CHALMERS |  UNIVERSITY OF GOTHENBURG

Contract Checking for Feldspar

Master of Science Thesis in Computer Sience

Fatemeh Lashkari

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2012

Contract Checking for Feldspar

Fatemeh Lashkari

Examiner: Mary Sheeran
Supervisor: Koen Claessen

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden March 2012

Abstract

"Contracts play an important role in the construction of robust software" [13]. Program invariants are expressed in familiar notation with known semantics by using contracts. Assertions based on contracts has been widely used in procedural and object-oriented languages. Findler and Felleisen presented contracts to higher-order functional languages and Hinze, Jeuring et al. implemented contracts for Haskell as a library.

In this thesis, a contract language is introduced and implemented for three libraries of the functional language Feldspar. Feldspar is a domain specific language (DSL) for Digital Signal Processing, embedded in Haskell, and generating C code. Contracts are written for functions of the Core Array, Vector and Matrix libraries and also for some practical Feldspar functions.

A contract language should create an informative error message to report the violation and the violator when a contract fails. In this thesis, an error message specifies the cause of violation by reporting the file name, line number and if an argument of a function is to blame, this argument is also mentioned in the error message.

Contract checking can be done statically or dynamically. Static checking concentrates on complete checking of limited specifications at compile time. Dynamic checking focuses on incomplete checking of expressive specifications, and detects errors during run time. Contracts that are written in this thesis are checked with a dynamic contract checker. Furthermore, they are tested with QuickCheck, to ensure that contracts satisfy given properties. The result of these tests shows that the contracts hold their properties and we cannot find any bugs in the contracts written; so we conclude that all of the contracts written satisfy their properties.

The assert function is implemented for the Feldspar language to have contracts in the language. This assert function is translated to the C assert function which suggests the opportunity for verifying C program properties with contracts too. When a contract fails, Feldspar programmers can narrow down the cause of the violation with the help of a precise error message generated by the contract checker. During work on this thesis, we found several bugs in the Feldspar implementation by using the contract checker.

Acknowledgements

I would like to thank my supervisor Koen Claessen, for many inspiring discussions, and the guidance and encouragement that he has given me during my work with this thesis. The same goes for Emil Axelsson and Anders Persson for helping me to work with Feldspar Language. Finally, I would like to thank my examiner, Mary Sheeran, for all help with the report.

Contents

Chapter 1

Introduction

Verifying properties of software is one of the main topics in computer science. The reason is that program errors are common in software systems, and detecting them is difficult and costly [1]. One way to improve software reliability is to detect errors early and report them precisely during program development; the use of contracts is one approach to software verification. "A contract in a programming language is a formal and checkable interface specification that allows programmers to declare what a function assumes and what a function guarantees" [1]. Consider the following scenario between the head function and the function that calls head (from head's perspective): if you pass me a non empty vector, I will return its first element. This restriction on the input means that the head function need not deal with the case for an empty vector.

The object oriented programming community uses contracts widely [1]. Contract is added to higher-order functional languages by Findler and Felleisen. In this thesis, the use of contracts in a functional embedded domain specific language (Feldspar) is explored, building on work on contract checking for Haskell by Hinze, Jeuring et al. [2].

Contract checking of functions' properties can be done statically or dynamically. Static checking focuses on checking all limited specifications. This method detects errors during compile time, but may require complex theorem proving. Instead, dynamic checking detects errors during run time and does not check all expressive specifications. Static contract checking is more costly than dynamic checking for a given function, because contracts may need to be written for many other functions, in order to permit the necessary proof. Dynamic contract checking needs only to check whether or not the function being checked obeys its contract in a given run.

Most Digital signal processing (DSP) software is written in low level C; the DSP algorithms usually work well in C, but moving an application to a different target platform is expensive and time consuming. The reason is that most of the time converting an application to different platforms demands that code be rewritten to preserve optimizations. Instead a high level language could be compiled for different platforms without sacrificing the performance; therefore writing the

DSP algorithms in this language would be prefect. Note that most DSP targets only have C compilers; thus a translator to generate C code is necessary for the higher level language.

Feldspar is a domain specific language (DSL) for Digital Signal Processing, embedded in Haskell, and generating C code [3]. It is implemented as a deeply embedded core language, with higher level constructs (such as functions on vectors) provided as further libraries that translate to core, that is as shallow embeddings.

A major problem of Feldspar is returning a wrong result from C code when a function attempts to access an array outside its bounds. For example:

```
firstElement :: Data [Int32] -> Data Int32
firstElement  xs = xs ! 0

> c_firstElement []
2536440
```

The firstElement function takes an array and returns the first element of an input array. Feldspar converts the firstElement function in Haskell to the firstElement function in C. The $c\_firstElement$ is a command that calls the C compiler for returning the result of the C version of firstElement function. The result of the firstElement function is wrong. The C compiler should return an exception for this kind of situation instead of returning a value from memory. The main reason for implementing contracts for the Feldspar language is to solve this problem, so in this thesis the focus of implemented contracts is on the index and length properties of vectors and arrays.

In addition, the most common error in Feldspar programs that are compiled with the Haskell compiler is this error:

***** Exception: List.genericIndex: index too large.**

This error happens when a user wants to access a location that is not allocated; such as

```
let xs = (value []::Data [Int32])
>eval( firstElement xs)
*** Exception: List.genericIndex: index too large
```

This error message is not sufficient to explain the reason for aborting the program. This gives no information on where in the program the index function (!) is called with an index outside the bounds of the array or the vector. Assume the index function is called many times in the program; when this error happens the programmer does not know which of these functions is to blame. This question can be answered by defining a contract for the index function, since the contract reports who causes the violation precisely.

The purpose of this thesis is to investigate contract checking for Feldspar. Implementing contracts for Feldspar starts by studying code examples in order to discover violations and function properties of the Core.Array, the Vector and

the Matrix libraries in Feldspar. Then, QuickCheck is used to test all written contracts to ensure that contracts fulfill their properties. In this thesis most of the contracts focus on index and length restriction of functions in the three libraries being examined.

To read this thesis, it is assumed that the reader knows Haskell. Feldspar notation is used throughout the thesis and Feldspar features that are used in this thesis are described briefly in chapter two. To learn more about this language the Feldspar tutorial [4] is recommended.

1.1 Thesis outline

The structure of this thesis is as follows:

In chapter two, a brief overview of Feldspar Core Array, Vector and Matrix libraries is given.

In chapter three, all types of contracts and assert functions are presented.

In chapter four, the implementation of contracts and assert functions for Feldspar is explained and implementing informative error message for reporting the violations is described.

In chapter five, some of the contracts that are written for Feldspar library functions and Feldspar practical functions (discrete cosine transform and low pass filter) are presented.

In chapter six, contract testing methods that are used in this thesis are introduced and applied.

In chapter seven, related works are presented.

In chapter eight, conclusions and future work are discussed.

Contracts for the Core Array, Vector and Matrix libraries are available in appendix A. Properties for testing all contracts with QuickCheck are available in appendix B.

Chapter 2

Feldspar

Feldspar (Functional Embedded Language for DSP and PARallelism) is a domain specific language embedded for being used with Haskell for programming DSP algorithms. It is a joint research project between Ericsson AB, Chalmers University of Technology (Göteborg, Sweden) and Eötvös Loránd University (Budapest, Hungary).

Feldspar is built around a core language, which is a purely functional language on a level of abstraction similar to C. There are a lot of libraries built upon this core language to give a higher level of abstraction for programming. Programming in the core language provides conditions to have more control on the generated C code. Programming in Feldspar is like programming in Haskell. For example, the vector library in Feldspar has most functions of Haskell's list library. Features such as higher order functions, anonymous functions and polymorphism are inherited from Haskell. Feldspar also has the same static and strong type system as Haskell. The core language has a constructor `Data a` for all types. Primitive functions in Feldspar act like their equivalent Haskell functions [3].

```
(==) :: Eq a => Data a -> Data a -> Data Bool
(+) :: Numeric a => Data a -> Data a -> Data a
max :: Ord a => Data a -> Data a -> Data a
```

2.1 Working with Feldspar

Feldspar is imported as a normal library in Haskell, which makes it very appropriate for Haskell's programmers. In this thesis, the interpreter GHCi version 7.0.3 is used. This version was supplied with the Haskell Platform [5]. The Feldspar version is 0.5.0.1 [6, 7], but since it was not released when the project started, this thesis used an internal development version of Feldspar.

Install instructions:

```
 > cabal install feldspar-language
 > cabal install feldspar-compiler
```

In this section, some Feldspar features used in this thesis are briefly described. The function `eval` evaluates Feldspar functions. Only functions that are provided with all of their inputs can be evaluated in this way [3]. The function `value` is used to convert a Haskell value into a Feldspar value, so, it can convert a Haskell list into a Feldspar core array [3]. To use Feldspar, import it by writing `import Feldspar` in the top of a Haskell source file. For example:

```
import Feldspar
example = value [2,5,7,8 :: Int32]

>eval (example)
[2,5,7,8]
```

Core Array

Core arrays are like lists in Haskell. Arrays are created in both parallel and sequential ways. Setting and getting elements of arrays are implemented as functions in the core array library.

`parallel` is a core language function that computes the elements in a core array independently of each other. The arguments of this function are the length of the array and a function for computing the value of each index respectively [3].

```
parallel :: (Type a) =>
                Data Length -> (Data Index -> Data a) -> Data [a]


import Feldspar.Core

array = parallel 5 (\i -> (2 + i))
```

Vector

The vector library has most functions in the Haskell list library. The only difference between vectors and core arrays is that generating C code with arrays allocates memory but constructing C code with vectors allocates memory if the programmer explicitly forces this. There are two ways to define a vector in Feldspar. The first is to use the `indexed` function, which builds a vector from a length and an index function. The second is to convert a Haskell list to a vector by using the `vector` constructor [3]. For example

```
import Feldspar.Vector

vct1 = indexed 5 (+2)

vct2 = (vector [1..10]:: Vector (Data Int32))
```

Matrix

A matrix is a vector of vectors in Feldspar; so many functions from the vector library are usable on matrixes. Only basic matrix operations (like transpose and matrix multiplication) are implemented in the matrix library [3].

A matrix can be defined by the vector functions or the `indexedMat` function which is similar to the `indexed` function in the vector library. `IndexedMat` takes two lengths to determine the dimension of the matrix, and a function for mapping an index to a value. For example:

```
import Feldspar.Matrix

mx1 = value [[1,2,3],[9,7,5],[6,4,8]] :: Matrix Int32

mx  = indexedMat  3  3  (+)


> eval(mx)
[[0,1,2],[1,2,3],[2,3,4]]
```

Loops

One basic difference between programming in Feldspar and programming in Haskell, is that recursion on Feldspar values is not permitted. Haskell programmers usually use recursion to accomplish looping; instead, there are special functions in Feldspar to achieve this goal, such as `forLoop`:

```
forLoop :: Syntax a =>
             Data Length -> a -> (Data Index -> a -> a) -> a
```

The first argument of the `forLoop` function specifies the exact number of iterations. The second argument is the starting state and the last argument is a function, which takes an index and the current state, and computes the next state. The final state is the value which is returned by the function. The following example shows a function which sums the elements of a vector:

```
sum :: Vector1 Int32 -> Data Bool
sum v = forLoop (length v) 0 (\i st -> st + v!i)
```

Here, the v ! notation indexes into the input vector.

condition

The (?) construct returns the first component of the pair if the condition is true; otherwise the second component of the pair is returned.

```
(?):: (Syntax a) => Data Bool -> (a,a) -> a
```

```
isEven i = (i mod 2 == 0) ?  (true,false)

> eval(isEven 124)
true
```

Compiling

To compile a Feldspar function, the first step is to import the Feldspar.Compiler module. The `compile` function can then be used to compile a Feldspar function. This function has four arguments. The first argument is the Feldspar function to compile, the second is the name of the output file, the third is the name of the C function and the last one is compilation options (defaultOptions is used in this thesis). The defaultOptions generate C code according to the ISO C99 standard. All compilation steps are performed and no loop unrolling is made.

See the user s guide to the Feldspar compiler for more details: `http://feldspar.inf.elte.hu/feldspar/documents`

For example, take the following function from `Examples/Simple/Matrices.hs` for generating a parallel matrix.

```
matrix1 :: Matrix Index
matrix1 = indexed 2 vec
  where
    vec x = indexed 10 ((+x) . (*10))

>compile  matrix1  matrix.c  matrix  defaultOptions
```

This command writes the C output into a file named `matrix.c`. In addition, the generated C code can be written in the Haskell interpreter directly by using the `icompile` (interactive compile) function; such as:

```
>icompile matrix1



#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <complex.h>

void test(struct array mem, struct array * out0)
{
  setLength(out0, 2);
   {
     uint32_t i1;
     for(i1 = 0; i1 < 2; i1 += 1)
```

```
      {
        setLength(&at(struct array,(* out0),i1), 10);
         {
           uint32_t i2;
           for(i2 = 0; i2 < 10; i2 += 1)
            {
              at(uint32_t,at(struct array,(* out0),i1),i2) =
                                                  ((i2 * 10) + i1);
            }
           }
        }
    }
}
```

Chapter 3

Contracts

"A contract specifies a desired property of an expression" [2]. A simple contract is called a `comprehension contract` and is usually designed in the form $\{x|e\}$. The type of this contract is the same as the type of x and e has Boolean type. A contract can be viewed as a type [1] and they can have a name. Functions can take contracts as arguments or return them as results, since contracts are first-class citizen. For instance, the `nonEmpty` contract checks that a given vector is not null in Feldspar.

```
nonEmpty :: contract (Vector a)
nonEmpty  = \ v -> length v > 0
```

Contracts can be defined for values of arbitrary types, including function types by using contract comprehensions. Consider the contract $(\lambda f- > f0 > 0)$ specifies that 0 is a lower bound of a function-valued expression. Contract comprehension is restricted by a Boolean expression, so they are not sufficient to indicate all kinds of property for an expression. Consider the increment function that only takes a natural number as its argument; this function should return an output larger than its input. So, the output of the increment function depends on its input and a contract comprehension is not adequate for this condition. There must be a contract to specify the domain and codomain of the increment function.

The `dependent function contract` is built for solving the above problem and this contract is usually defined in the form $(x : e1)- > e2$ where $e1$ and $e2$ are contracts and x represents the arguments to the function [3].

```
inc ::Data Int32 -> Data Int32
inc = (\n -> n+1)
incCnt :: Contract(Data Int32)
incCnt = (\n -> inc n > n)
```

In general, a contract combinator for each parametric data type is required [2]. Pair and list data types can be expressed as contract combinators. The `dependent`

`product contract` is generated by combining two contract with the contract combinator for pair. The list contract creates a list of contracts by taking a contract on all elements of a list. The And contract combines contracts by using conjunction on two contracts: `c1 & c2` holds if `c1` and `c2` hold. The usability of contracts is increased with conjunction, because programmers can specify independent properties separately [2]. The last contract is the `Any` contract that satisfies any expression. The implementation of the contract data type is presented in chapter 4.

3.1 Assert

A contract is attached to an expression by using the `assert` function, such as:

```
head :: Vector1 a -> Data a
head = assert (notEmpty >->> Any) (\v -> head v)
```

Assert has following type:

```
assert  :Contract a -> (a-> a)
```

The `nonEmpty` is the precondition and `Any` is the postcondition of the `head` function. It means that the `head` function requires its argument to be a non empty vector and the output can be anything with type of `Data a`. These restrictions are specified for the `head` function by attaching the contract to the function with the help of `assert` function. The details of the code are explained in the next chapter.

When a contract is attached to an expression, the contract is dynamically monitored at run-time. Assert acts as the identity when the contract is satisfied. Otherwise, the evaluation is terminated with an error message; the error message must report the reason for violation precisely and correctly.

Contracts work in this way: if a contract fails, the error message points to the cause of violation. A dependent function contract is violated when wrong arguments are given to the function or the function itself is wrong. For instance, the decrement function should take a natural number as an argument and return a natural number.

```
nat :: Contract (Data Int32)
nat = (\i -> i> =0)

dec :: Data Int32 ->Data Int32
dec = assert (nat >->> nat)(\x -> x-1)

> eval(dec 1)
0
> eval (dec (-2) )
*** Exception Exception: Assert failed: contract failed.
```

```
> eval (dec 0)
*** Exception Exception: Assert failed: contract failed.
```

The first contract violation is caused by passing a negative value to the `dec` function : its precondition is violated, thus the argument is to blame. In the last call, the `dec` function is to blame, because it cannot deliver a natural number, so its postcondition is violated.

Contracts range from very specific to very general. The `nonEmpty` contract checks that the input vector is not null. This contract is a general contract for the vector library since most functions of this library require a none-empty argument. On the other hand, a contract may uniquely determine a value. Consider the function `reverse` which is supposed to reverse an input vector.

```
reversed :: Vector (Data a) -> Contract ( Vector (Data a))
reversed = \ xs rxs -> (isReverse xs rxs))

isReverse::(Type a, Eq a) =>
               Vector (Data a) -> Vector (Data a) -> Data Bool
isReverse xs ys = (lenx == leny)&&
                     forLoop lenx true
                        (\i result-> result &&
                          (xs!i ==ys!(leny-1 -i)))
    where lenx = length xs
          leny = length ys
```

This contract checks that `reverse xs` returns the elements of `xs` in reverse order by calling the function `isReverse`.

Chapter 4

The implementation of Feldspar contracts

The contract data type and the assert function are implemented based on the contract implementation by Hinze, Jeuring et al. [2]. There are some changes in the definition of the contract data type and in the assert function for using Feldspar notation; their implementation is explained in section 4.1. The implementation of an informative error message for reporting contract violations is presented in section 4.2.

4.1 The implementation of Contracts

All contract types that are explained in chapter three are implemented in Feldspar. The name of the contract comprehension's constructor is `Prop`.

```
data Contract aT where
 Prop    ::(aT -> Data Bool) -> Contract aT
 Function::Contract aT-> (aT ->Contract bT)->Contract(aT -> bT)
 Pair    ::Contract aT-> (aT -> Contract bT)-> Contract(aT, bT)
 List    ::Contract aT-> Contract [aT]
 And     ::Contract aT -> Contract aT -> Contract aT
 Any     ::Contract aT
```

The implementation of the `assert` function is presented in the following code.

```
assert                    :: Contract aT -> (aT -> aT)
assert (Prop p)  a         =(p a)? (a,(error "contract failed."))
assert (Function c1 c2) f  =(\ x -> (assert (c2 x).f)x).assert c1
assert (Pair c1 c2)(a1, a2)=(\ x -> (x , assert (c2 x) a2))
                                                    (assert c1 a1)
assert  (List c)    as     = map (assert c) as
```

```
assert  (And c1 c2)  a     = (assert c2 . assert c1 ) a
assert   Any         a     =  a
```

In the `assert` function, only the comprehension contract is checked immediately based on its definition. In the remaining cases, the contract components are attached to the related parts of the value to be checked. The checked argument `x` is passed to the codomain contract `c2` in the Pair and the Function cases. Hence, unchecked arguments could cause a runtime error in the postcondition [2]; for instance:

```
nonEmpty >->> (Prop (\y -> y <head x))
```

If `x` is a null vector, the postcondition cause a runtime error because of calling the `head` function on the null vector.

The `assert` function for a prop contract in the above code always returns an error message, independent of the condition value. The reason for this problem is the implementation of the conditional in Feldspar. The conditional evaluates both the if and the then branch, but when one of the branches contains an error command it returns an error without checking the condition and the creation of C code is aborted because of this situation. If we assume that the conditionals in Feldspar do not have this problem, the assert function in Feldspar is translated to a conditional expression in C code; but this is not our aim for this thesis. The goal of this thesis is that the assert function in Haskell is converted to the assert function in C to discover violtions through running C code too. Therefore, Feldspar needs to have an `assert` function to support the implementation of contracts.

One of the Feldspar developers added a module `Error` to the Core library for solving these problems. One of The `Error` module functions is the `assertMsg` function that is used in this thesis for implementing the `Prop` contract. The function `assertMsg` only returns an error message that is given as a string to the function when the condition is false. Also, this function translates the assert function in Haskell to the C assert function.

```
assertMsg :: Syntax a => String -> DataBool->a ->a
assertMsg  = sugarSym.Assert
```

The implementation of the `assert` function for the `Prop` contract is changed to the following code

```
assert (Prop p) a = assertMsg ("contract failed.") (p a) a
```

The relation between restrictions of a function contract is defined with the following infix operators.

```
pre >->>   post  =  Function pre (const post)

pre >>->  post  =  Function pre post
```

The >>-> is used when the postcondition is not a constant contract [2], such as:

```
reverse ::(Type a,Eq a) =>
             Contract (Vector (Data a) -> Vector (Data a))
reverse  = Any >>-> \xs -> Prop(\rxs -> (reversed xs rxs))
```

The reverse function accepts any vector, so the `Any` contract is selected for its precondition. The result of the `reverse` function must be the reverse of the input vector; this property is tested with the `reversed` function. The `reversed` function needs the input vector for checking the postcondition of the `reverse` function. Therefore, the postcondition of the `reverse` function is not a constant contract. In this example both precondition and postcondtion are merged in a contract with named `reverse`.

Now that the contract language is created for Feldspar, it is time to observe contracts in action. As an example, the `getIx` function from Core library takes an array and an index and returns the value of the index, such as:

```
>array   =parallel 3 (\i -> i+2)
> eval (getIx array  1 )
3

>eval (getIx array 5)
*** Exception: List.genericIndex: index too large.

> nullArray = value [] :: Data [Int32]

> eval(getIx nullArray 0)
*** Exception: List.genericIndex: index too large.
```

The `getIx` function returns an exception if the index is greater than or equal to the length of the input array or the array is null . The `getIx` contract is defined to prevent this uninformative exception. The preconditions of this function must check that the input array is not null and that the index is less than the array length; the `getIx` postcondition accepts any value. `notNullArray`, `validIndex` and `Any` contracts evaluate the `getIx` restrictions respectively.

```
import Feldspar.Core as F

getIx :: Type a=> Data [a] -> (Data Index -> Data a)
getIx = assert
        (notNullArray >>-> (\xs -> validIndex xs >->> Any))
          (\ xs x -> F.getIx xs x)

notNullArray :: (Type a,Syntax a) => Contract (Data [a])
notNullArray = Prop(\xs -> (getLength xs)> 0)

validIndex ::Type a=> Data [a]-> Contract (Data Index)
validIndex = (\xs -> Prop (\i -> (getLength xs) > i))
```

The argument of the `validIndex` contract is the input array that passes from the `notNullArray` contract to this contract. The `getIx` contract is converted to the following C code that contains the C assert functions for checking this function critical limitation and reporting the violations.

```
>icompile  (getIx :: Data [Int32] -> Data Index -> Data Int32)

#include "feldspar_c99.h"
#include "feldspar_array.h"
#include <stdint.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include <complex.h>
void test
   (struct array mem,struct array v0, uint32_t v1,int32_t * out)
{
    uint32_t e0;

    assert((getLength(v0) > 0));
    // {contract failed.}
    copyArray(&at(struct array,mem,0), v0);
    assert((getLength(at(struct array,mem,0)) > v1));
    // { contract failed.}
    e0 = v1;
    (* out) = at(int32_t,at(struct array,mem,0),e0);
}
```

When the `getIx` contract is called with the previous failing test cases, the results are:

```
>eval(getIx array 5)
*** Exception Exception: Assert failed: contract failed
>eval(getIx nullArray 0)
*** Exception Exception: Assert failed: contract failed
```

`Assert` plays the main role in contracts' implementation, but practical contracts needs to return an error message that indicates the cause of violation when a contract does not satisfy. However, the `assertMsg` function returns an uninformative error message and makes this implementation rather useless for practical purposes. The error message should point to the source location of the expression that causes the violation; the location is passed to the assert and contract functions to enable them to assign blame correctly. The following section considers the problem of creating accurate error messages.

4.2 The implementation of error messages

The implementation of error messages needs some functions and data types for keeping the source location of a program. The main type in this implementation is a type `Loc` that is defined for saving the location of arguments and functions. Blame assignment contains at least one location for holding a contract comprehension violation and two locations for a dependent function contract violation. In the case of dependent function contract, the argument location is addressed when the precondition fails and the function's location is reported when the postcondition fails. For holding the location of the argument in the former case, type `Locs` is passed to the `assert` function instead of type `Loc`; type Locs contains one or more location [1].

```
infixr :->
newtype aT :-> bT =Fun { apply :: Locs -> aT -> bT }
```

The type of functions that use the assert function is `aT :-> bT`, so the expression $(\lambda x : -> e)$ is written instead of writing `Fun (ls -> x -> e)` to simplify writing contracts.

The type of the Function constructor must be adapted because contracted functions have a distinguished type.

```
Function ::Contract aT -> (aT -> Contract bT) ->
                                          Contract (aT :-> bT)
```

The following code shows the complete implementation of the `assert` function with blame assignment. The `assert` function use following functions from the `Blame.lhs` file [10] to properly assign blame: the `blame` function creates the error message based on the given locations and the symbol `+>` is a function for combining two elements of type `Locs`;

```
assert                    ::  Contract aT -> (Locs -> aT -> aT)
assert (Prop p)  locs   a  =
       assertMsg  ("contract failed: "++ blame locs ) (p a) a

assert  (Function c1 c2)  locsf  f  =
       Fun (\locx ->(\x ->(assert (c2 x) locsf.apply f locx)x).
                                   assert c1 (locsf +> locx))

assert  (Pair c1 c2)  locs   (a1, a2)=
       (\ b1 -> (b1 , assert (c2 b1) locs a2))
                                          (assert c1 locs a1)
assert  (List c)    locs as = map (assert c locs) as
assert  (And c1 c2) locs a  = (assert c2 locs . assert c1 locs) a
assert   Any        locs a  = a
```

The Function contract implementation is explained in this paragraph.

```
assert  (Function c1 c2)  locsf  f  =
        Fun (\locx ->(\x ->(assert (c2 x) locsf.apply f locx)x).
                                        assert c1 (locsf +> locx))
```

Note that `locsf` are the locations involved in function contract `f` and the location of function argument is specified with `locx` (`locx` has type `locs` but it is always a single location). The steps of checking function contract `steps` are as follows. First, the precondition `c1` is checked; in this part `locsf` or `locx` can be blamed. Then function evaluation may contain extra checking and the argument location is passed to the function. At the end, `locsf` is passed to the postcondition `c2` for checking because the checked argument is given to `c2` and only function can cause the violation.

The `getIx` contract is rewritten to match the new implementation of the `contract` and `assert` functions.

```
getIx :: Type a=> Data [a] :-> Data Index :-> Data a
getIx = cAssert "getIx"
    (notNullArray >>-> (\xs -> validIndex xs >->> Any))
       (fun (\ xs-> fun ( \x ->(Feldspar.getIx xs x))))
```

To demonstrate the effects of these changes, the previous example from section 4.1 is called; the numbers 1 and 2 are given to the function as locations for arguments.

```
> eval(apply( apply getIx  1 array ) 2 10)
*** Exception: Assert failed: contract failed:
                         the expression labeled ‘2’ is to blame.

> eval(apply( apply getIx  1 nullArray ) 2 0)
*** Exception: Assert failed: contract failed:
                         the expression labeled ‘1’ is to blame.
```

The `cAssert` function is defined to simplify the use of the `assert` function. The user give, a string and this function converts the input string to a location and then calls the `assert` function.

```
cAssert  s c = assert c (makeloc (Def s))
```

Blame assignment is implemented for Feldspar by using the `Blame.lhs` file [10] from the implementation of contracts for Haskell by Hinze, Jeuring et al [2]. Functions of this file are briefly explained in this report; further information about this file can be found in section 5 of the Hinze, Jeuring et al. paper [2]. The `makeloc` function was defined in the `Blame.lhs` file [10]. This function changes the type `Loc` to type `Locs`, since the `assert` function input has type `Locs` for handling dependent function contracts as explained before.

The `fun` function is used in the `assert function` instead of a `Fun` expression:

```
fun f =  Fun (\ _ x -> f x)
```

However, applying the `fun` function in the `assert` needs to be changed because for each argument of a function the `fun` should be written. The `Functions` class is created to solve this issue; this class instantiates all sorts of `fun` instances. The `Functions` class can be found in appendix A.

```
getIx :: Type a=> Data [a] :-> Data Index :-> Data a
getIx = cAssert "getIx"
              (notNullArray >>-> (\xs -> validIndex xs >->> Any))
                   (functions (\ xs x ->( F.getIx xs x)))
```

When the `assert` function is called, locations for each contract should be passed to the assert function and this work is done by the `apply` functions. However, this is not practical because for each argument of a function an apply should be called. As an example, a function with three arguments needs to call the `apply` function three times; to avoiding writing apply for function arguments, these functions are defined:

```
apply1 f loc x  =  apply f (makeloc (App loc)) x

apply2 f loc x1 x2 = apply (apply f
        (makeloc (App loc)) x1) (makeloc (App (loc+1))) x2

apply3 f loc x1 x2 x3 =  apply(apply (apply f
        (makeloc (App loc)) x1) (makeloc (App (loc+1))) x2)
           (makeloc (App (loc+2))) x3
```

Note that passing the location to the function as an argument is not practical because, the real source location of a program is needed for reporting the violation in practice.

One method for accessing the source location of Haskell code is using the C preprocessor which is used in this thesis. The C preprocessor, known as cpp, transforms a program before compilation automatically. "Cpp is a standalone program. It reads a program file containing CPP directives and generates a program file without CPP directives. In the process, the program is transformed according to the directives" [11]. The preprocessing language is executed via directives and macros assist this language to be expanded. Macros are short form for arbitrary parts of C code. The preprocessor changes the macros with their definitions throughout the program.

Before useing a macro, it should be defined explicitly with the `# define` directive; a macro name and the intended extension of the macro are should be written after the `# define` directive [12]. There are two kinds of macros. An `object-like` macro is an identifier that will be changed by a code fragment. For example,

```
#define array_size 100
```

A `function-like` macro is the second type of macro that is defined like a function call. When a `function-like` macro is used the function pointer will get the address of the real function [12], as in the `LOC` macro which is explained in the following paragaraphs.

The standard predefined macros have the same meanings for all machines and operating systems on which GNU C is being used. Their names all start and end with double underscores. For instance, "__FILE__ macro expands to the name of the current input file, in the form of a C string constant" [12]. Such as:

`/usr/contract/ContractMacros.h`

"__LINE__ macro expands to the current input line number, in the form of a decimal" [12].

An informative error message can be generated if more details of the source location are accessible; so the type `Loc` is changed to hold more information. The new type `Loc` contains a file name, line number and number of function arguments(e.g. `arg`); the type of number of `arg` is `Maybe Int` because this data only exists for dependent function contracts.

```
data Loc
  = Loc
  { file :: String
  , line :: Int
  , arg  :: Maybe Int
  }
```

In this thesis, __FILE__ and __LINE__ macros are used to report an error message. Macros LOC, assert and three kinds of apply functions are defined for CPP in the file `ContractMacros.h`.

```
#define LOC (makeloc (Loc __FILE__ __LINE__ Nothing))

#define assert  assertLoc LOC
#define apply1  applyLoc1 LOC
#define apply2  applyLoc2 LOC
#define apply3  applyLoc3 LOC
```

The assert function implementation is changed based on its definition for CPP.

```
assertLoc :: Locs -> Contract a -> a -> a

assertLoc  locs (Prop p)  a           =
      assertMsg  ("contract failed: " ++ blame locs ) (p a) a

assertLoc  locsf (Function c1 c2) f   =
      Fun(\ locx -> (\ x -> (assertLoc locsf  (c2 x) .
                              applyP f  locx) x) .
                                assertLoc (locsf +> locx) c1)
```

```
assertLoc  locs (Pair c1 c2) (a1, a2) =
      (\ b1 -> (b1 , assertLoc locs (c2 b1') a2))
                                        (assertLoc locs c1 a1)

assertLoc  locs (List c)     as = map (assertLoc locs c ) as

assertLoc  locs (And c1 c2)  a =
       (assertLoc locs c2 . assertLoc locs c1 ) a

assertLoc  locs  Any         a = a
```

The C preprocessor scans the file that its name is mentioned after the `#include` directive then continuing work on current file. The order of the output from the C preprocessor is the output already generated, the included file's output and the output from the rest of text after the `#include` directive [12]. For example, given a header file to `Fcontract.h` (contains all written contracts) as follows,

```
{-# LANGUAGE CPP #-}
# include  ContractMacros.h

getIx :: Type a=> Data [a] :-> Data Index :-> Data a
getIx = assert (notNullArray >>-> (\xs -> check xs >->> Any))
                                        (functions Feldspar.getIx)
gtx = apply2 Fcontrac.getIx
```

The only problem of using the C processor is that the user is not able to call the apply functions in the Haskell interpreter(GHCi) because macros that are defined for CPP are not accessible from the Haskell interpreter. To solve this problem the programer should apply `# include ContractMacros.h` as a header of each file that needs to use the `apply` macro for calling a contract. Note that CPP reports the source location that calls `apply` functions, this means whenever the `gtx` is called the reported location is **Fcontract.hs:113**.

The `getIx` contract is called with the previous examples to show the error message that is created with help of CPP.

```
> eval (gtx array 5)
*** Exception: Assert failed: contract failed:
            the expression Fcontract.hs:113(arg#2) is to blame.

>eval(gtx nullArray  0)
*** Exception: Assert failed: contract failed:
            the expression Fcontract.hs:113(arg#1) is to blame.
```

Chapter 5

Examples

In this chapter some of the contracts written during this thesis are presented in detail. All contracts written are found in appendix A. The focus of the contracts written is on Feldspar critical parts (indexing and length functions). The `Any` contract is used for checking the output of Feldspar functions because we decide to trust the output of Feldspar functions. However, we found several bugs in Feldspar functions during this thesis by defining properties for their outputs; as an example, the `setIx` function is explained in the next section.

5.1 Contracts for the Array library

SetIx

The `setIx` function changes the value of a given index from the array. This function takes any array so the `Any` contract is selected for first argument of the `setIx` function. The next precondition of the `setIx` is that the given index should be less than the length of the array; this property is checked with the `validIndex` contract. There is no limitation for the input value so the `Any` contract is used for the last argument of this function; the result is not checked based on our assumption that the output of a Feldspar function is always correct.

```
setIx :: Type a => Data [a] -> Data Index -> Data a -> Data [a]
```

The `setIx` contract is written based on the above description:

```
setIx :: Type a =>
           Data [a] :-> Data Index :-> Data a :-> Data [a]
setIx = assert
          (Any >>->(\xs -> validIndex xs >->>Any >->> Any))
              (functions F.setIx)
```

```
stx = apply2 Fcontract.setIx
```

During testing of this contract, some of the wrong test cases do not fail and some of the correct test cases return strange results, such as:

```
array = value [4,2,1,2,3] :: Data [Int32]

wrongTest = Fcontract.stx array 10 12

>eval wrongTest
[4,2,1,2,3,12]

correctTest = Fcontract.stx array 0 9

>eval correctTest
[9,4,2,1,2,3]
```

`Fcontract` is a file that contains all contracts for functions. Note that if a contract does not terminate with a wrong test this means the contract does not check all violations; so instead of the `Any` contract as postcondition we defined the `equalLenArray` contract for checking that the length of the output array is the same as the input array length. The `setIx` contract changes to:

```
setIx :: (Type a,Eq a) =>
                Data [a] :-> Data Index :-> Data a :-> Data [a]
setIx = assert
        (Any >>->
          (\xs -> validIndex xs >->> Any >->> equalLenArray xs))
                     (functions F.setIx)

equalLenArray ::(Type a) => Data [a] ->  Contract (Data [a])
equalLenArray = (\a1 -> Prop(\a2 ->
                          (getLength a1) == (getLength a2)))
```

These strange results of testing are reported to the Feldspar developer team and they have now fixed the `setIx` bugs.

SetLength

The `setLength` function changes an array to an array with a desired length. Note that the given length must be less than or equal to the length of the input array; otherwise undefined elements are created, such as:

```
> eval(setLength 10 array)
[4,2,1,2,3]

>eval ((setLength 10 array)! 7)
*** Exception: List.genericIndex: index too large.
```

This property of the `setLength` function specifies one of its preconditions; the contract that checks this property is called `setLenArray`.

The `setLength` contract is implemented as follows:

```
setLength :: Type a =>  Data Length :->Data [a] :-> Data [a]
setLength = assert
               (Any >>-> (\len -> setLenArray len >->> Any))
                  (functions F.setLength)
```

```
>eval(setLength 10 array)
*** Exception: Assert failed: contract failed:
         the expression Fcontract.hs:108(arg#2) is to blame.
```

The contract violation is caused by passing number 10 as the length of the array to the `setLength` function. This number is bigger than the array length, so its precondition is violated, and hence the second argument is to blame.

5.2 Contracts for the Vector library

ScalarProd

The scalarProd function returns the scalar product of the two vectors. If the lengths of these vectors are not the same, Feldspar assumes the smaller length for both vectors. Consider the following example:

```
v = (vector [12,22,17,9] ::Vector1 Int32)
v1 = (vector [2,8,4] ::Vector1 Int32)

> eval(scalarProd v v1)
268
```

This result is not correct because scalar product is defined only for two input vectors with the same length in mathematics. The `scalarProd` contract is defined for checking this violation and solving this conceptual problem.

```
Import Feldspar.Vector as V

scalarProd :: (Numeric a,Eq a) =>
                  Vector (Data a) :-> Vector (Data a) :-> Data a
scalarProd = assert
                (Any >>-> (\v1 -> validLenVector v1 >->> Any))
                                    (functions V.scalarProd)


sclrPrd = apply2 Fcontract.scalarProd
```

```
validLenVector ::(Type a) =>
                    Vector (Data a) -> Contract (Vector (Data a))
validLenVector = (\v1 -> Prop(\v2 -> (length v1) == (length v2)))
```

Here, the precondition `validLenVector` precisely captures the intended semantics of scalar product. Evaluating the above example again with the scalarProd contract returns the following exception.

```
>eval(sclPrd v v1)
*** Exception: Assert failed: contract failed :
       the expression Fcontract.hs:64(arg#2) is to blame.
```

The second argument is identified as the cause of the contract violation since the length of `v` is not equal to the length of the `v1`.

Maximum

The `maximum` function returns the biggest number in a vector. This function fails if the input vector is a null vector.

```
nullV = (vector[])::Data Int32

>eval(maximum nullV)
*** Exception: List.genericIndex: index too large.
```

This property should be checked as the precondition of the function. The postcondition of the `maximum` function checks the result of the functionand with the help of the `maxed` function. `maxContract` contains both a precondition and a postcondition of the `maximum` function.

```
maximum  :: (Eq a,Ord a) => Vector (Data a) :-> Data a
maximum   = assert  maxContract (functions V.maximum )

mxmn = apply1 Fcontract.maximum

maxContract:: (Eq a,Ord a) =>
                     Contract (Vector (Data a):-> (Data a))
maxContract = nonEmpty >>-> \xs -> Prop(\m -> (maxed xs) == m)

maxed :: (Type a,Ord a) => Vector (Data a) -> Data a
maxed xs = forLoop (length xs) bMax (\i mMax -> max mMax (xs!i))
      where bMax = Contract.head xs
```

Examples of calling maximum contracts are:

```
>eval(mxmm v1)
8
```

```
>eval(mxmm nullV)
*** Exception: Assert failed: contract failed :
   the expression Fcontract.hs:64(arg#1) is to blame.
```

5.2.1 Contracts for higher order functions

fold1

One of the higher order functions in the vector library is `fold1`. The `fold1` function in Feldspar is the same as the `foldl` function in Haskell. Therefore, the only limitation for this function is that the input vector must not be null. We assume that the function parameter to the `fold1` does not have any restriction in order to simplify the contract. However, that function parameter to fold1 can have contracts to specify its properties like any function.

```
fold1 :: (Type a) =>
    (Data a -> Data a -> Data a) :-> Vector (Data a)  :-> Data a
fold1 = assert (Any >->> nonEmpty >->> Any) (functions V.fold1)

fld  = apply2 FContract.fold1
```

Examples:

```
>eval(fld1 (+) v1)
14

>eval(fld1 (+) nullV)
*** Exception: Assert failed: contract failed :
       the expression Fcontract.hs:89(arg#1) is to blame.
```

The contract violation is caused by passing a null vector to the `fold1` function.

Indexed

The `indexed` function takes a length to determine the length of the vector, and an index function which computes the elements based on their index in the vector. A simple example of using indexed is:

```
 indexed :: (Syntax a) =>
              Data Length -> (Data Index -> a) -> Vector a


> eval (indexed 4 (\i -> i+1))
> [1,2,3,4]
```

This function only has a postcondtion for checking that the vector has the requested length. The assumption about the `indexed` function parameter is similar to that used in the `fold1` function. Also, the length parameter can be any number, so it is represented by the Any contract.

```
indexed :: (Syntax a) =>
              Data Length :-> (Data Index -> a) :-> Vector a
indexed = assert (Any >>-> (\ len -> Any >->> validLen len))
                   (functions V.indexed)
```

`validLen` is a contract that checks that two vectors have equal length.

```
validLen:: (Syntax a ) => Data Length -> Contract (Vector a)
validLen= (\len -> Prop (\v -> length v == len ) )
```

This contract does not fail for some wrong test cases; instead it goes into an infinite loop during execution. All test cases that go into an infinite loop have the same property: a negative number is given as a length value. Since the `Data Length` type cannot be a negative number conceptually, the negative number input is viewed as a very big positive number in the Feldspar implementation. Therefore, we assume an upper bound for the given length to prevent infinite loops during execution with `positive` contract.

```
indexed :: (Syntax a) =>
              Data Length :-> (Data Index -> a) :-> Vector a
indexed  = assert
             (positive >>-> (\ len -> Any >->> (validLen len)))
               (functions V.indexed)


positive  ::  Contract (Data Length)
positive  =   Prop(\  i ->   i < 2000)

>eval(Fcontract.indxd (-2) (\i -> i+1))
*** Exception: Assert failed: contract failed :
     the expression Fcontract.hs:60(arg#1) is to blame.
```

The precondition of the `indexed` function fails because the given length is a negative number.

5.3 Contracts for the Matrix library

A matrix is defined as a vector of vectors in Feldspar. Consequently, a matrix in Feldspar may be built of rows of different lengths. We know the number of columns must be equal in all rows of a matrix in mathematics. For instance Feldspar allows the definition of the following matrix:

```
mx =  value [[9,0,8],[3,2]] :: Matrix Int32
```

`mx` is not a matrix but Feldspar accepts it as a matrix. Therefore, the `isMatrix` contract is built for checking that a matrix in Feldspar has the matrix property based on its definition in mathematics. This contract is used for all functions in the Matrix library whose input or output is a matrix.

```
isMatrix :: Type a => Contract ( Matrix a)
isMatrix = Prop(\xs -> ismatrix xs)

ismatrix :: Matrix a -> Data Bool
ismatrix xs = fold1 (&&)(map (== len) numColumn)
     where numColumn = map length xs
                 len = head numColumn
```

Multiply two matrices

The `mulMat` function multiplies two matrices, for example:

```
mulMat :: (Eq a ,Numeric a) => Matrix a -> Matrix a -> Matrix a

mx1 = value [[1,2,3],[9,7,5],[6,4,8]] :: Matrix Int32
mx2 = value [[9,7],[6,8]] :: Matrix Int32

> eval (multMat mx1 mx2)
[[21,23],[105,95]]
```

Note that two matrices can be multiplied in mathematics only if the number of columns in the first matrix is equal to the number of rows in the second matrix. Therefore, this result is wrong and the function should return an exception, because `mx1` number of columns is not equal to the `mx2` number of rows. The mulMat contract is defined for solving this problem, with the help of the `ismatch` contract. The `ismatch` contract checks if the number of columns of the first matrix is the same as the number of rows in the second matrix.

```
import Feldspar.Matrix as M

mulMat ::(Eq a ,Numeric a) =>
             Matrix a :-> Matrix a :-> Matrix a
mulMat = assert (isMatrix >>->
                (\mx -> (And isMatrix (isMatch mx)) >->> Any))
                         (functions M.mulMat)
mlMat = apply2 Fcontract.mulMat

ismatch :: (Eq a ,Numeric a) => Matrix a -> Contract(Matrix a)
ismatch = (\mx1 -> Prop(\mx2 -> (match mx1 mx2) ))


match ::(Type a,Eq a)=> Matrix a -> Matrix a -> Data Bool
match  mx1 mx2 =  (length (head mx1)) == (length mx2)


> eval (mlMt mx1 mx2)
*** Exception: Assert failed: contract failed :
        the expression Fcontract.hs:130(arg#2) is to blame.
```

Transpose

The `transpose` function exchanges the rows and columns of a matrix. For example,

```
transpose :: (Type a,Eq a) => Matrix a -> Matrix a

>eval(transpose mx2)
[[9,6][7,8]]
```

The transpose function transposes the rows and the columns of its input matrix. All these properties are checked with the `isTranspose` contract.

```
transpose :: (Type a,Eq a) => Matrix a :-> Matrix a
transpose = assert isTranspose (functions M. transpose )


trnsps = apply1 Fcontract.transpose

isTranspose :: (Type a,Eq a)=> Contract(Matrix a -> Matrix a)
isTranspose =  ismatrix >>-> (\m ->
                  Prop(\ tm -> (match m tm) && (match tm m))

> eval (trnsps mx)
*** Exception: Assert failed: contract failed :
     the expression Fcontract.hs:126(arg#1) is to blame.
```

`mx` is not a matrix; consequently the isMatrix contract raises the alarm.

5.4 Contracts for some Feldspar examples

In this part, some practical Feldspar functions are presented.

DCT

"The discrete cosine transform (DCT) is a technique for converting a signal into elementary frequency components" [8]. The DCT is generally used in image compression [8]. As an example the most common form of the DCT that is implemented as an example in the Feldspar tutorial with the name of `dct2` function is explained here. "DCT is often expressed such that each element in the output vector is a sum of the input elements multiplied by appropriate factors" [4]. The `dct2` function is implemented in the Feldspar tutorial as a matrix multiplication, the factors are the elements of the matrix. In this thesis instead of applying the function `indexedMat` for implementing this function the `indxMt` contract is used to prevent any violation while creating a matrix with the `indexedMat` function. The `indxMt` contract can be found in Appendix A.

```
dct2 :: (Vector1 Float) -> (Vector1 Float)
dct2 xn = mat *** xn
where mat = indxMt (length xn) (length xn)
                        (\k l -> dct2nkl (length xn) k l)
```

This function accepts any vector and the length of the output must be equal to the input vector length; so the contract of `dct2` is:

```
dCT2 :: (Vector1 Float) :-> (Vector1 Float)
dCT2 = assert (Any >>-> (\v -> checkLenVector v))
                                  (functions dct2)
```

The `dct2` function uses the helper function `dct2nkl` to compute all the values in the DCT-2n matrix. The `dct2nkl` function can be found in Appendix A.

low pass filter

The low pass filter is the next example that is explained here. A low-pass filter is a circuit offering easy passage to low-frequency signals and difficult passage to higher-frequency signals. The low-pass filter implemented in the Feldspar tutorial.

```
lowPassCore :: (Numeric a) =>
                  Data Index -> Vector1 a -> Vector1 a
lowPassCore k v = take k v ++
                        replicate (length v - k) 0
```

This function goes into an infinite loop if the input index is bigger than the input vector length because of the `replicate` function that is called as a helper function. There are two possibilities for solving this problem. One method is to use a contract for the lowPassCore function

```
lowPassCore ::(Numeric a) =>
                   Data Index :-> Vector1 a :-> Vector1 a
lowPassCore = assert
             (Any >>->(\i -> validIndex i >>->
                             (\v -> checkLenVector v)))
                                   (functions lowPassCore)
```

The following contract tests if the given index is less that the vector length

```
validIndex :: Type a => Data Index -> Contract (Vector1 a)
validIndex = (\i -> Prop (\v -> length v > i))

vf = (vector [7.8]) ::Vector1 Float

> eval(Fcontract.lPssCr 2 vf)
*** Exception: Assert failed: contract failed:
    the expression Fcontract.hs:290(arg#2) is to blame.
```

Another solution is to write a contract for the `replicate` function and use it in the `lowPassCore` function. The `replicate` contract can be found in Appendix A; it is called `rplct` contract.

```
lowPassCore    :: (Numeric a) =>
                  Data Index -> Vector1 a -> Vector1 a
lowPassCore k v = take k v ++ rplct (length v - k) 0

> eval(lowPassCore 2 vf)
*** Exception: Assert failed: contract failed:
    the expression Fcontract.hs:102(arg#1) is to blame.
```

Both solutions are implemented; however, the error message from the `lowPassCore` contract shows the reason for the violation very precisely.

Chapter 6

Testing Contracts

A program should not return any unexpected run time error if all functions in the program satisfy their contracts [1]. There are to check satisfaction of a contract, static contract checking and dynamic contract checking.

Dynamic contract checking checks contract satisfaction at run time and if any run time input violates the function's contract the failure is reported. Note that dynamic contract checking clarifies where the bug is and the compiler explains the reason for the bug. Dynamic checking often returns an incomplete result and detects a violation late; this is because it only checks the data values and code path of actual execution. For instance, the decrement (`dec`) function introduced in section 3.1 obviously does not satisfy its contract `(nat :-> nat)`, although this fact is not identified until the `dec` function takes zero as its argument. This property becomes even more important when the language has higher order functions. For instance,

```
f     :: (Data Int32 -> Data Int32) -> Data Int32 -> Data Int32
f g x = g x

cntF :: (Data Int32 :-> Data Int32) :-> Data Int32 :-> Data Int32
cntF =  assert((nat >->> nat) >->> nat >->> nat)(function f)

applyF = apply2 cntF
```

The function `f` takes a function argument of type `Data Int32 -> Data Int32`. Detecting contract violations cannot be expected when the `f` is applied to a function. We consider the `dec` function as a function argument for this example. Violations are discovered when the `dec` is later applied in the body of `f`. In the case where the parameter does not appear in the body, only a negative result raises the alarm. Consider the following example:

```
> eval(applyF dec 2)
1
```

```
> eval(applyF dec 0)
*** Exception: Assert failed: contract failed:
       the expression Fcontract.hs:12 is to blame.
```

An error is only detected in the second call, though the first call is also wrong.

Static checking can improve software productivity because it detects errors at compile time and reduces the cost of correcting such errors [1]. Static checking avoids runtime overhead but typically involves difficult, often incomplete program analyses. "Theorem prover can be used as an assisting tool for static contract checking" [1]. If the static checking of a program succeeds, it means that the program cannot crash [1]. Static contract checking reports who is to blame and points to the location of the violation. There is still no compiler that supports static automatic verification of high-level languages, since these languages support advanced features (such as higher-order functions, complex recursions, laziness) to help programmers [1]. Dana N.Xu et al. described a sound and automatic static verification tool for Haskell, that is based on contracts and symbolic execution. Their approach returns precise blame assignments at compile-time, in the presence of higher order functions and laziness [1].

In this thesis, a dynamic contract checker for Feldspar is implemented completely and some of the written contracts are presented in chapter five. Static contract checking is not implemented; however, a C static contract checker could be used for checking code generated by the Feldspar language. This is because Feldspar generates C code and the assert function in Haskell translates to the assert function in C. Unfortunately, a static checker for C that works correctly could not be found during this thesis, so we cannot test contracts with a C static checker.

In practice, contract checking improves conditions for using tools for expressing and testing general algebraic properties like QuickCheck; so using QuickCheck for verification of Feldspar can possibly be a lot more effective with the help of contracts. QuickCheck is an automated and random testing tool for Haskell programs [9]. In this thesis QuickCheck is used for testing contracts because dynamic contract checking is not sufficient to prove the correctness of Feldspar programs.

6.1 Testing with QuickCheck

QuickCheck is an automated testing tool that helps Haskell programmers in formulating and testing properties of programs [9]. Program properties are defined via a pre/post, algebraic style or are model-based (functions or relations); note that programmers must specify fixed types for arguments of properties to prevent overload problems because of polymorphic type for QuickCheck. The user can specify random test data generators for more complex data structures with QuickCheck. Properties are passed to the quickCheck function for testing with randomly generated test cases.

Some of the properties that have been written for Feldspar contracts are pre-

sented in the next section, and all properties can be found in Appendix B.

The `freezeVector` function converts a vector to a core array without any change in length and values of the vector; the `thawVector` takes a core array and returns a vector with same length and value of the core array. The `freezeVector` and the `thawVector` contracts that can be found in Appendix A. The former is tested with the following property by calling the quickCheck function. `prop_freezeVector` first converts the input vector to a core array and then this array is transformed to a vector by applying the `thawVector` contract. This vector should be equal to the input vector if these contracts act correctly.

```
prop_freezeVector  :: (Type a,Eq a) =>
                           Vector (Data a) -> Data Bool
prop_freezeVector v =  equal v (Fcontract.thwVctr
                                  (Fcontract.frzVctr v))

>quickCheck prop_ freezeVector
+++ OK, passed 100 tests.
```

The next property that we present here is a property for the `fold1` contract. This function takes the `sum` function as its first parameter.

```
prop_fold1    :: Vector1 Int32 -> Property
prop_fold1 v = (eval (fld1 (+) v) P.== P.foldl (+) 0 (eval v))

>quickCheck prop_fold1
*** Failed! Exception: 'Assert failed: contract failed:
     the expression Fcontract.hs:89(arg#2) is to blame.'
     (after 1 test):
[]
```

The first test case fails because of the contract violation; the precondition of `fold1` is not satisfied. The error message mentions that the argument of the function is to blame.

If a test case of quickCheck fails, the counter example that causes this problem is reported as result. The counter example for `prop_fold1` is a null vector. Note that the QuickCheck is used in this thesis for discovering the bugs in the written contracts to achieve this goal test cases should have correct preconditions. So, a contract is satisfied if functions passes all tests. The QuickCheck property language provides conditional properties; the conditional property C ==> P holds if the property (P) after ==> holds whenever the condition (C) does. Therefore, test cases which do not satisfy the condition are discarded and generating test cases does not stop until 100 test have passed or number of test cases are reached to the overall limitation (1000). In the second case, a message reports the number of test cases that satisfy the condition and the property is said to hold in these cases. The ==> function is defined as follows in Feldspar.

```
(===>) :: Testable prop => Data Bool -> prop -> Property
a ===> b = eval a ==> b
```

The `fold1` property is written as a conditional property:

```
prop_fold1    :: Vector1 Int32 -> Property
prop_fold1  v = notnull v ===>
                  (eval (fld1 (+) v) P.== P.foldl (+) 0 (eval v))

notNull :: Vector1 Int32 -> Data Bool
notNull v = length v > 0

>quickCheck prop_fold1
+++ OK, passed 100 tests.
```

The next property has a condition that is seldom satisfied and after generating 1000 test cases only 28 test cases satisfied the condition and the property.

```
prop_scalarProd :: Vector1 Int32 -> Vector1 Int32 ->Property
prop_scalarProd v1 v2 =(length v1== length v2) ===>
    ((Fcontract.sclrPrd v2 v1) == (Fcontract.sclrPrd v1 v2))

>quickcheck prop_ scalarProd
*** Gave up! Passed only 28 tests.
```

Some of the properties that are written for Feldspar contracts give up their test case because of using conditional properties to satisfy their precondtions. To find bugs and redundancy of conditions, we define incorrect pre/ post conditions for contracts and then call QuickCheck for testing these contracts. All these contracts should fail during testing. If one of the wrong test cases is passed, this illustrates that the conditions of the contract are not defined correctly. The contract `nonEmpty` checks that the given vector is not null so the length of the vector must be bigger than zero. The wrong version of this contract checks the length of the input vector is bigger than or equal to zero. Function `quickCheck` is called for the `fold1` function with the new contract:

```
>quickCheck prop_fold1
+++ OK, passed 100 tests.
```

This test passes all 100 tests because of using the `notNull` condition for generating random test cases. If this condition is removed from the property, the quickCheck function fails.

```
prop_fold1  v = (eval (fld1 (+) v) P.== P.foldl (+) 0 (eval v))

> quickCheck prop_fold1
*** Failed! Exception: 'List.genericIndex: index too large.'
(after 1 test):
[]
```

The next example is the `minimum` contract; the incorrect version of function `mined` is defined as

```
mined :: (Type a,Ord a) => Vector (Data a) -> Data a
mined xs = forLoop (length xs -1) bMin
                           (\i mMin -> min mMin (xs!i))
    where bMin = head xs
```

This function does not consider the last value of the list; quickCheck should fail and report that the function is to blame because the postcondition fails.

```
>quickCheck prop_minimum
*** Failed! Exception: 'Assert failed: contract failed:
       the expression ./Fcontract.hs:96 is to blame.'
(after 3 tests):
[0,-1]
```

Using test cases like these, we have checked our Feldspar contracts and found no bugs. Therefore, we decide to accept that the presented contracts for Feldspar are correct and valid.

Chapter 7

Related Work

Parnas first presented the concept of contracts for software in 1970s. In the next decade, this idea was developed to design a software based on the concept of contracts for an object-oriented programming language (Eiffel) by Meyer. Contracts were implemented for many programming languages (e.g., C, C++, Java and Scheme [13]). Assert functions are popular and practical in C codes but this function does not have enough information to report the violation precisely. "In fact, 60% of the C and C++ entries to the 2005 ICFP programming contest used assertions, despite the fact that the software was produced for only a single run and was ignored afterwards" [16].

Contracts for higher-order functional programming were introduced by Findler and Felleisen [13]. They implemented dynanic contract checking for Scheme. Their implementation contains blame assignment that reports the location of the violation without specifying who is to blame. Blume and McAllester presented a sound and complete model to prove that Findler and Felleisen's dynamic contract checker detects all violations and always specifies blame correctly. After these studies much research was started to answer the questions about the nature of contracts specially the `Any` contract.

R.Hinze et al. [2] used generalised algebraic data types to implement contracts as a library in Haskell. Their implementaion includes contract constructors for pairs, lists, algebraic data types and a combinator for conjunction. They prove that algabric properties of contracts provid the conditions for optimizing contracts and showing that a function satisfies its contract. They added the cause of violation to the blame assignments of Findler and Felleisen's implementation to report the reason for a contract failure precisely at run time. Their work has been the main inspiration for this thesis.

Hybrid contracts combine static and dynamic contract checking to enable program verification and error detection. Hybrid contract checking for scheme is proposed by Flanagan. His implementation discovers a contract failure statically (whenever possible) and dynamically (only when necessary) [1].

Static contract checker for an advanced functional programming language was presented by Dana N. Xu et al. [1]. Their symbolic evaluation follows closely the

lazy semantics of Haskell. They have proved the soundness of each simplification rule and given a proof of the soundness of their static contract checking. Their approach is modular and returns complete and accurate blame assignments at compile time in the presence of higher order functions and laziness [1].

Chapter 8

Conclusions

We have presented the contract comprehension, the dependent function contract, the dependent product contract, the list contract, the And contract and the Any contract for Feldspar (an embedded language).

The `assert` function is implemented for these contracts to check them during run time. To implement the assert function for contracts it is necessary that the Feldspar language has an assert function because the conditional expressions don't work correctly when one of the branches contains an error command. Also, if the assert function in Haskell is converted to the assert function in C, we can check contracts with a C static contract checker during compile time. However, we could not find any C static contract checker that works correctly. The `assert` function was implemented and added to the Feldsapr Core library by the Feldspar developers.

Higher order and first class contracts are introduced and implemented for three of Feldspar's libraries (core array, vector and matrix). Furthermore, we have implemented contracts for some practical examples of Feldspar functions such as discrete cosine transform and low pass filter in this thesis.

The implementation of contracts solves the following problems in Feldspar. First, the C compiler returns a wrong answer instead of returning an exception when an indexed (!) function takes an index bigger than the length of its input vector. For example, a null vector is passed to the `fold1` function. Second, matrix library has conceptual problems, like a matrix can be defined with different number of columns. Also, Feldspar's error message is converted to an informative error message, since the error message that is created with the help of a contract reports the expression that causes the violation.

Blame assignment is implemented by using a C preprocessor (CPP) to report the source location of the program that causes the contract violation. Having blame assignment point to real source locations is one of the future works of Typed Contracts for Functional Programming by Hinze, Jeuring et al.

The implementation of the dynamic contract checker is presented via examples and we have presented several bugs of Feldspar functions that were detected

by dynamic contract checking. Also the contracts written were checked with a testing tool (QuickCheck) to discover bugs in the Feldspar contracts. At the end, incorrect contracts are implemented and tested with QuickCheck; also, wrong arguments are passed to the contracts written by defining wrong conditions for QuickCheck properties. These contracts and test cases are created to ensure correctness and validation of contracts. All contracts pass the above testing methods and we decide to accept that the written contracts for Feldspar are valid and correct.

This thesis left some topics for future work, in particular, the use of template Haskell for reporting the source location as blaming point in the error message; this would solve problems of using a C preprocessor. Furthermore, a static contract checker for Feldspar should be written to allow contract satisfaction checks at compile time.

Bibliography

[1] Dana N. Xu, Peyton J. Simon, K. Claessen, *Static contract checking for Haskell*, ACM, SIGPLAN Not, volume 44, Programming Languages (POPL '09), 2009.

[2] R. Hinze, J. Jeuring, and A. Löh, *Typed Contracts for Functional Programming*. In FLOPS 06: Functional and Logic Programming: 8th International Symposium, pages 208 - 225, Springer LNCS 3945, 2006.

[3] E. Axelsson, K. Claessen, M. Sheeran, J. Svenningsson, D. Engdal, and A. Persson, *The Design and Implementation of Feldspar: an Embedded Language for Digital Signal Processing*, Springer LNCS 6647, 2011.

[4] E. Axelsson , A. Persson, M. Sheeran, J. Svenningsson, G. Deval, *A Tutorial on Programming in Feldspar*. `http://feldspar.inf.elte.hu/feldspar/documents/FeldsparTutorial.pdf`, 2011.

[5] The Haskell Platform. `http://hackage.haskell.org/platform`, March 2012.

[6] The feldspar-language package. `http://hackage.haskell.org/package/feldspar-language`, March 2012.

[7] The feldspar-compiler package. `http://hackage.haskell.org/package/feldspar-compiler`, March 2012.

[8] Steven W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, ISBN 0-9660176-6-8, second Edition,California Technical Publishing, 1999.

[9] K. Claessen, J. Hughes, *Testing Monadic Code with QuickCheck*, ACM, SIGPLAN Not, volume 37, 2002.

[10] All codes of `Typed Contracts for Functional Programming` paper. `http://www.andres-loeh.de/Contracts.html`, March 2012.

[11] K. Wansbrough, *Macros and Preprocessing in Haskell*, third Haskell workshop, Paris, France, 1999. The author's homepage: `http://www.lochan.org/keith/publications/index.html`, March 2012.

[12] Richard M. Stallman, Z. Weinberg, *The C Preprocessor*. `gcc.gnu.org/onlinedocs/cpp.pdf`, 2011.

[13] Robert B. Findler, M. Felleisen, *Contracts for higher-order functions*. In ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pages 48 - 59, New York, NY, USA, ACM Press, 2002.

[14] M. Nyrenius, D. Ramström, *Generating Embedded C Code for Digital Signal Processing*, Master of Science Thesis in Computer Science, Department of Computer Science and Engineering, Chalmers University of Technology, 2011.

[15] Avraham E. Shinnar, *Safe and Effective Contracts*, PhD thesis of Philosophy in the subject of Computer Science, Harvard University, 2011.

[16] Robert B. Findler, M. Blume, *Contracts as Pairs of Projections*. In Functional and Logic Programming, pages 226 - 241. Springer LNCS 3945, 2006.

[17] K. Claessen, J. Hughes, *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, ACM SIGPLAN Notices,35(9). 2000.

[18] Robert B. Findler, M. Blume, M. Felleisen, *An investigation of contracts as projections*, Technical Report TR-2004-02, The University of Chicago, 2004.

Appendix A

Contract.hs

```
 {-# LANGUAGE TypeOperators,FlexibleContexts,TypeFamilies,GADTs,FlexibleInstances #-}
module Contract where

import qualified Prelude as P
import Blame
import Feldspar
import Feldspar.Vector
import Feldspar.Matrix

---------------------------------Contract -------------------------------------
data Contract aT where
    Prop      :: (Syntax aT) => (aT -> Data Bool) -> Contract aT
    Function  :: Contract aT -> (aT -> Contract bT) -> Contract (aT :-> bT)
    Pair      :: (Syntax aT,Syntax bT) => Contract aT ->
                                  (aT -> Contract bT) -> Contract (aT, bT)
    List      :: (Syntax aT) =>  Contract aT -> Contract [aT]
    And       :: Contract aT -> Contract aT -> Contract aT
    Any       :: Contract aT

fun f        =  Fun (\ _ x -> f x)

(=.) :: Locs -> Maybe Int -> Locs
(NegPos neg pos@(p:ps)) =.  v =  NegPos neg  ((p{arg=v}):ps)

applyLoc1  loc f x        = apply f (loc=. (Just 1)) x
applyLoc2 loc f x1 x2     = apply (apply f (loc=. (Just 1))  x1)
                                  (loc=. (Just 2)) x2
applyLoc3 loc f x1 x2 x3  = apply (apply (apply f (loc=. (Just 1)) x1)
                                  (loc=. (Just 2)) x2) (loc=. (Just 3)) x3

---------------------------------Assert----------------------------
assertLoc :: Locs -> Contract a -> a -> a
assertLoc  locs (Prop p)  a           =
                assertMsg  ("contract failed: " P.++ blame locs ) (p a) a
assertLoc  locsf (Function c1 c2) f   = Fun(\ locx -> (\ x' -> (assertLoc locsf  (c2 x') .
                                        apply f  locx) x') .
                                            assertLoc (locsf +> locx) c1 )
assertLoc  locs (Pair c1 c2) (a1, a2) = (\ a1' -> (a1' , assertLoc locs (c2 a1') a2))
                                             (assertLoc locs c1 a1)
assertLoc  locs (List c)  as          =  P.map (assertLoc locs c ) as
assertLoc  locs (And c1 c2)  a        =  (assertLoc locs c2 .assertLoc locs c1 ) a
assertLoc  locs  Any         a        =  a

---------------------------------infix---------------------------------

infixr :->
newtype aT :-> bT  =  Fun { apply :: Locs -> aT -> bT }

infixr 4 >->>
pre >->>   post  =  Function pre (const post)

infixr 4 >>->
pre >>->  post  =  Function pre post
```

```
infixr 9 `o`
f `o` g  =  \a -> f (g a)

infixl 3 &
(&) = And
-------------------------------- functions------------------------
class Functions a
    where
      type Funs a
      functions :: a -> Funs a

instance Functions b => Functions (a ->b)
    where
     type Funs (a->b) = a :-> Funs b
     functions f = fun (\x -> functions (f x))

instance Functions (Data a)
   where
    type Funs (Data a) = Data a
    functions x = x

instance  Functions (Vector a)
   where
    type Funs (Vector a) = Vector a
    functions x = x

instance (Functions a, Functions b) => Functions (a,b)
   where
    type Funs (a,b) = (Funs a, Funs b)
    functions (x,y) = (functions x, functions y)


-------------------------------- Feldspar contracts --------------------------------
nat:: Contract (Data Int32)
nat = Prop(\i -> i >=0)

positive ::  Contract (Data Length)
positive  =   Prop(\  i ->   i < 2000)

---------------------------------contracts for Array library---------------------------

notNullArry :: (Type a) => Contract (Data [a])
notNullArry = Prop(\xs -> (getLength xs)> 0)

validIndex ::Type a=> Data [a]-> Contract (Data Index)
validIndex = (\xs -> Prop (\i -> (getLength xs) > i))

setLenArray :: Type a =>  Data Length -> Contract (Data [a])
setLenArray  = (\len -> Prop(\a -> ((getLength a)>=len )))

validLenArray :: (Type a) => Data [a] ->  Contract (Data [a])
validLenArray =  (\a1 ->
                   Prop(\a2 -> (getLength a1) == (getLength a2)))

---------------------------------contracts for Vector library-----------------------------

nonEmpty :: Syntax a => Contract (Vector a)
nonEmpty =  Prop(\xs ->(length xs)> 0)

validLen :: (Syntax a ) => Data Length -> Contract (Vector a)
validLen = (\len -> Prop (\v -> (length v == len)))

bound ::Vector a -> Contract (Data Index)
bound =(\xs -> Prop (\i -> (length xs) > i))

validLenVector :: (Syntax a) => Vector a ->  Contract (Vector a)
validLenVector =  (\v1 -> Prop(\v2 -> (length v1) == (length v2)))

validLenV :: (Syntax a,Syntax b) => Vector a ->  Contract (Vector b)
validLenV =  (\v1 -> Prop(\v2 -> (length v1) == (length v2)))

reversed :: (Type a,Eq a) =>
                  Contract (Vector (Data a) :-> Vector (Data a))
```

```
reversed = Any >>-> \xs -> Prop(\rxs -> (isReversed xs rxs))

isReversed      :: (Type a,Eq a) =>
                        Vector (Data a) -> Vector (Data a) -> Data Bool
isReversed xs ys= (lenx == leny)&&
                     forLoop lenx true
                         (\i result-> result && (xs!i ==ys!(leny-1 -i)))
    where lenx = length xs
          leny = length ys

get :: (Type a,Eq a) => Data Index -> Data a -> Contract(Data [a])
get = (\i a -> Prop(\rs-> (getIx rs i)== a))

equal       :: (Type a,Eq a) =>
                   Vector (Data a) -> Vector (Data a) -> Data Bool
equal xs ys =  (length xs == length ys) &&
                     fold (&&) true (zipWith (==) xs ys)


maxContract :: (Eq a,Ord a) =>Contract (Vector (Data a):-> (Data a))
maxContract = nonEmpty >>-> \xs -> Prop(\m -> (maxed xs) == m)

maxed :: (Type a,Ord a) => Vector (Data a) -> Data a
maxed xs = forLoop (length xs) bMax (\i mMax -> max mMax (xs!i))
      where bMax = head xs

minContract :: (Eq a,Ord a) =>
                      Contract (Vector (Data a):-> (Data a))
minContract = nonEmpty >>-> \xs -> Prop(\m -> (mined xs) == m)

mined :: (Type a,Ord a) => Vector (Data a) -> Data a
mined xs = forLoop (length xs) bMin (\i mMin -> min mMin (xs!i))
      where bMin = head xs

sameLength :: Type a => Vector (Data a) -> Contract (Data [a])
sameLength =  (\v -> Prop (\a -> (length v) == (getLength a)))

sameLen :: Type a => Data [a] -> Contract (Vector (Data a))
sameLen =  (\a -> Prop (\v -> (length v) == (getLength a)))

---------------------------------contracts for Matrix library ---------------------------------

isMatrix :: Type a => Contract ( Matrix a)
isMatrix = Prop(\xs -> ismatrix xs)

ismatrix :: Matrix a -> Data Bool
ismatrix xs = (length numColumn >0) && fold (&&) true
                              (map (== head numColumn) numColumn )
     where numColumn = map length xs

validInput ::Type a => Contract(Data [[a]])
validInput = Prop(\aa -> isValid aa)

isValid :: Type a=> Data [[a]] -> Data Bool
isValid aa = forLoop (getLength aa) true
                  (\i result -> result && (getLength (aa!i))== col)
       where col = getLength(aa!0)

isTranspose :: (Type a,Eq a)=> Matrix a -> Contract(Matrix a)
isTranspose =  \m -> Prop(\ tm -> condition(length tm ==0)
                                  true ((match m tm) && (match tm m)))

match ::(Type a,Eq a)=> Matrix a -> Matrix a -> Data Bool
match  mx1 mx2 =  (length (head mx1)) ==(length mx2)

isMatch :: (Eq a ,Numeric a) => Matrix a -> Contract(Matrix a)
isMatch = (\mx1 -> Prop(\mx2 -> (match mx1 mx2) ))

boundry :: Type a => Data Length -> Data Length ->Contract (Data [[a]])
boundry = \row col -> Prop (\a -> (getLength a >= row)&&
                                  (getLength (a!0) >= col))

diagonalPrp :: (Type a) => Contract (Matrix a)
diagonalPrp = Prop (\mx -> (ismatrix mx) && length(head mx) >= length (mx))
```

```
notNullMatrix :: (Type a) => Contract (Matrix a)
notNullMatrix = Prop (\mx -> (length mx == 0) ?
                  (false ,((length(head mx)==0)? (false, true))))

---------------------------------search-------------------------
validOutput :: Vector1 a -> Contract (Data Index)
validOutput = (\v ->
                 Prop (\i-> (i< length v) || (i == length v+100)))


---------------------------------Fft ----------------------------

isPow2 :: Contract (Vector1 (Complex Float))
isPow2 = Prop(\v -> ispow2 (length v))

ispow2 :: Data Length -> Data Bool
ispow2 x = x .&. (x-1) == 0
```

Fcontract.hs

```
{-# LANGUAGE CPP,TypeOperators,GADTs,FlexibleContexts ,NoMonomorphismRestriction #-}
module Fcontract where
#include "assert.h"

import Control.Arrow ((&&&))
import Contract
import qualified Prelude as P
import Blame
import Feldspar as F
import qualified Feldspar.Core
import Feldspar.Vector as V
import Feldspar.Matrix as M
import Feldspar.Compiler --hiding(setLength)
import FFT as T

---------------------------------vector -------------------------------------
freezeVector:: Type a => Vector (Data a) :-> Data [a]
freezeVector = assert (Any >>-> (\v -> sameLength v)) (functions V.freezeVector)
frzVctr      = apply1 Fcontract.freezeVector

thawVector :: Type a => Data [a] :-> Vector (Data a)
thawVector = assert (Any >>-> (\a -> sameLen a)) (functions V.thawVector)
thwVctr      = apply1 Fcontract.thawVector

indexed :: (Syntax a) =>  Data Length :-> (Data Index -> a) :-> Vector a
indexed = assert (positive >>-> (\ len -> Any >->> (validLen len))) (functions V.indexed)
indxd   = apply2 Fcontract.indexed

scalarProd :: (Numeric a,Eq a) => Vector (Data a) :-> Vector (Data a) :-> Data a
scalarProd = assert (Any >>-> (\v1 -> validLenVector v1 >->> Any)) (functions V.scalarProd)
sclrPrd    = apply2 Fcontract.scalarProd

head :: Syntax a => Vector a :-> a
head = assert (nonEmpty >->> Any) (fun (\x -> (V.head x)))
hd   = apply1 Fcontract.head

last :: Syntax a => Vector a :-> a
last =  assert (nonEmpty >->> Any) (fun (\x -> (V.last x)))
lst  =  apply1 Fcontract.last

tail  :: Syntax a => Vector a :-> Vector a
tail  =  assert  (Any >->> Any) (functions V.tail)
tl    =  apply1 Fcontract.tail

maximum  :: (Eq a,Ord a) => Vector (Data a) :-> Data a
maximum  =  assert  maxContract (functions V.maximum )
mxmm     =  apply1 Fcontract.maximum

minimum  :: (Ord a,Eq a)=> Vector (Data a) :-> Data a
minimum  =  assert  minContract (functions  V.minimum )
```

```
mnmm     =  apply1 Fcontract.minimum

reverse  ::(Eq aT,Type aT) => Vector (Data aT) :-> Vector (Data aT)
reverse  = assert  reversed (functions V.reverse)
rvrs     = apply1 Fcontract.reverse

fold1 :: (Type a) => (Data a -> Data a -> Data a) :-> Vector (Data a)  :-> Data a
fold1 = assert (Any >->> nonEmpty >->> Any) (functions V.fold1 )
fld1  = apply2 Fcontract.fold1

replicate :: Data Length :-> a :-> Vector a
replicate = assert (positive >->> Any >->> Any) (functions V.replicate)
rplct     = apply2 Fcontract.replicate
---------------------------------array ----------------------


setLength :: Type a =>  Data Length :->Data [a] :-> Data [a]
setLength = assert  (positive >>-> (\len -> setLenArray len >->> Any))  (functions F.setLength)
stLngth   = apply2 Fcontract.setLength

getIx     :: Type a=> Data [a] :-> Data Index :-> Data a
getIx     = assert (nullarray >>-> (\xs -> validIndex xs >->> Any))  (functions F.getIx)
gtx       = apply2 Fcontract.getIx

setIx     :: (Type a,Eq a) => Data [a] :-> Data Index :-> Data a :-> Data [a]
setIx     = assert (Any >>->(\xs -> validIndex xs >>->(\i -> Any>>-> (\a -> (get i a)))))  (functions F.setIx)
stx       = apply3 Fcontract.setIx

---------------------------------matrix-----------------------

freezeMatrix :: Type a => Matrix a :-> Data [[a]]
freezeMatrix = assert  (isMatrix >->> Any) (functions M.freezeMatrix)
frzMtrx      = apply1 Fcontract.freezeMatrix

thawMatrix :: Type a => Data [[a]] :-> Matrix a
thawMatrix =  assert (validInput>->>isMatrix)  (functions  M.thawMatrix )
unfrzMtrx      =  apply1 Fcontract.thawMatrix

transpose     :: (Type a,Eq a) => Matrix a :-> Matrix a
transpose     = assert  (Any >>-> (\m -> isTranspose m))  (functions M.transpose )
trnsps        = apply1 Fcontract.transpose

mulMat        :: (Eq a ,Numeric a) => Matrix a :-> Matrix a :-> Matrix a
mulMat        = assert  (isMatrix >>-> (\mx -> (And isMatrix (isMatch mx)) >->> Any)) (functions M.mulMat)
mlMt          = apply2 Fcontract.mulMat

diagonal     :: Type a => Matrix a :-> Vector (Data a)
diagonal      = assert((And notNullMatrix diagonalPrp) >->> Any)(functions M.diagonal)
dgnl          = apply1 Fcontract.diagonal

indexedMat :: Type a => Data Length :-> Data Length :-> (Data Index -> Data Index -> Data a):-> Matrix a
indexedMat = assert(positive>->> positive >->> Any >->> isMatrix)(functions M.indexedMat)
indxdMt = apply3 Fcontract.indexedMat

---------------------------------test--------------------
-- connot use Fcontract.ScalarProd because of inits have different length vectors
convolution :: Vector1 Float -> Vector1 Float -> Vector1 Float
convolution kernel input = map ((V.scalarProd kernel) . rvrs) $ inits input

convolutionC :: Vector1 Float :-> Vector1 Float :-> Vector1 Float
convolutionC = assert (Any >->> Any >>-> (\v -> validLen (length v +1)))(functions convolution)
cnvltn       = apply2 convolutionC


---------------------------------sort--------------------------------
-- Bubble sort (permute)

bubbleSort :: Vector1 Int32 -> Vector1 Int32
bubbleSort v = forLoop len v inner
  where len = length v
        inner i nv = forLoop (len-1) nv bubble
        bubble j nv' =(nv'!j > nv'!(j+1)) ?((V.thawVector $ swap (V.freezeVector nv') j (j+1)), nv')
        swap a i1 i2 = F.setIx (F.setIx a i1 (F.getIx a i2)) i2 (F.getIx a i1)
```

```
ordered    ::  (Type a,Ord a) => Vector1 a -> Data Bool
ordered v = fold (&&) true (V.zipWith (<=) v (V.tail v))

sort = Prop(\sx ->  ordered sx)

bubblesort :: Vector1 Int32 :-> Vector1 Int32
bubblesort = assert (Any >->> sort) (functions bubbleSort)
bbblsrt    = apply1 Fcontract.bubblesort

bubbleSort2 :: Vector1 Int32 -> Vector1 Int32
bubbleSort2 v = forLoop len v inner
 where len = length v -1
       inner i nv = forLoop (len-2) nv bubble
       bubble j nv' = (nv'!j > nv'!(j+1)) ? (swap2 nv' j (j+1), nv')
       swap2 v i1 i2 = permute (\l i -> (i == i1) ? (i2, (i == i2) ? (i1, i))) v

bubblesort2 :: Vector1 Int32 :-> Vector1 Int32
bubblesort2 = assert (Any >->> sort) (functions bubbleSort2)
bbblsrt2    = apply1 Fcontract.bubblesort2

---------------------------------search ------------------------------
--linear search

search ::(Ord a,Eq a) =>  Vector1 a -> Data a -> Data Index
search v key = forLoop (len) (len+100) (\i s -> (list1! i)? (i,s))
    where list1 = (map (==key) v)
          len  = length list1

searchCnt :: (Ord a,Eq a) =>  Vector1 a :-> Data a :-> Data Index
searchCnt = assert (Any >>-> (\v ->Any >->> validOutput v)) (functions search)
srchCnt   = apply2 Fcontract.searchCnt

-- binary search

binarySearch ::  Data WordN -> Vector1 WordN -> Data Index
binarySearch key v = fst $ forLoop iters (0,len-1) f
      where len = length v
            iters = ceiling $ logBase 2 $ i2f len
            f _ (low,high) =(v!d == key) ? ((d, d), ((key < v!d) ? ((low, d-1), (d+1, high))))
               where d = (low + high) `div` 2

searchB ::  Data WordN :-> Vector1 WordN :->  Data Index
searchB = assert (Any >->> sort >->> Any) (functions binarySearch)
srchB   = apply2 searchB

--------------------------------- DCT ---------------------------------

-- Discrete Cosine Transform type 2
dct2 :: (Vector1 Float) -> (Vector1 Float)
dct2 xn = mat *** xn
    where
      mat = M.indexedMat (length xn) (length xn) (\k l -> dct2nkl (length xn) k l)

dCT2      :: (Vector1 Float) :-> (Vector1 Float)
dCT2      = assert (Any >>-> (\v -> validLenVector v))(functions dct2)
applydct2 = apply1 dCT2

-- Helper function defining all the values in the DCT-2n matrix
dct2nkl :: Data Length -> Data WordN -> Data WordN -> Data Float
dct2nkl n k l = cos ( (k' *(2*l' +1)*pi)/(2*n') )
  where
    n' = i2f n
    k' = i2f k
    l' = i2f l

dct2NKL :: Data Length :-> Data WordN :-> Data WordN :-> Data Float
dct2NKL = assert (positive >->> Any >->> Any >->> Any)(functions dct2nkl)
applydct2nkl = apply3 dct2NKL

--------------------------------- Low-pass filter ----------------------------------

lowPassCore :: (Numeric a) => Data Index -> Vector1 a -> Vector1 a
lowPassCore k v = take k v ++ V.replicate (length v - k) 0
```

```
lPassCore ::(Numeric a) => Data Index :-> Vector1 a :-> Vector1 a
lPassCore = assert (Any >>->(\i -> validIndexs i >>-> (\v -> validLenVector v)) ) (functions lowPassCore)
lwPssCr = apply2 lPassCore

lowPass :: Data Index -> Vector1 Float -> Vector1 Float
lowPass k = frequencyTrans (lwPssCr k)

frequencyTrans :: (Vector1 (Complex Float) -> Vector1 (Complex Float))
               -> Vector1 Float
               -> Vector1 Float
frequencyTrans innerFunction v = map realPart $ ifft
                                 $ innerFunction
                                 $ fft $ map (\a -> complex a 0) v

---------------------------------- FFT -----------------------------------

ffT :: Vector1 (Complex Float) :-> Vector1 (Complex Float)
ffT = assert(isPow2 >->> Any ) (functions fft)
applyfft = apply1 ffT
```

Appendix B

```
{-# LANGUAGE CPP,TypeOperators,FlexibleContexts,TypeFamilies,GADTs,FlexibleInstances,UndecidableInstances,NoMonomorphismRestric
module Test where
#include "assert.h"

import Contract
import Blame
import Feldspar as F
import qualified Prelude as P
import Feldspar.Vector as V
import Test.QuickCheck
import Fcontract as C
import Feldspar.Matrix as M
import System.Random
import Insrtsort

------------------------------Generate Length------------------------------
deriving instance Random Length

instance Arbitrary (Data Length) where
     arbitrary = natural

natural :: Gen (Data Length)
natural = do i <- choose(0,2000)
             return (value i)

--------------------------Helper Functions ------------------------------------------------------------------------

validIndexs ::(Type a) => Data Index -> Data Index -> Contract(Vector (Data a))
validIndexs = (\r c -> Prop (\v -> ((r^2)*(c^2) )== (length v)))

nullMat mx = (length mx == 0) ? (false ,((length(V.head mx)==0)? (false, true)))
integer n = n >= 0
notnull v = (length v) > 0
notempty xs = (getLength xs >0)
correctIndex xs i = (F.getLength xs > i)&& (i>= 0 )
notnullMatrix mx = condition (length mx == 0) false (notnull (V.head mx))


equalArray :: (Type a,Eq a)=>Data [a] -> Data [a] -> Data Bool
equalArray ax ay = (getLength ax == getLength ay) ?(checkNull , false)
    where checkNull = (getLength ax == 0) ? (true, (checkEq ax ay))
          checkEq ax ay = forLoop (getLength ax) true (\i result -> result && (F.getIx ax i)== ( F.getIx ay i))

prop_eqArray       :: Data [Int32] -> Data [Int32] -> Data Bool
prop_eqArray a1 a2 = equalArray a1 a2 == equalArray a2 a1

---------------------------- Vector Library-----------------------------------------------------------------------
prop_freezeVector :: Vector (Data Int32) -> Data Bool
prop_freezeVector  v  =  equal v (apply1 C.thawVector(apply1 C.freezeVector v))

xx v = equal v (apply1 C.thawVector(apply1 C.freezeVector v))

prop_thawVector :: Data [Int32] -> Data Bool
prop_thawVector  v  =  equalArray v (C.frzVctr(C.thwVctr v))
```

```
prop_indexed :: Data Length -> Data Bool
prop_indexed i  = (length (apply2 C.indexed i id) == i)

prop_scalarProd :: Vector1 Int32 -> Vector1 Int32 -> Property
prop_scalarProd v1 v2 =  (length v1 == length v2) ===> ((apply2 C.scalarProd v2 v1) == (apply2 C.scalarProd v1 v2))

prop_head :: Vector1 Int32 -> Property
prop_head xs = notnull xs ===> (C.hd xs == (!) xs 0)

prop_last :: Vector1 Int32 -> Property
prop_last xs = notnull xs ===>(C.lst xs == C.hd (C.rvrs xs) )

prop_tail1 = eval (V.tail :: Vector1 Int32 -> Vector1 Int32) === eval (V.drop 1 :: Vector1 Int32 -> Vector1 Int32)

prop_maximum    :: Vector1 Int32 -> Property
prop_maximum xs = notnull xs ===> ( C.mxmm xs == C.lst (insertionSort xs))

prop_minimum    :: Vector1 Int32 -> Property
prop_minimum xs = notnull xs ===> (fld1 (&&) (V.map ( >= (mnmm xs)) xs))

prop_reverse    :: Vector1 Int32 -> Data Bool
prop_reverse xs = equal xs $ V.reverse $ V.reverse xs

prop_fold1   :: Vector1 Int32 -> Property
prop_fold1 v = notnull v ===> (eval (fld1 (+) v) P.== P.foldl1 (+)(eval v))


prop_replicate       ::Data Length -> Vector1 Int32 -> Data Bool
prop_replicate len a = ((length rpV  == len) && equalAll)
   where rpV   = apply2 C.replicate len a
         equalAll=  forLoop (length a) true (\i result -> result && equal a (rpV! i) )

---------------------------- Core.Array Library-----------------------------------------------------------------------

prop_setLength        :: Data Length -> Data[Int32] -> Property
prop_setLength len xs = (len <= getLength xs )===> ( (len == getLength  output) && equality)
   where
         output = F.setLength len xs
         equality = forLoop len  true (\i result -> result && (output !i == xs!i))

prop_getIx      :: Data[Int32] -> Data Index -> Property
prop_getIx xs i = (correctIndex xs i && notempty xs)===>( ( C.gtx xs i) == xs!i)


prop_setIx          :: Data[Int32] -> Data Index -> Data Int32 -> Property
prop_setIx xs i key = (correctIndex xs i && notempty xs)===> (( output! i) == key  && before && after)
  where  output= F.setIx xs i key
         before = forLoop i true (\j result -> result && (output !j == xs!j))
         after = forLoop (getLength xs -(i+1)) true (\j result -> result && (output !(j+ index) == xs!(j+ index)))
             where index = (i+1)

---------------------------- Matrix Library--------------------------------------------------------------------------

prop_freezeMatrix    :: Matrix Int32 -> Property
prop_freezeMatrix mx =  notnull mx ===> equalMatrix (apply1 C.thawMatrix(apply1 C.freezeMatrix mx)) mx
  where types = mx :: Matrix Int32

equalMatrix mx1 mx2 = (length mx1 == length mx2)? (notNull,false)
  where notNull = ( length mx1 == 0) ?(true, iteration mx1 mx2)
iteration mx1 mx2 = forLoop (length mx1) true (\i result -> result && equal (mx1 ! i)(mx2 ! i))

prop_transpose    :: Matrix Int32 -> Property
prop_transpose mx = ( ismatrix mx) ===>(equalMatrix mx $ C.trnsps $ C.trnsps mx)

prop_diagonal     :: Matrix Int32 -> Property
prop_diagonal mx  = (ismatrix mx && nullMat mx) ===> (length mx == length (C.dgnl mx))
prop_indexedMat :: Data Length -> Data Length -> Property
prop_indexedMat r c = (r<2000 && r > 0 && c >0 && c < 2000 ) ===> ( (length newMt == r) && (length (C.hd newMt) == c) && ismatr
            where f     = (\i j -> i)
                  newMt = C.indxdMt r c f
---------------------------- Sort-----------------------------------------------------------------------------------
prop_bubblesort   :: Vector1 Int32 -> Data Bool
prop_bubblesort v =  equal (apply1 C.bubblesort v) (insertionSort v)
```

```
----------------------------- LowPassCore--------------------------------------------------------------------------

prop_lowPassCore i v = ((length result == length v ) && (equal (take i result)(take i v)))
  where types  = v :: Vector1 Int32
        result = apply2 lPassCore i v
```