => (a1 -> r -> r) -> Dfval r -> Datafield a a1 c -> [Dfval r]
scanrDf f a df = let f' x z = if isoutOfBounds (M.liftM2 f (df!x) z) then z else (M.liftM2 f (df!x) z) in scanr f' (a) (domain df)

-- | 'scanr1Df' is a scanr1 variant for datafields.
scanr1Df :: forall a1 c a. (Bounds c a, DeepSeq a) => (a1 -> a1 -> a1) -> Datafield a a1 c -> [Dfval a1]
scanr1Df f df = let f' x z = if isoutOfBounds (M.liftM2 f (df!x) z) then z else (M.liftM2 f (df!x) z)
                  in scanr f' (df!(last (domain df))) (init (domain df))


-- | 'failwhere' gives an error with location information.
failwhere :: String -> a
failwhere = failwhere' modulename


-- Tests

```

```haskell

-- End of Module Datafield
```

A.3 Dfcommon.hs

```haskell
-- Package: Datafield Haskell Library
-- Module: Dfcommon
-- Author: Jesper Simos
-- Copyright (c) 2007, Jesper Simos
-- License: GPLv2 (see base folder)
-- E-Mail: jss03001@student.mdh.se
-- Date: 2006-12-01
-- Last Change: 2007-02-12
--


-- | "Dfcommon" provides a set of useful functions related to datafields.
module Dfcommon (Dfval, failwhere', deepS, seqval, dfvalfun, dflookup, dfval, isoutOfBounds, outOfBounds, module DeepSeq) where

-- Imports ----------------------------------------------------------
import DeepSeq
import Monad
import List

-- Constants --------------------------------------------------------

-- | 'modulename' gives the name of the module as a string. Useful together with 'DFcommon.failwhere''.
modulename = "Dfcommon.hs"

-- Data Declarations ------------------------------------------------

-- | 'Dfval' is the value returned from datafields.
-- The 'Dfval' datatype is a member of the Monad instance. It works exactly as the Maybe datatype
-- where the corresponding constructors are (Just a - Dfval a, Nothing - OutOfBounds).
-- 'Dfval' differs from Maybe in some of its properties. Constructors are private and values remain in
-- the monad. Currently there is no way to extract the value from the monad, similar to values in the
-- IO monad.
--
-- When using a monadic style of programming, one can actually ignore return values in some instances.
-- Ex: We assume that df is a datafield value that can be "a" or "OutOfBounds".
```

```
--      Regardless of the value of df, the following lines of code will ignore the value of df.
--
-- @
-- do df
--      x <- somevariable
--      return x
-- @
data Dfval a = Dfval a | OutOfBounds deriving (Eq, Ord, Show)

-- Class Declarations ------------------------------------------


-- Instance Declarations ---------------------------------------

instance (DeepSeq a) => DeepSeq (Dfval a) where
     deepSeq (OutOfBounds) y = y
     deepSeq (Dfval x) y = deepSeq x y

instance Monad Dfval where
     (OutOfBounds) >>= f = OutOfBounds
     (Dfval x) >>= f = f x
     return         = Dfval
     fail _ = OutOfBounds

instance Functor Dfval where
     fmap f (OutOfBounds) = OutOfBounds
     fmap f (Dfval x) = Dfval (f x)

-- Functions ---------------------------------------------------

-- | 'dfvalfun' converts a function into a Datafield value function.
dfvalfun :: (a -> b) -> (a -> Dfval b)
dfvalfun f = \x -> dfval (f x)

-- | 'dflookup' works like lookup in Prelude but with Dfval instead.
dflookup :: Eq a => a -> [(a, Dfval b)] -> Dfval b
dflookup k []        = OutOfBounds
dflookup k ((x,y):l)
       | k==x       = y
       | otherwise = dflookup k l

-- | 'seqval' forces evaluation of its argument.
seqval :: a -> a
seqval val = val `seq` val

-- | 'deepS' forces deep evaluation of its argument.
```

```haskell
deepS :: (DeepSeq a) => a -> a
deepS val = val `deepSeq` val

-- | 'dfval' is a convenient wrapper for 'Dfval'.
dfval :: a -> Dfval a
dfval val = Dfval val

-- | 'isoutOfBounds' checks if a value is out of bounds.
isoutOfBounds :: Dfval a -> Bool
isoutOfBounds OutOfBounds = True
isoutOfBounds (Dfval _) = False

-- | 'outOfBounds' provides the out of bounds value.
outOfBounds :: Dfval a
outOfBounds = OutOfBounds

-- | 'failwhere'' raises an error and includes the module the error occured.
failwhere' :: String -> String -> a
failwhere' modul errorstr = error ("In " ++ modul ++ ": " ++ errorstr)

-- | 'failwhere' gives an error with location information.
failwhere :: String -> a
failwhere = failwhere' modulename

-- Test


-- End of Module Dfcommon
```

A.4 Pord.hs

```haskell
 -- Package: Datafield Haskell Library
-- Module: Pord
-- Author: Jesper Simos
-- E-Mail: jss03001@student.mdh.se
-- Date: 2006-12-01
-- Last Change: 2007-02-12
--

-- | This module provides the operations glb(greatest lower bounds) and lub(least upper bounds)
-- for select types and tuples ranging from 2 - 5.
-- The code in this module has been compiled from various Pord class related files
-- from the previous Data Field Haskell implementation(dfhc98, http:\/\/www.mrtc.mdh.se\/projects\/DFH\/docs\/).
```

```haskell
-- It has been extended with an instance for 5-tuples and comments.

-- Copyright notices and license covering the relevant files of dfhc98:
-- The data field extensions are written by Jonas Holmerin 1998-1999, and
-- some example code were contributed by BjÃ¶rn Lisper.
-- Modifications for dfhc98 contributed by Andreas SjÃ¶gren 2000-2001.

{-
License
.
.
.
see original file
-}

-- This module compilation, Copyright (c) 2007 Jesper Simos


module Pord (Pord (glb, lub, lt))where

-- Class Declaration ----------------------------------
-- | The 'Pord' class
class Ord a => Pord a where
    -- | 'glb' provides the greatest lower bounds of its arguments
    -- For tuples 'glb' can be seen as a pointwise 'Prelude.min' operation. Ex @ glb (0,5) (5,0) => (0,0) @
    glb :: a -> a -> a
    -- | 'lub' provides the least upper bounds of its arguments
    -- For tuples 'lub' can be seen as a pointwise 'Prelude.max' operation. Ex @ lub (0,5) (5,0) => (5,5) @
    lub :: a -> a -> a
    -- | 'lt' provides a partial order. It should be reflexive, antisymmetric and transitive.
    -- For non-tuple types 'lt' is the same as '<=' in Prelude.
    lt  :: a -> a -> Bool

-- Instance Declarations ------------------------------

instance Pord Bool where
```

```
 glb x y = min x y
 lub x y = max x y
 lt x y  = x <= y

instance Pord Char where
 glb x y = min x y
 lub x y = max x y
 lt x y  = x <= y

instance Pord Int where
 glb x y = min x y
 lub x y = max x y
 lt x y  = x <= y

instance Pord Integer where
 glb x y = min x y
 lub x y = max x y
 lt x y  = x <= y

instance Pord Ordering where
 glb x y = min x y
 lub x y = max x y
 lt x y  = x <= y

instance (Pord a, Pord b) => Pord (a,b) where
    glb (x1,y1) (x2,y2) = (glb x1 x2, glb y1 y2)
    lub (x1,y1) (x2,y2) = (lub x1 x2, lub y1 y2)
    lt (x1,y1) (x2,y2)  = lt x1 x2 && lt y1 y2

instance (Pord a, Pord b, Pord c) => Pord (a,b,c) where
    glb (x1,y1,z1) (x2,y2,z2) = (glb x1 x2, glb y1 y2, glb z1 z2)
    lub (x1,y1,z1) (x2,y2,z2) = (lub x1 x2, lub y1 y2, lub z1 z2)
    lt (x1,y1,z1) (x2,y2,z2)  = lt x1 x2 && lt y1 y2 && lt z1 z2

instance (Pord a, Pord b, Pord c, Pord d) => Pord (a,b,c,d) where
    glb (x1,y1,z1,u1) (x2,y2,z2,u2) = (glb x1 x2, glb y1 y2, glb z1 z2, glb u1 u2)
    lub (x1,y1,z1,u1) (x2,y2,z2,u2) = (lub x1 x2, lub y1 y2, lub z1 z2, lub u1 u2)
    lt (x1,y1,z1,u1) (x2,y2,z2,u2)  = lt x1 x2 && lt y1 y2 && lt z1 z2 && lt u1 u2

instance (Pord a, Pord b, Pord c, Pord d, Pord e) => Pord (a,b,c,d,e) where
    glb (x1,y1,z1,u1,v1) (x2,y2,z2,u2,v2) = (glb x1 x2, glb y1 y2, glb z1 z2, glb u1 u2, glb v1 v2)
```

```
    lub (x1,y1,z1,u1,v1) (x2,y2,z2,u2,v2) = (lub x1 x2, lub y1 y2, lub z1 z2, lub u1 u2, lub v1 v2)
    lt (x1,y1,z1,u1,v1) (x2,y2,z2,u2,v2)  = lt x1 x2 && lt y1 y2 && lt z1 z2 && lt u1 u2 && lt v1 v2

-- End of Module Pord
```