Implementing Data Fields in Haskell
Implementation av datafält i Haskell

Jonas Holmerin

November 1, 1999

Abstract

Implementing data fields in Haskell

Indexed data structures such as arrays are basic building blocks in many applications. The data field model is a general semantic model which provides an abstract and homogeneous way of describing such structures and operations on them. A data field is a generalized indexed structure, and is as such a function with explicit information about its domain. Data fields can be seen as partial functions, and this intuition is used to guide the semantics for the operations on data fields.

This model for indexed data structures is functional in its design, and is thus natural to include in a functional language. In this thesis, we design and implement a dialect of the functional language Haskell which has been extended with data fields. Since the implementation for practical reasons had to be based on an existing Haskell system, we also evaluate different Haskell system to see how suited they are for such extensions.

Referat

Implementation av datafält i Haskell

Indexerade data strukturer som till exempel arrayer är viktiga byggstenar i många programmeringspråk. Datafältsmodellen är en generell semantisk modell som ger ett abstrakt och homogent sätt att beskriva sådana strukturer och operationer på dem. Datafältsmodellen generaliserar indexerade strukturer genom att betrakta dem som funktioner med explicit information om domänen. Ett sådant datafält kan ses som en partiell funktion, och denna intuition används för definiera semantiken för de olika operationerna på datafält.

Datafältsmodellen är i grunden funktionell, och det är därför naturligt att inkludera den i ett funktionellt språk. I detta examensarbete beskriver vi designen och implementationen av en Haskell-dialekt som utvidgats med datafält. Eftersom implementationen av praktiska skäl måste baseras på ett existerande Haskell-system, så utvärderar vi också olika Haskell-system med avseende på hur lämpade de är för detta.

Preface

This thesis describes my Master's project in computing science on the First Degree Programme in Mathematics and Computer Science at Stockholm University. The work was done at the Department of Teleinformatics (IT), the Royal Institute of Technology. Supervisors at Nada (The Department of Numerical Analysis and Computing Science),and IT were Karl Meinke and Björn Lisper, respectively. Examiner was Johan Håstad.

I would like to thank my supervisors, especially Björn Lisper, whose work was the basis for this Master's project. I would also like to thank Karl-Filip Faxén for useful discussions on the implementation of functional languages, and Niklas Röjemo for writing the Haskell compiler NHC, and for allowing me to use it as a basis for implementing our Haskell dialect.

Contents

Chapter 1

Introduction

The indexed data structure is a basic building block in many programming languages. An indexed data structure is a collection of data which can be indexed (explicitly or implicitly) to retrieve a certain value. The "canonical" examples of indexed data structures are the array and the linked list, where the indexing is explicit and implicit, respectively. More complex examples are hash tables and trees. In most languages computations on indexed data structures are expressed by explicit iteration or recursion over the data structure, but some languages, such as APL [9], and many data parallel languages also provide operations which operate directly on indexed data structures.

In the data parallel programming paradigm, parallelism is achieved by carrying out operations in parallel on indexed data structures. This programming paradigm is closely associated with SIMD parallel computers, and some data parallel languages are tied very tightly to a specific SIMD parallel architecture, typically providing parallel arrays where the elements are distributed over the processors and indexed by processor number. Since these arrays are manipulated in parallel, operations which operate on the entire structure are provided. Examples of languages with this programming model are *Lisp and C* [47, 46] which are used to program the Connection Machine [45]. Some data parallel languages are more abstract, however. Fortran 90 [7], High Performance Fortran (HPF) [21] and Sisal [12] all take a more high-level view of data parallel arrays.

The *data field model* is intended to capture the essence of indexed data structures. It provides an abstract view of indexed data structures by viewing them as partial functions.

In this thesis we describe the design and implementation of a dialect of the functional language Haskell with extensions based on the data field model. The goal of the design has been to provide more general ways of operating on such structures, thus relieving the programmer from bothering about the details of specific data structures. The goal of the implementation was that it should be simple and extensible, the performance was less of a concern. Since we choose to base our implementations on an existing Haskell system, we look at the different Haskell systems and give our impression of how suited they are for extending with data fields.

1.1 Operations on indexed data structures

In this section we describe some common classes of operations on indexed data structures. We will also describe some different notations for the operations. The most basic operation is of course indexing, and we will write $C(i)$ for the value at index i in the indexed data structure C.

Elementwise applied operations

Elementwise application is the operation of applying a function to the elements of one or more indexed data structures. An example of an elementwise applied operation is the multiplication of all elements in a vector v with a scalar a. The result of this is a vector v' where $v'(i) = av(i)$. This can be generalized for operations which take more than one argument. The elementwise addition of A and B result in a structure C where where $C(i) = A(i) + B(i)$. More generally, for an operation f which takes k arguments and indexed data structures $A_1, \ldots, A_k$ we can create the collection C' where $C'(i) = f\ A_1(i) \ldots A_k(i)$.

There are several possible notations for elementwise applied operations. One possibility is to have special elementwise applied versions of each "scalar" operation. In *Lisp, for example, elementwise applied operations are suffixed with "`!!`". Thus the elementwise sum of `A` and `B` is written

```
(+!! A B).
```

Of course, this notation does not make it possible to use other than the built in operations as elementwise applied operations in this convenient way.

Another possibility is to *overload* the scalar operator, and let `+` denote both the scalar operation and the elementwise applied operation. In this notation, the elementwise addition of `A` and `B` is simply written

```
A + B
```

Overloaded operators of the kind described above is available in e.g Sisal and Fortran 90. In Fortran 90 this kind of overloaded operators are called *elemental intrinsics*. Usually only a small set of built-in operations are provided as overloaded functions. It might be possible to provide a more general form of elemental intrinsic overloading, however [48]. Yet another possibility is to use some form of "quantification" over the indices. The usual for-loops in C and Pascal are examples of this notation, but these imply sequential execution. An example of a quantifying notation with a parallel implementation is the `for`-notation in Sisal:

```
for i in 1,n returns array of A[i]+B[i]
```

Another quantifying notation which has a parallel implementation is the `FORALL` statement in HPF.

Another notation often used in functional languages is to use *higher order functions.* In Haskell (see section 1.3) for example, the elementwise application of any function `f` on a list `l` can be written:

```
map f l
```

This can be generalized to more arguments with the `zipWith` family of functions. For example, the elementwise addition of two lists `a` and `b` can be written

```
zipWith (+) a b
```

Elementwise applied operations with several arguments raises some semantical questions. Suppose we add two arrays `A` and `B` where `A` allows indices in the range $[0, 5]$ and `B` allows indices in the range $[2, 8]$. This is no problem if we use some quantifying notation (since the indices are explicit), but for the other notations we have some questions to answer. Should elementwise application of operation on data structures with different range be allowed at all? If we allow it, what should the range of the result be? If we as above sum two lists using `zipWith`, lists of different length are allowed, and the result is the same as if the longer list would be truncated to the length of the shorter list, and then the lists were added. In Sisal 2.0 and Fortran 90, *conformance* of the arrays is required, i.e one-dimensional arrays must have the same length, two-dimensional arrays must have the same length in both dimensions, etc. The arrays are aligned and then added elementwise. This still leaves open what the index range of the result should be. There are several alternatives. We could require it to be explicitly given by the programmer, or we could give the resulting array an index range starting on `1` regardless of the operand (this approach is taken in Sisal). In an imperative language we have another alternative. Since we usually store the result in some array, the resulting array should have the same index range as this array. Take the expression

```
C := A + B
```

Here the range of the array resulting from the addition of `A` and `B` is the same as the range of `C`. This approach is taken in Fortran 90.

Communication operations

Operations which reorder data structures can be viewed as *communication* in the data parallel model where indices correspond to processors, since the reordering corresponds to transfer of elements between processors. The two main communication primitives are *get communication* and *send communication*. Get communication from a data structure `A` defines a new data structure for which the value at index i is `A`($source(i)$), where *source* is some function (or data structure). This can be interpreted as a *parallel read*, where each processor reads an element from another processor. Get communication can be be expressed in HPF as

```
FORALL(I=1:N) B(I) = A(SOURCE(I))
```

The overloaded syntax is `B = A(SOURCE)`, and is supported by HPF and Fortran 90.

Send communication is in a sense the inverse of get communication. In send communication, processor i sends its element of the data structure `A` to the processor $dest(i)$, where it is received into the local element of the data structure `B`. In the `FORALL` notation, we can write

```
FORALL(I=1:N) B(DEST(I)) = A(I)
```

If $dest(i) = dest(j)$ for some $i \neq j$, two different values should be stored at the same address, which is a *write conflict*. This can be resolved by somehow *combining* the elements sent to the same address (e.g adding them or taking the maximum), and storing the combined value. This is sometimes referred to as a *combining send*.

Selection

Selection operations selects part of a data structure. *Projection* operations on arrays selects sub-arrays of lower dimension. The simplest example is indexing of an array, where `A(i)` selects the element at index i (which has dimension zero), but we also have projection operations which selects rows or columns of matrices, such as `A(1,:)` in Fortran 90 which selects the first row of the matrix `A`.

Restriction operations selects some subset of the data structure, but retains the dimension. An example is the `filter` function in Haskell, where

```
filter p l
```

denotes the list which contains all the elements in `l` for which the predicate `p` is *true*. Another example is operations which selects some subrange of an array, e.g

```
A(I:J)
```

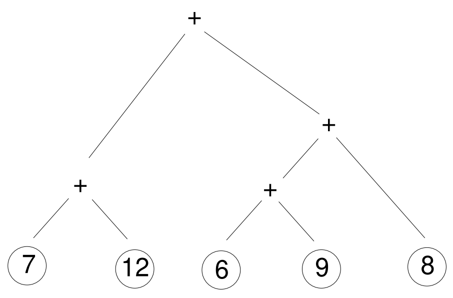

Figure 1.1: Computing the sum of all elements in a data structure.

selects the sub-array of `A` ranging from `I` to `J`.

Restrictions can also be seen as more temporary constructions, where the restriction selects the "active" part of an array where some operation is applied. An example is the range specification in the HPF `FORALL`-statement, as in

```
FORALL(K=I:J) C(K) = A(K) + B(K)
```

Where the "active" parts of `A`, `B` and `C` are the sub-arrays ranging from `I` to `J`. In the low-level data parallel model, where data are indexed by processor number, restriction can be seen as selecting the processors which should be active for a certain operation with a boolean "mask".

Replication

Replication operations create larger data structures by replicating smaller data structures. For example, the creation of an array where each element is a copy of a given element is a replication operation. This can be done in Sisal with the operation `array_fill`. In languages with overloaded elementwise applied operations, we are often allowed to write

```
C = A + 42
```

When `A` and `C` are arrays, the result of this statement is to add `42` to each element of `A` and store the result in `C`. This can be seen as if we first *promote* `42` to an array with the same range as `A`, filled with the value `42` and then add this array elementwise to `A`.

We can also allow replication of higher dimensions, e.g the creation of a matrix with copies of some vector as columns (or rows).

Reduction

Reduction operations are operations composed by applying some function (usually a binary operation) repeatedly to the elements of a data structure.

A typical example is the summation of all elements in an data structure. Here the repeatedly applied binary operation is addition. Examples of reduction operations on lists are the `foldr` and `foldl` functions in Haskell. To sum the elements of a list `l`, we write

```
foldr (+) 0 l
```

If the operation is associative and commutative, the reduction can be performed in any order (e.g the sum of the list `[1,2,3]` can be computed as `(1+2)+3` or `1+(2+3)` or `(2+3)+1`). But if not, the reduction must be performed according to some ordering of the elements. For one-dimensional arrays and lists there are natural orderings, but the situation is less clear for multi-dimensional arrays and other more complex indexed structures.

Reduction with an associative operation can be implemented efficiently in parallel if we perform the reduction according to a balanced binary tree, see Figure 1.1.

Related to reduction operations are *scan* operations, which works like reductions except that the result of a scan is a data structure of all partially reduced values. An example is the `scanl` and `scanr` functions in Haskell, where

```
scanl (+) 0 l
```

will produce a list sums of all prefixes of `l` (including the empty list).

1.2 The data field model

In this section, and the rest of this thesis, we take a *functional* view of indexed data structures. In functional languages, computation is achieved by evaluating expressions to their values, rather than by executing commands (languages where the computation is achieved by executing commands are called *imperative*). In a (pure) functional language, we have no destructive updates, or other side effects. *Functions* are prevalent in functional languages (thus the name *functional*), and a computation which would be performed by a `while` or `for`-loop in a imperative program is typically expressed by a recursive function in the corresponding functional program. Functional languages usually have some form of λ-*abstraction*, i.e some syntax for writing function-valued expressions without giving the functions explicit names. To write the function which adds one to its argument using λ-abstraction, we write

$$\lambda x.x + 1$$

In this section, we will use an informal functional language with function definitions, λ-abstractions, `if` expressions and the usual arithmetic operations.

In our functional view of data structures, we view indexed data structures as functions with explicit information of their domain. This is the basic idea of the *data field model.* A *data field* is a pair (f, b) of a function f and a *bound* b, which has an interpretation as a predicate $[\![b]\!]$ (the term data field is borrowed from the language Crystal [53]). Take the following array in C:

```
int A[5] = {6,7,8,0,1};
```

The data field view of `A` is as a pair $(f, b)^1$, where f is a function defined on the integers $\{0, \dots, 4\}$, and the value of $f(0) = 6$, $f(1) = 7$, etc, and $[\![b]\!]\,(x)$ is true iff $x \in \{0, \dots, 4\}$

A data field (f, b), can be interpreted as a partial function $[\![(f, b)]\!]$ by using $[\![b]\!]$ a restriction of the domain of f. Thus $[\![(f, b)]\!]$ is f restricted to the set for which $[\![b]\!]$ is *true.* The interpretation of data fields as partial functions is used to guide the semantics for operations on data fields. The idea is that one should be able to think as data fields as partial functions and define them in a functional fashion.

To explain this in more detail we need some concepts. Let the value "$\perp$" stand for "non-termination". To see what this means, let *bot* be defined as follows:

$$bot = bot$$

The evaluation of *bot* will never terminate, so we say the value of *bot* is $\perp$.

Now consider the following expression:

$$(\lambda x.x + 1)\, bot$$

The evaluation of this expression will not terminate, since $+$ need to evaluate both its arguments. Functions such as $\lambda x.x + 1$ which has the property $f \perp = \perp$, are called *strict.*

A *partial function* is a function for which $f\,x = \perp$ for some $x \neq \perp$. The *domain* of f, $dom(f)$, is the set $\{x \mid f\,x \neq \perp\}$

Using $\perp$ we can define a restriction operation "$\backslash$" as follows:

$$f\backslash p = \lambda x.\texttt{if}\ p\,x\ \texttt{then}\ f\ x\ \texttt{else}\ \perp$$

Then, using $\backslash$, we can define the partial function $[\![(f, b)]\!]$ as follows:

$$[\![(f, b)]\!] = f\backslash\,[\![b]\!]$$

In the context of partial functions we can express the operations described in section 1.1 succinctly using λ-abstraction.

[1] Of course, in C no information is kept at runtime about the bounds of arrays, but here we assume such information is kept, and that the bounds are checked before accessing an array.

Elementwise applied operations of functions with one argument is simply function composition. The elementwise application of g on f is :

$$\lambda x.g\,(f\,x)$$

For functions g which take more than one argument, we write

$$\lambda x.g\,(f_1\,x)\ldots(f_n\,x)$$

for the elementwise application of g on $f_1, \ldots, f_n$.

Get communication is also function composition. The *get of f from g* is

$$\lambda x.f\;(g\;x)$$

General *Send communication* has no functional meaning due to its imperative nature.

Projection can also be expressed. $\lambda x.f\;(x, k)$, where k is a constant represent the k:th row of f.

Restriction can be expressed using the $\backslash$ operation described above.

Replication is basically λ-abstractions $\lambda x.t$ over a variable x which do not occur in t. In the simplest case this is just a constant function, e.g $\lambda x.2$ is the value 2 replicated to all the possible index values of x.

Reduction can be expressed as

$$\begin{aligned}&\mathit{red}\;f\;g\;i\;n =\\ &\qquad \texttt{if}\;n = 1\;\texttt{then}\;f\;(i\;0)\;\texttt{else}\;g(\mathit{red}\;f\;g\;i\;(n-1))\;(f\;(i(n-1)))\end{aligned}$$

Here f is the partial function being reduced, g is the binary operation, i is an *enumeration function*, which maps integers to the index type of f, giving the ordering of $\mathit{dom}(f)$, and n is the number of elements used in the reduction.

We get identities for the explicit restriction operator $\backslash$, which can be used to guide the definition of operations on data fields. For elementwise applied operations we have, if g is strict in all arguments:

$$\begin{aligned}&\lambda x.g(f_1\backslash b_1\,x)\ldots(f_n\backslash b_n\,x) =\\ &\qquad (\lambda x.g\,(f_1\,x)\ldots(f_n\,x))\backslash(\lambda y.(b_1\,y)\wedge\ldots\wedge(b_n\,y))\end{aligned}$$

For `if`-expressions, we have the following identity:

$$\begin{aligned}&\lambda x.\texttt{if}\;(f_1\backslash b_1)\,x\;\texttt{then}\;(f_2\backslash b_2)\,x\;\texttt{else}\;(f_3\backslash b_3)\,x =\\ &\qquad (\lambda x.\texttt{if}\;f_1\,x\;\texttt{then}f_2\,x\;\texttt{else}\;f_3\,x)\backslash(\lambda y.b_1\,y\wedge(b_2\,y\vee b_3\,y)\end{aligned}$$

In practice we want to distinguish “out of bounds” from other errors (and especially from non-termination), so we introduce a value “$*$” which is the value which data fields assume outside of their domain. Thus we redefine $\backslash$ as

$$f\backslash p = \lambda x.\texttt{if}\ p\,x\ \texttt{then}\ f\,x\ \texttt{else}\ *$$

$*$ should behave as $\perp$. We would like $*$ to have the property that for each strict function f , $f* = *$. Things are not quite so simple for functions which take more than one argument, since we have to answer the question of what happens when we mix $*$ and $\perp$. There are several different ways of extending functions to handle $*$, but a basic requirement of an extended function is that if we replace $*$ with $\perp$, the result should be as for the original function. For a more detailed treatment of partial functions, and the proofs of identities above we refer to [15] or [23].

One possible way of using the data field model as a programming language would be to collapse the concepts "function" and "data field" and view ordinary functions as data fields with infinite bounds. The resulting language would allow one to define data fields directly by λ-abstraction, as above. This approach is taken in [13].

However, there are some problems with this approach. For one thing, a straightforward implementation would represent all functions as pairs, which would add some unnecessary overhead since ordinary functions, i.e data fields with infinite bounds, probably would be the most used. For another, a tabulated data field will be *hyperstrict*, since the argument must be fully evaluated to be used as an index in the table, while a non-tabulated data field might have quite different strictness-properties. A function f is hyperstrict if the result of applying f on a data structure which contains $\perp$ is $\perp$. Thus a function f on pairs is hyperstrict if $f(x, y) = \perp$ whenever $x = \perp$ or $y = \perp$. Ideally one would want a data field to have the same strictness properties regardless of whether it is tabulated or not. One way of solving this would be to define all data fields as hyperstrict, but if we do not distinguish data fields and functions, this would make the whole language strict.

Here we take an approach based on [25], where data fields and functions are kept separate.

The basic operations on data fields are *application*, which is analogous to function application or array indexing, *φ-abstraction* from which is is a counterpart to λ-abstraction, and *data field evaluation* which tries to evaluate a data field with finite bounds in all points in which it is defined.

To explain these operations in more detail, we need to give a more complete description of the structures of data fields and bounds. We will describe data fields and bounds abstractly as objects which have a set of primitive operations defined on them. It might be easier to think of bounds as representations of sets, rather than as representations of predicates, since the operations have more direct interpretations as operations on sets. The interpretation of a bound as a predicate can be seen as defining the membership test for the set represented by the bound. That is

$$\mathit{member}\, b\, x = [\![b]\!]\, x$$

We also need to be more explicit about types. Let the set[2] of data fields from α to β be $\mathcal{D}(\alpha, \beta)$. $\mathcal{D}(\alpha, \beta)$ is defined as the set of pairs $[\alpha \to \beta] \times \mathcal{B}(\alpha)$, where $[\alpha \to \beta]$ is the set of functions[3], from type α to type β and $\mathcal{B}(\alpha)$ is the set of bounds over the type α. Here we let $*$ represent "out of bounds", and thus we distinguish between out bounds and non-termination.

We assume the following about $\mathcal{D}(\alpha, \beta)$ and $\mathcal{B}(\alpha)$:

- As mentioned above, every bound $b \in \mathcal{B}(\alpha)$ has an interpretation as a predicate $[\![b]\!] \in [\alpha \to \mathit{bool}]$.

- Every data field $(f, b) \in \mathcal{D}(\alpha, \beta)$ has an interpretation as a hyperstrict function $[\![(f, b)]\!] = \overline{f \backslash [\![b]\!]}$, where $\overline{g}$ is the hyperstrict version of the function g. $\overline{g}$ is hyperstrict in $\perp$, but not in $*$. That is, if $x \neq \perp$ then $\overline{id}\,(x, \perp) = \perp$, but $\overline{id}\,(x, *) = (x, *)$, where id is the identity function.

- There is a *data field application* operation "!" such that $d \,!\, x = [\![d]\!]\ x$

- The bounds are partitioned into two disjoint sets $\mathcal{B}(\alpha) = \mathcal{B}_{\text{fin}}(\alpha) \cup \mathcal{B}_{\infty}(\alpha)$, where the bounds in $\mathcal{B}_{\text{fin}}(\alpha)$ are guaranteed to represent finite sets, while the bounds in $\mathcal{B}_{\infty}(\alpha)$ might represent infinite set (but need not). We call the bounds in $\mathcal{B}_{\text{fin}}(\alpha)$ *finite* bounds and the bounds in $\mathcal{B}_{\infty}(\alpha)$ *infinite* bounds.

- There is a bound $\mathit{none}_\alpha \in \mathcal{B}_{\text{fin}}(\alpha)$, which represents the empty set, and a bound $\mathit{all}_\alpha \in \mathcal{B}_{\infty}(\alpha)$, which represent the set of all elements of type α.

  That is, for all x

$$\begin{array}{rcl} [\![\mathit{none}_\alpha]\!]\, x & = & \mathit{false} \\ [\![\mathit{all}_\alpha]\!]\, x & = & \mathit{true} \end{array}$$

- For bounds $b \in \mathcal{B}_{\text{fin}}$, there are functions *size* and *enum* such that $\mathit{size}\, b$ is the size of the set represented by b, and $\mathit{enum}\, b$ is a function $\{1, \ldots, \mathit{size}\ b\} \to \alpha$ which can be used to enumerate all x such that $[\![b]\!]\ x = \mathit{true}$. Furthermore, $\mathit{size}\ \mathit{empty}_\alpha = 0$.

[2]Actually, we want the data fields to form a *domain*, which is a more specialized structure than a general set. See [52] for basic definitions and [25] for the detailed definition of data fields.

[3]More precisely, the set of continuous functions from the domain of elements of type α to the domain of elements of type β, see [52].

- There are operations $\sqcap$ and $\sqcup$ which approximate the set operations intersection and union, respectively. More precisely, we require

$$\begin{array}{rcl} [\![b_1]\!]\ x \wedge [\![b_2]\!]\ x & \Rightarrow & [\![b_1 \sqcap b_2]\!]\ x \\ [\![b_1]\!]\ x \vee [\![b_2]\!]\ x & \Rightarrow & [\![b_1 \sqcup b_2]\!]\ x \end{array}$$

- There is an operation $\times : \mathcal{B}(\alpha) \to \mathcal{B}(\beta) \to \mathcal{B}(\alpha \times \beta)$ such that

$$\begin{array}{rcl} [\![b_1 \times b_2]\!]\ (x_1, x_2) & = & ([\![b_1]\!]\ x_1) \wedge ([\![b_2]\!]\ x_2) \\ \mathit{size}\ (b_1 \times b_2) & = & \mathit{size}\, b_1 \cdot \mathit{size}\, b_2 \\ & & \text{if } b_1,\ b_2 \text{ finite} \\ \mathit{enum}\ (b_1 \times b_2)\ n & = & (\mathit{enum}\, b_1\ (n \bmod (\mathit{size}\, b_1)), \mathit{enum}\, b_2\ (n \div (\mathit{size}\, b_1))) \\ & & \text{if } b_1\ , b_2 \text{ finite.} \end{array}$$

  Here $\div$ stands for integer division.

  This can be generalized to operations $\times_k$ which create higher dimensional products.

φ-abstraction makes it possible to define data fields similarly to how functions are defined by λ-abstraction. Thus $\varphi x.d!x + d'!x$ defines a data field which is the elementwise sum of the data fields d and d'. The function part of the pair (f, b) defined by $\varphi x.t$ is basically $\lambda x.t$, and the bound b should be an approximation of $\{x \mid (\lambda x.t)\ x \neq *\}$. The bounds are propagated from the bounds of sub-expressions in a way corresponding to the identities for explicit restriction mentioned above. For example,

$$\begin{array}{l} \varphi x.(f_1, b_1)!x + (f_2, b_2)!x = \\ \quad (\lambda x.(f_1, b_1)!x + (f_2, b_2)!x, b_1 \sqcap b_2) \\ \varphi x.\texttt{if}\ (f_1, b_1)!x\ \texttt{then}\ (f_2, b_2)!x\ \texttt{else}\ (f_3, b_3)!x = \\ \quad (\lambda x.\texttt{if}\ (f_1, b_1)!x\ \texttt{then}\ (f_2, b_2)!x\ \texttt{else}\ (f_3, b_3)!x, b_1 \sqcap (b_2 \sqcup b_3)) \end{array}$$

The data field evaluation operation $\{\cdot\}$ tabulates data fields with finite bounds, and evaluates all the possibly defined elements. We let $x : xs$ denote the list which consist of x followed by the list xs, $[]$ denote the empty list, and $[x_1, x_2, \ldots, x_n]$ denote the list of the elements $x_1, \ldots x_n$. Suppose *lookup* is defined as:

$$\begin{array}{rcl} \mathit{lookup}\ x\ [] & = & * \\ \mathit{lookup}\ x\ (x', v) : xs & = & \texttt{if}\ x' = x\ \texttt{then}\ v\ \texttt{else}\ (\mathit{lookup}\ x\ xs) \end{array}$$

Then we can define $\{d\}$ for $d = (f, b)$ as:

$$\begin{array}{rcl}\{(f,b)\} & = & (\overline{\lambda x.\mathit{lookup}\ x\ (\mathit{list}\ f\ b)}, b) \\ \mathit{list}\ f\ b & = & \overline{\mathit{id}}\,[(\mathit{enum}\,b\,1, f(\mathit{enum}\,b\,1)), \ldots, ((\mathit{enum}\,b\,n), f(\mathit{enum}\,b\,n))] \\ & & \text{where } n = \mathit{size}\ b\end{array}$$

1.3 Haskell

Haskell [29] is a member of what we will loosely call the *ML family of languages*. ML stands for Meta Language, and the name comes from the fact that ML was originally designed to be used to program the Edinburgh LCF theorem prover (i.e to be used as the *Meta Language* for the prover). Some descendants of ML besides Haskell is Standard ML (SML) [26], Hope and Miranda[4] [50] (which is one of the most immediate predecessors to Haskell). These languages all incorporate features such as *pattern matching*, a *strong polymorphic type system*, *user defined data types* and *higher-order functions*. These features will be described in some detail below, along with some features of Haskell which are shared with only some or none of the other languages in the ML-family (*non-strict semantics* and *type classes* respectively). Since we are particularly concerned we indexed data structures, we also take a look at how arrays are handled in Haskell.

We will assume some acquaintance with languages such as Lisp or Scheme and describe distinguishing features of the ML-family of languages in general, and Haskell is particular. For introductory programming in Scheme, see [2]. For programming in SML (and functional programming in general) see [28]. For an introduction to Haskell, see [17]. A good source for information on functional languages in general is [1].

Pattern matching

Functions can be defined using *pattern matching*. This means the arguments of functions are matched against the patterns in the definition. As a simple example, take the function which calculates the length of a list:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Here `[]` matches the empty list, and `(x:xs)` matches any list with at least one element, binding the name `x` to the first element of the list and the name `xs` to the rest of the list.

Pattern matching can also be used in `case`-expressions. Using `case`, we can write `length` as

[4]Miranda is a trademark of Research Software Ltd.

```
length l =
  case l of
    []     -> 0
    (x:xs) -> 1 + length xs
```

Patterns can be arbitrarily nested, but there are some other restrictions on patterns. Patterns must be *linear*, i.e each variable can only occur once. Thus `(x:x:xs)` is an illegal pattern. Furthermore, patterns can only contain *constructors* and variables. A constructor is a function which creates values of some data type. This includes the list constructor `:` (pronounced "cons"), but also constants such as `1` or `'c'`, which are seen as constructors with no arguments. Constructors are defined by data type declarations, see below.

Types

Languages in the ML-family are statically typed, i.e each object in the program can be given a type a compile time. Types can be the usual integers (`Int`), real numbers (`Float`) or booleans (`Bool`), but also function types such as `Int -> Int` (functions from integers to integers), as well as more complex types such as `[Int]` (Lists of integers) or `(Int,Bool)` (pairs of integers and booleans). We also have *polymorphic* types; more about them later.

The programmer is not forced to give the types of each defined object explicitly. Instead the compiler can figure out the type of an object from the contexts the object occur in. For example, take the following function definition:

```
impl (a,b) = (not a) || b
```

Here the compiler will see that both `a` and `b` are used by functions which expects booleans as arguments (`not` and `||`), and will assign the type `(Bool,Bool) -> Bool` (i.e a function from pairs of booleans to booleans) to the function `impl`.

Polymorphic types are types which are universally quantified. To see what this means, take the `length` function again:

```
length []     = 0
length (x:xs) = 1 + (length xs)
```

The types of the elements of the list are irrelevant to `length`, it will work regardless of whether the list contains integers, real number, or even functions. `length` can be given the type `[a] -> Int`, which means that the function works for *any* type `a`, i.e the type is universally quantified over `a` (actually the type derived for `length` by the compiler will be more general than above. See section 1.3).

User defined data types

Language in the ML-family of languages provide user-defined *data types*. For example, in Haskell we can define a data type for binary trees of integers like this:

```
data Tree = Node Int Tree Tree
          | Empty
```

Here Node and Empty are the constructors of the data type, and can be used to create trees. For example, the tree which consists of one node containing the integer 1 can be created as:

```
Node 1 Empty Empty
```

Pattern matching can be used to define operations on user defined data types. For example, the size function on trees can be defined like this:

```
size Empty = 0
size (Node x l r) = 1 + (size l) + (size r)
```

User defined data types can be polymorphic. A data type for binary trees which is polymorphic in the type of object contained in the tree can be defined like this:

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
```

Higher order functions

The ML family treat functions as first class objects. This means they can be treated as any other type of object, i.e they can be passed as arguments to functions, returned from functions, stored in lists, etc. Functions can also be created anonymously, using λ-abstraction. A function which adds 1 to it argument is created by writing \x -> x + 1. Here "\" stands for λ.

Functions which take functions as parameters and/or return functions, are usually referred to as *higher order functions*. A typical example is function composition:

```
f . g =  \x -> f (g x)
```

here . takes two functions as parameters and returns their composition.

When we have higher order functions, we can choose between two different ways of writing functions with more than one argument. The following two versions of a function which adds two numbers will help to demonstrate:

```
add        :: (Int,Int) -> Int
add (x,y) = x + y

add'      :: Int -> Int -> Int
add' x y = x + y
```

`add` is the version which would be used in most programming languages. `add'` is really a higher-order function, which given an integer returns add function which adds that integer. For example, we can write `(add' 2 3)` to get `5`, or write `(add' 2)` to get a function which adds `2` to its argument.

The function `add'` is said to be on *curried*[5] form, and the function `add` is said to be on *uncurried* form. We can define functions which convert between the two forms:

```
curry       :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry         :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (x,y) = f x y
```

In the Haskell prelude and libraries, the curried form is almost always used, and the same seems to hold for most programs written in Haskell.

Non-strict languages

The probably most significant difference between Haskell and ML is that Haskell is *non-strict*, while ML is *strict*.

In a strict language, all functions are strict. In a non-strict language, functions can be non-strict. Take the following function in Haskell, for example:

```
c x = 2
```

and let `bot` be defined as:

```
bot = bot
```

In Haskell, `c bot` will have the value 2, but in a strict language such as ML, `c bot` will have the value $\perp$.

Non-strict semantics is usually implemented by *lazy evaluation*, which means arguments to functions are evaluated only when they are needed, and at most once. The opposite of lazy evaluation is *eager evaluation*, which means arguments to a function are evaluated before they are passed to the function. Eager evaluation implements strict semantics.

Non-strictness makes it possible to use conceptually infinite data structures. For example, we can define the list of all integers like this:

```
ints = 1 : (map (1+) ints)
```

Here `map` is a function which applies a function to all elements of a lists, and `(1+)` is a convenient way of writing the function which adds one to its argument.

Since the list constructor is non-strict, no part of the list will be evaluated until actually needed.

[5]After Haskell Curry.

Type classes

The kind of polymorphism described in section 1.3, where the type is parameterized over one or more type-variables is usually called *parametric polymorphism*. Another example of parametric polymorphism is templates in C++. In this kind of polymorphism, one definition works for all types.

Another kind of polymorphism is *ad-hoc polymorphism* or *overloading*. In ad-hoc polymorphism, a function may be defined differently for different types. A simple example of an overloaded function is the addition operator "+" which works both for integer and floating point operands in many languages (e.g C). Some languages (such as C++) allows the programmer to define his own overloaded functions.

One of the novel features of Haskell is the system of *type classes*, which systematizes overloading. Basically, a type is an instance a specific type class if some specific function(s) is defined on the type. As an example, take the the following function which tests if a value is in a list:

```
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

`elem` has the type `Eq a => a -> [a] -> Bool`, which means `elem` works for types which are instances of the `Eq` class, i.e types on which equality is defined.

Type classes are defined by *class declarations*, in which the functions which must be defined for types in the class are listed (these are called *class operations*). The `Eq` class can be defined like this:

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

An *instance declaration* defines the behavior of the class operations on a type. For example, here is an instance declaration which defines equality on integers, using the primitive `primEqInt` integer equality function:

```
instance Eq Int where
  (==) = primEqInt
```

The definition of `==` in the instance declaration is called a *class method*.

For polymorphic data types, we can give instance declarations which define the class operations using the class operations on the types parameterized over. Thus we get an instance declaration which requires the types parameterized over to be instances of the class. An instance declaration which defines equality for lists whose component type is an instance of the `Eq` class can be written like this:

```
instance Eq a => Eq [a] where
  [] == []         = True
  [] == _           = False
  (x:xs) == (y:ys) = x == y && xs == ys
```

Some standard classes other than `Eq` are `Ord` (which provides comparison operations), `Show` (which provides operations which convert an object to a string), and a plethora of numeric classes, of which the most basic is `Num`, which provide operations such as addition and multiplication, as well as overloaded numerical constants. Thus the type of

```
length []     = 0
length (x:xs) = 1 + (length xs)
```

will be inferred by the compiler to be `Num b => [a] -> b`

Some classes are subclasses of other classes. For example `Ord` is an subclass of `Eq`, which means that an instance of `Ord` must also be an instance of `Eq`.

This means that when writing a type signature for a function such as `ordelem` which test membership for ordered lists, which uses both `==` and `<`, we only has to specify `Ord a` and not `(Ord a, Eq a)` in the type signature:

```
ordelem :: Ord a => a -> [a] -> Bool
ordelem x (y:ys) = x == y || (x < y && ordelem x ys)
```

Since it would be tedious to define equality "by hand" for all user defined data types, Haskell offers *automatic derivation* of instances for some of the standard classes. For example, `Eq` instances can be automatically derived:

```
data Tree a = Node a (Tree a) (Tree a) deriving Eq
```

will automatically define equality on binary trees where the component type is an instance of the `Eq` class.

A class for which instances is automatically derived for all types is the `Eval` class. This class provides the operations `seq` and `strict` which control evaluation of values. `x 'seq' y` evaluates `x` and returns `y`, which `strict f x` evaluates `x` and then returns `f` applied to `x` (i.e `strict f` is a strict version of `f`).

For more information about type classes in Haskell, see [17, 29].

Haskell arrays

Haskell arrays are defined in the Haskell library report [16], and they are thus not a part of the language *per se.*

In Haskell, arrays are created from a limiting pair (a pair containing the lower and upper bound of the array) and a list of associations. Thus

```
doubles = array (1,5) [(x,2*x) | x <- [1..5]]
```

defines an array `doubles` which can be indexed with integers in $\{1, \ldots, 5\}$ to retrieve a value which is the index multiplied by two. E.g `doubles!2` evaluates to `4`. This definition makes use of *list-comprehension*, which is a notation for defining operations on lists which is inspired by the *set-comprehensions* used in mathematics. Thus `[(x,2*x) | x <- [1..5]]` is the list of all pairs `(x,2*x)` where `x` is drawn from the list `[1..5]`. List-comprehension makes it possible to define arrays succinctly.

The type of the index is not restricted to integers; any type which is an instance of the `Ix` class can be used as an array index. Thus the type of the array creation function is

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

The `Ix` class contains the operations `range`, `index` and `inRange`. `range` is used to enumerate the subscripts in the range defined by a limiting pair, `index` is used to map a limiting pair and an index to an integer, and `inRange` is used to test if an index lies in the range defined by a limiting pair.

For more information about Haskell arrays, see [17, 16]

1.4 Implementing non-strict functional languages

In this section, we first give an overview of *graph reduction*, the usual method of implementing non-strict functional languages. Then we give a description of the basic structure of a "typical" compiler for a non-strict functional language. Lastly, we try to give an idea of how type classes can be implemented.

The implementation of non-strict functional languages poses certain challenges which the implementation of conventional languages (or even strict functional languages) do not pose. This has to do with laziness (which is difficult to implement efficiently) and with features such as higher-order functions and parametric polymorphism.

A good description of the difficulties in compiling lazy functional languages can be found in [11, section 1.2].

1.4.1 Graph reduction

The execution of a program in a functional language can be defined as rewriting a given expression to a *canonical form* [52]. If we simplify slightly, canonical forms for a non-strict language are

- Constructors applied to zero or more arguments (ordinary constants such as `1`, or `'c'` are considered constructors with zero arguments)
- Expressions such as $(\lambda x_1 \ldots x_n.exp)\, e_1 \ldots e_k$, where $k < n$, i.e lambda expression which do not have all of their arguments. Since all function

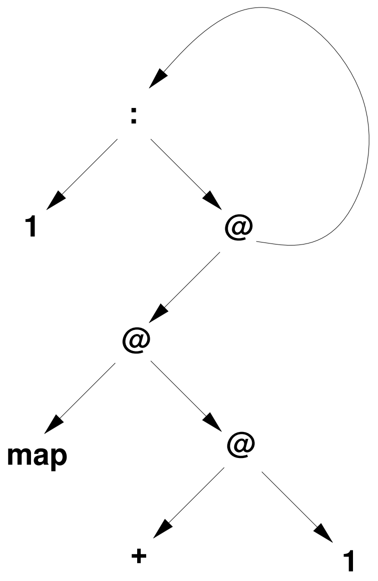

Figure 1.2: Graph-representation of the circular structure `ints`. Application nodes are marked with "@", cons nodes are marked with ":" and nodes which contain a variable of constant are marked with the variable or constant.

definitions can be written as definitions of λ-abstractions, this also covers function applications.

Expressions in these forms are said to be on *weak head normal form* or *whnf*.

How expressions should be evaluated can be defined by an operational semantics as in [52]. As an example, take the rule for applications:

$$\frac{t_1 \to \lambda x.t_1' \quad t_1'[t_2/x] \to c}{t_1\, t_2 \to c}$$

This can be read as "if t_1 evaluates to (or *reduces to*) the canonical form $\lambda x.t_1'$, and t_1' with t_2 substituted for x evaluates to the canonical form c, then t_1 applied to t_2 evaluates to c".

Implementing the above rule directly, i.e representing the program as a string and performing substitutions literally would be very expensive, both because substitution is expensive, and because it would mean some expressions would be needlessly evaluated more than once. As an example, take the expression $(\lambda x.x + x)$ (`expensive` 4711). Here we would have to evaluate (`expensive` 4711) + (`expensive` 4711), i.e `expensive` 4711 would be evaluated twice.

A better approach is to represent the initial expression as a syntax tree, and when substituting an expression for a variable, we replace the variable

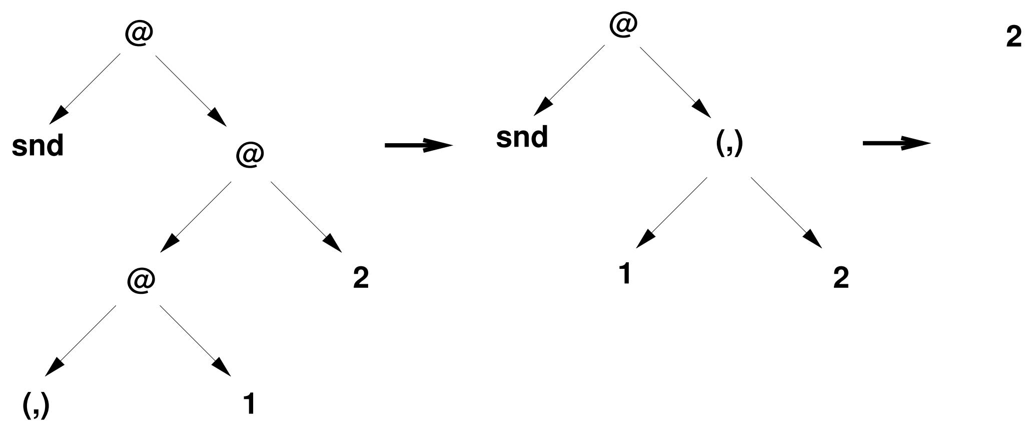

Figure 1.3: Graph-reduction of the expression `snd (1,2)`. "(,)" denotes both pair nodes and the pair constructor function.

with a pointer to the expression instead of a literal copy. This may create cyclic structures, so we get a directed graph instead of a tree. See Figure 1.2 for a cyclic graph representation of the data structure `ints`, defined as

```
ints = 1 : (map (1+) ints)
```

Since we do not want to evaluate anything more than once, we also overwrite expressions we have evaluated with their value. This is illustrated in Figure 1.3, where some steps in the reduction of the expression `snd (1,2)` is shown. This approach is called *graph reduction*, and is used (in more sophisticated forms) by all implementations of non-strict functional languages that we know of. For more information about graph reduction see [32, 19].

1.4.2 Compiler structure

The basic structure of a "typical" compiler for a functional language, one whose implementation closely follows [32] is shown in Figure 1.4. The various stages in the compilation process are described below.

Lexical and syntactical analysis

The input is first broken into a stream of tokens in the lexical analysis, which is then analyzed syntactically and turned into an abstract syntax tree (if no lexical or syntactical errors are found). A full description of lexical and syntactical analysis can be found in any textbook on compiler construction, for example in [3].

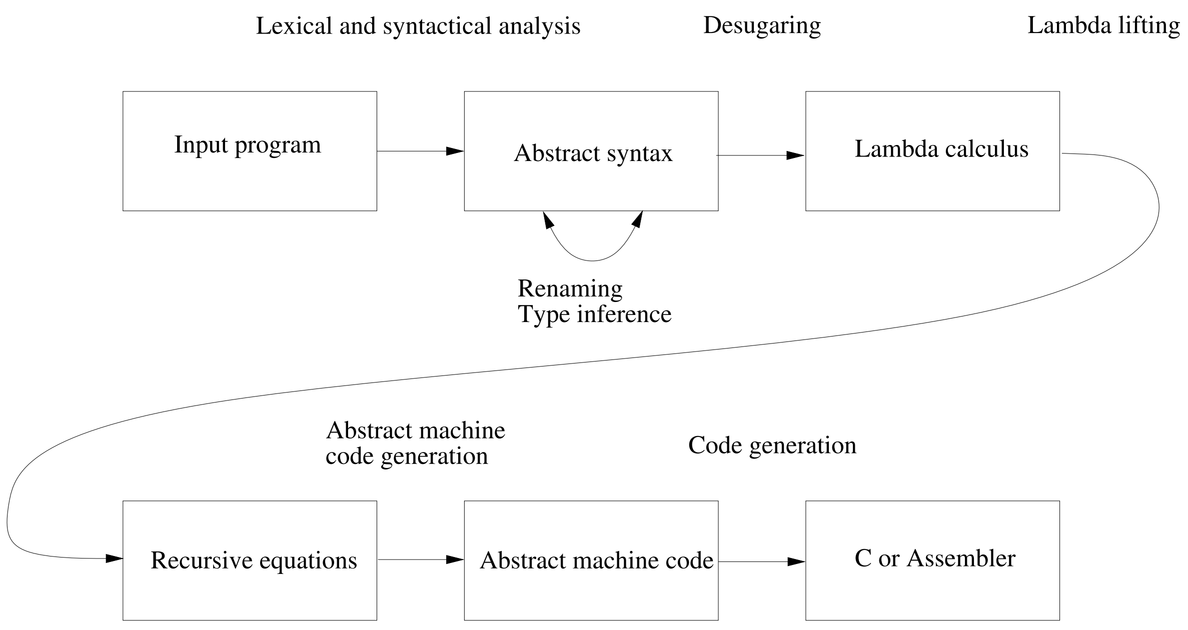

Figure 1.4: Structure of a “typical” compiler for a lazy functional language.

Renaming

In the renaming phase, each identifier is given an unique name (e.g identifiers with the same name in different scopes are given different names), and module exports and imports are resolved.

Type inference

Since the programmer does not need to specify the types of objects in a program, the compiler must be able to check that the program can be typed consistently, as well infer the actual types of objects from the contexts they appear in. This process is called *type inference* or *type checking*.

Desugaring

When type inference has finished, there are no more errors which can be detected by the compiler, so the compiler does not need to keep the entire syntax tree in order to be able to give good error messages. Since it is easier to do program analysis and transformations on a smaller language, the abstract syntax tree is translated into an equivalent abstract syntax tree for a simpler language. This is typically a very simple functional language, where all function definitions are expressed as the binding of λ-abstractions to values, and pattern-matching is expressed by “flat” `case`-expressions. Flat `case`-expressions are `case`-expressions where the scrutinized expression is a variable, and the arguments of the constructors in the alternatives are variables. An example of a "flat" `case`-expression is:

```
case v of
  C1 x y -> ...
  C2 x   -> ...
```

We usually refer to this simplified language as the compiler's *intermediate code*.

The process is called "desugaring" since it removes "syntactic sugar": syntactic constructs which are not strictly necessary, but which are included to sweeten the language.

Lambda lifting

Some abstract machines (for example the G-machine, see below) needs the program to be *lambda lifted* before it is translated into code for the abstract machine. This has to do with how functions are represented and how free variables are handled. The stack can not be used to store the free variables as described in [3], since this method assumes that a function defined in some scope will be activated only in that scope. This is true for e.g Pascal, but not for a language with higher order functions, since we can write functions such as

```
f x = \y -> (x+y)
```

in which a function which uses local variables is created and then returned. The returned function can be activated in any scope.

This problem can be solved by either storing the values of the free variables explicitly in the representation of functions, or by getting rid of the free variables. Lambda lifting is the process of removing free variables from λ-abstractions. When all free variables have been removed, the λ-abstraction is given a name and "lifted" out to the top-level. As an example take the following definition:

```
f = \x -> \y -> x+y
```

Here `x` is a free variable in the sub-expression `\y-> x+y`. `x` is removed by rewriting the definition to

```
f = \x -> (\z y -> z+y) x
```

Now `\z y -> z+y` does not contain any free variables, so the expression can be lifted out to the top level:

```
f = \x -> g x
g = \z y -> z + y
```

The output from the lambda lifting phase is a set of recursive equations, or *super combinators*. Lambda lifting is described in detail in [32].

Abstract machine code generation

The *abstract machine* makes graph-reduction concrete. The constructs of the abstract machine language have direct interpretations as operations on the graph.

The abstract machine code is generated from the super combinators, or from the intermediate code if lambda-lifting is not done.

The *G-machine* [19, 32] (G stands for *Graph*) is used in the Chalmers Haskell B Compiler (HBC), and in Nearly a Haskell Compiler (NHC). The input to the abstract code generation for the G-machine is a set of super combinators, which are translated into a sequences of G-code instructions.

The spineless tagless G-machine (STG-machine) [33], which is used in the Glasgow Haskell Compiler, is a bit different. The program is still represented as a graph, but a program for the machine is not a sequence of abstract machine instructions. Instead the abstract machine language is a simple functional programming language, the *STG-language*, in which each language construct has a direct operational meaning in terms of the STG-machine. This means the abstract machine code generation is very simple. The compiler converts the slightly more complicated *core-language* (the intermediate code of the Glasgow Haskell Compiler) to the STG-language. In addition, the STG-machine does not need the input to be lambda lifted, instead handling free variables by storing their values explicitly in the representation of functions.

Code generation

Finally, assembler or C code is generated from the abstract machine code. C, if generated, is usually used as a "portable assembler"; the generated code is very low-level.

1.4.3 Runtime environments and storage management

The *runtime environment* contains everything the generated code needs to run. This includes things such as one or more *stacks*, which are used to hold arguments of functions being evaluated, to keep track of which pieces of the graph are being reduced, and to hold pointer to code sequences where execution should continue when the expressions have been reduced. There is also the *heap* which implements the graph. Each node in the graph occupies a *cell* in the heap. The *storage manager* is used to allocate cells in the heap, and is also responsible for finding and reclaiming the *garbage* cells, cells in the heap which can no longer be reached. The part of the storage manager which finds and reclaims garbage cells is the *garbage collector*.

Two basic algorithms for garbage collection are *mark and scan* and *copying*. In mark and scan all accessible cells are first *marked* (usually by doing depth first search of the heap with the pointers on the stack as roots), then the entire heap is scanned and all unmarked cells are reclaimed. In copying, the heap is divided into two sections, *from-space* and *to-space*. All allocation of new cells is done in from-space. When from-space fills up, all accessible cells are copied into to-space, where they are placed contiguously. Then from-space and to-space are flipped. The point of placing the cells contiguously is to make all the free space contiguous. This makes allocation of new cells very cheap.

Many implementations of non-strict functional languages seem to use *generational* garbage collection [38, 40, 10]. This is a hybrid method, which builds on the observation that most objects have a very short life time (i.e, the time between the allocation of the cell for the object and the point in time when all references to the object is gone). This is taken advantage of by dividing the heap into two or more *generations*, which each contain objects with a particular range of ages. The basic idea is, since most objects die young, we should collect younger generations more often than older generations, and move objects which survive one or more collections to an older generation. Usually an older generation is collected only if all younger generations have run out of memory. Different algorithms can be used to collect different generations.

1.4.4 Implementing type classes

The standard way to implement type classes is to transform the program with overloading into a equivalent program without overloading. The principles of the translation are as follows:

Each instance declaration generates a *dictionary* declaration. A dictionary is a tuple which contains the class methods. As an example, take the following class declaration of a simplified version of `Ord`:

```
class  Eq a => Ord a where
    (<), (>=)       :: a -> a -> Bool
```

The instance definition

```
instance Ord Int where
  (<)  = primLtInt
  (>=) = primGeqInt
```

will generate the dictionary definition

```
dictOrdInt = (dictEqInt, primLtInt, primGeqInt)
```

The class declaration for `Ord` generates definitions of the class operations as *selector functions* on dictionaries:

```
(<) (dictEq, lt, geq)  = lt
(>=) (dictEq, lt, geq) = geq
```

A function which retrieves the dictionary for `Eq` from a dictionary for `Ord` will also be generated:

```
getEqFromOrd (dictEq, lt, geq) = dictEq
```

Overloaded functions are given extra arguments, as a simple example take the `max` function:

```
max :: Ord a -> a -> a -> a
max x y = if x > y then x else y
```

`max` will be translated into:

```
max ordDict x y = if (>) ordDict x y then x else y
```

Things get more complicated when we have instances for parameterized data types such as lists, but the basic idea is as described above. For a more complete and more formal treatment of the transformation, see [14]. For implementation techniques, see [6] and [31].

Chapter 2

Data field Haskell

This sections gives an informal description of our Haskell-dialect. A more formal description can be found in appendix A.

We extend Haskell with data fields, i.e objects which represent generalized indexed data structures, and which we should be able to think of as partial functions. In comparison with Haskell-arrays, this makes it possible to define indexed data structures without explicitly giving the bounds. This is achieved by using φ-abstraction, the data field analogue to λ-abstraction. As an example, the data field definitions corresponding to the following array definitions in Haskell

```
a = array (1,10) [(i,i) | i <- [1..10]]
b = array (1,10) [(i,i*i) | i <- [1..10]]
sumab = array (1,10) [(i, a!i + b!i) | i <- [1..10]]
```

can be written as

```
a = datafield (\i -> i) (1 <:> 10)
b = datafield (\i -> i*i) (1 <:> 10)
sumab = forall i -> a!i + b!i
```

We write φ as `forall` in Haskell, due to the limited character set.

Data fields can also have more general shape than Haskell arrays. For example, we allow "sparse" data fields, i.e data fields which are defined on general finite sets.

To be able to use the data field extensions, the `Datafield` module must be imported.

2.1 Data fields

Data fields are represented by the abstract data type `Datafield a b`, where `a` is the type of the index, and `b` is the type of the indexed elements. Data fields can be constructed implicitly by using `forall`- or `for`-expression (more about them below), or explicitly by using the `datafield` function. The `datafield` function takes a function and a bound as parameters, and returns a data field. Elements of data fields are accessed by using the `!` operator. The following examples shows how `datafield` and `!` are used:

```
d = datafield (\x -> x) (1 <:> 10)
```

defines a data field `d` defined in the interval $[1, 10]$, for which `d!x` will return `x` for `x` in $[1, 10]$, and $*$ (the value which represent "out of bounds") for other `x`. `1 <:> 10` is an example of a *dense* bound. They, and other kinds of bounds, are described below. The bounds of a data field can be retrieved by the `bounds` function.

The index type of data fields is not restricted to integers. Any type which is an instance of the type classes `Ix` and `Pord` can be used as an index type. The `Ix` class is used to map continuous subranges of values in a type onto integers [16, Section 5]. The `Pord` class is used mostly to provide a pointwise partial ordering `lt` of tuples (as opposed to the lexicographical total order `<=` provided by the `Ord` class). The `Pord` class also provides the `glb` (greatest lower bound) and `lub` (least upper bound) operations as defined by the partial order. These can be thought of (in the case of tuples) as a pointwise `min` and `max` operations, i.e `glb (1,3,4) (2,1,2)` evaluates to `(1,1,2)`. These operations are needed since we for instance have to be able to calculate the intersection of two dense bounds; this amounts to calculating the `lub` of the lower bounds and the `glb` of the upper bounds.

Of course it is possible to define `Pord` instances for user defined data types and use them to index data fields.

Thus the types for `datafield` is

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

the type for `bounds` is

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

and the type for `!` is

```
(!) :: (Pord a,Ix a,Eval a) => Datafield a b -> a -> b
```

The `Eval a` context in the type of `!` is needed for technical reasons. Data field application should be hyperstrict (i.e the index should be evaluated to its innermost constructor), and the implementation of `!` achieves this by using `hseq`, which is an operation in our extended `Eval`-class. See section A.4.

Data field evaluators

Data fields with finite bounds can be tabulated by a functions which perform different degrees of evaluation of the elements.

- `tab` creates a tabulated version of a data field with finite bounds. The elements of the data field are evaluated on demand. `tab` has the type:

  ```
  tab :: (Pord a, Ix a, Eval a) =>
         Datafield a b -> Datafield a b
  ```

- `strictTab` differs from `tab` in that the elements in the tabulated data field will be evaluated to whnf (i.e to the outermost constructor). `strictTab` has the type:

  ```
  strictTab :: (Pord a, Ix a, Eval a, Eval b) =>
               Datafield a b -> Datafield a b
  ```

- `hstrictTab` works as `tab` and `strictTab` except that the elements in tabulated data field will be totally evaluated (i.e to the innermost constructor) `hstrictTab` has the type:

  ```
  hstrictTab :: (Pord a, Ix a, Eval a, Eval b) =>
                Datafield a b -> Datafield a b
  ```

Neither `strictTab` nor `hstrictTab` are $*$-strict, i.e if any element evaluates to $*$, the result of the evaluation will not be $*$, instead the $*$ will be stored in the table.

Reduction

`foldlDf` applied to a function `f`, a starting value `z` and a data field `df` with finite bounds, reduces the data field from left to right (according to the enumeration of the bounds, see below) using `f`. The type is:

```
foldlDf :: (Pord a, Ix a, Eval a) =>
           (b -> c -> b) -> b -> (Datafield a c) -> b
```

Indices `i` where `f r' (df!i)` returns $*$ are skipped in the reduction (`r'` represents the result of reducing to the index preceding `i`). To make sure all indices where `df!i` $= *$ are skipped, one can write `foldlDf (\z -> strict (f z)) z df` There are several other folds, such as `foldrDf` which reduces from right to left. For details, see Appendix A.

Restriction

The restriction operator `<\>` can be used to further restrict the domain of a data field. For example,

```
f = (datafield  (\x -> x) (1<:>10)) <\> (0<:>5)
```

creates a data field `f` which is defined in the interval $[1, 5]$. The type of `<\>` is

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

2.1.1 Printing data fields

Data field are instances of the class `Show` given that the index type and the element type are instances of the class show. This means they can be converted to strings as usual, using the `show` function (or printed directly, using `print`. Elements which are $*$ are shown as `<OUB>` as default, but this can be defined for each user defined type by defining the $showsOutoufbounds$ method in the instance definition for `Show`.

2.2 Bounds

In section 1.2, we described bounds abstractly as a set of objects which has certain operations defined on them. In this section, we will be more concrete, and define bounds which are either the usual array-type bounds, sparse bounds (i.e general finite sets), bounds which are general predicates, and products of these bounds. Our approach is based on [25].

Bounds are represented by the abstract data type `Bounds a`, where `a` is the type of the members of the set (which is the type of the index of the data field), and the operation `inBounds` with type `(Pord a,Ix a) =>a -> Bounds a -> Bool`, is used to test whether a value is an element of a bound.

2.2.1 Dense bounds

Dense bounds are isomorphic to contiguous sets of integers. These are the usual "array-bounds" found in most programming languages. Dense bounds are created with the `<:>` operator:

```
(<:>) :: (Ix a, Pord a) => a -> a -> Bounds a
```

For example `(1,1) <:> (10,20)` will create a bound which contains all elements in the the rectangle with the lower left corner $(1, 1)$ and upper right corner $(10, 20)$.

2.2.2 Sparse bounds

Sparse bounds represent general finite sets. Sparse bounds are created with `sparse` which takes a list and returns a sparse bound containing the elements of the list:

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

For example `sparse [1,17,42,4711]` will create a sparse bound which contains the elements 1, 17, 42, and 4711.

2.2.3 Predicate bounds

A predicate bound represents a set which contains the values for which a given predicate is true. While dense and sparse bounds represent finite sets, a predicate bound can represent a infinite set. This means that there are fewer operations defined on them, and thus predicate bounds are mostly useful for use with the restriction operator `<\>` and sparse or dense bounds. Predicate bounds are created with `pred`:

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

An example:

```
(datafield (\x -> x+1) (0<:>1000)) <\> (predicate even)
```

creates a data field which is defined only on the even integers in the interval $[0, 1000]$.

2.2.4 The bounds `universe` and `empty`

The special bound `universe` represents all elements in the index type, and `empty`[1] represents the empty set. That is ``x `inBounds` universe`` returns `True` for all `x`, and ``x `inBounds` empty`` returns `False` for all `x`. The bounds `universe` and `empty` have the types

```
universe :: (Pord a, Ix a) => Bounds a
empty    :: (Pord a, Ix a) => Bounds a
```

2.2.5 Product bounds

Product bounds represent Cartesian products of bounds. Product bounds are created with the `<*>` operator:

```
(<*>) :: (Pord a, Ix a, Pord b, Ix b) =>
         Bounds a -> Bounds b -> Bounds (a,b)
```

[1]These are the bounds *all* and *none* mentioned in section 1.2, but a function with the name `all` is already defined in Haskell. `universe` and `empty` were the best names we could think of which caused no conflicts with existing functions.

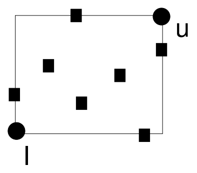

Figure 2.1: Greatest lower and least upper bounds for a two dimensional sparse bound.

The Cartesian product `bx<*>by` of two bounds `bx` and `by` contains all elements (x, y) such that x is an element of `bx` and y is an element of `by`.

Product bounds make it possible to form bounds which are combinations of sparse and dense bounds, as well as bounds which are combinations of finite and infinite bounds. For example, `(sparse [5,7,11,13])<*>(1<:>10)` will represent the set of all pairs (x, y), where x is taken from the set $\{5, 7, 11, 14\}$, and y is taken from the set $[1, 10]$. As an example of a product of a finite and and infinite bound, take `universe<*>(1<:>10)`. This will represent the set of pairs (x, y) where x can have any value, and y is taken from $[1, 10]$.

2.2.6 Some operations on bounds

In addition to the functions `inBounds`, `<:>`, `sparse`, `pred`, `universe`, `empty`, `<*>` and `<\>`, which are described above, some of the other operations on bounds are:

- `finite`, which returns true for finite bounds. Finite bounds are dense bounds, sparse bounds, `empty`, and (inductively) products where the components are finite bounds. The type of `finite` is

  ```
  finite :: (Pord a, Ix a) => Bounds a -> Bool
  ```

- `enumerate`, which returns a sorted list containing the elements of a finite bound. An example is `enumerate (1<:>5)` which returns `[1,2,3,4,5]`. The type of `enumerate` is

  ```
  enumerate :: (Pord a, Ix a) => Bounds a -> [a]
  ```

- `size`, which returns the number of elements in a finite bound. The type of `size` is

  ```
  size :: (Pord a, Ix a) => Bounds a -> Int
  ```

- `lowerBound`, which returns the greatest lower bound of a finite bound as defined by the `glb` method in the `Pord` class. For a dense bound constructed as `l<:>u`, `lowerBound` always returns `l`. For a sparse bound, `lowerBound` is the lower left corner of the smallest (hyper) rectangle which contains the bound. See Figure 2.1. The type of `lowerBound` is

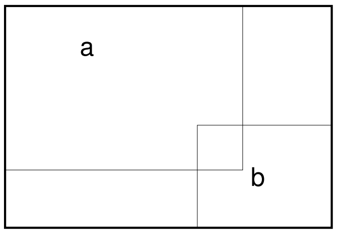

Figure 2.2: Approximation of the union of the bounds `a` and `b`.

.

```
lowerBound :: (Pord a, Ix a) => Bounds a -> a
```

- `upperBound`, which returns the least upper bound of a finite bound as defined by the `lub` method in the `Pord` class. For a dense bound constructed as `l<:>u`, `upperBound` always returns `u`. For a sparse bound, `upperBound` is the upper right corner of the smallest (hyper) rectangle which contains the bound. See Figure 2.1. The type of `upperBound` is

  ```
  upperBound :: (Pord a, Ix a) => Bounds a -> a
  ```

Applying one of the functions which are only defined on finite bounds (`enumerate`, `size`, `lowerBound`, and `upperBound`) to an infinite bound result in an error (i.e $\perp$).

For a full description of all operations on bound, see appendix A.

2.3 Syntactic constructs

Data fields can be defined by `forall`- and `for`-abstraction. `forall`-abstraction is ASCII syntax for the φ-abstractions of [25], mentioned in section 1.2, while `for`-abstractions is a syntax for defining data fields by cases on the index.

2.3.1 `forall`-abstraction

The bounds of a data field defined by a `forall`-abstraction is automatically inferred from the form of the expression and the bounds of data fields which the expression depend on. The rules for the inference of bounds are guided by the view of data fields as partial functions. The basic intuition is that if the data field `d` depends on the data fields `a` and `b`, then `d` should be defined for values where both `a` and `b` are defined. A simple example is

```
d = forall x -> a!x + b!x
```

The bound of `d` will be the intersection of the bounds of `a` and `b`. So, if `a` has the bound `1<:>10` and `b` has the bound `5<:>12`, then `d` will have the bound `5<:>10`. If `a` and `b` has the sparse bounds `sparse [2,3,5,7,11]` and `sparse [1,2,4,7,10,11]`, then `d` will have the bound `sparse [2,7,11]`. If `d` depends on one of the data fields `a` and `b`, then `d` should be defined wherever one of the data fields `a` or `b` is defined. As an example, take

```
d = forall x -> if x > 5 then a!x else b!x
```

the bounds of `d` should be the union of the bounds of `a` and `b`. With the same bounds for `a` and `b` as above, we get the bounds `1<:>12` for `d`.

However, the union of two dense bounds need not be a dense bound. This means we either have to convert the dense bound into a sparse bound, or approximate with the smallest dense bound containing the union, as shown in Figure 2.2. Since we would like “denseness” to be preserved, the solution we choose is to approximate.

`forall`-abstractions can be used to express projections and other similar operations. Consider

```
a = datafield (\(x,y) -> x+y) ((1,1)<:>(10,10))
c = forall x -> a!(1,x)
d = forall x -> a!(x,x)
```

The data field `c` will consist of the first row of `a`, and will have the bounds `1<:>10`. The data field `d` will have the same bounds, but will consist of the diagonal of `a`, i.e `d!x` will have the the value `x+x` for `x` in `bounds a`.

We can also use `forall`-abstractions to *translate* data fields. Thus we can write

```
d = forall (x,y) -> a!(x+1,y+1)
```

with `a` as above, this will create a data field `d` with the bounds `(0,0)<:>(9,9)`, and `d!(x,y)` will be `(x+1)+(y+1)`.

For a full description of how the bounds are inferred for a `forall`-abstraction, see Figure A.4 in appendix A.

2.3.2 `for`-abstraction

`for`-abstractions gives the programmer more explicit control of the bounds than `forall`-expressions. They make it possible to define data fields by cases, i.e data fields can be defined by different expressions for indices which lie in different bounds. For example, to define a dense data field `d` with the bounds `1<:>10` where `d!x` is `1` for $1 \leq x \leq 5$ and `2` for $6 \leq x \leq 10$, we write

```
d = for x in 1<:>5  -> 1
             6<:>10 -> 2
```

This is somewhat similar to the `for`-expressions in Sisal [12], but the most immediate inspiration was the `case` construct for data fields in [25].

2.4 Examples

```
-- Create a datafield containing the first 10000 fibonacci
-- numbers

fib :: Datafield Int Int
fib  = for x in 1<:>2     -> 1
                3<:>10000 -> (fib ! (x-2)) + (fib ! (x-1))

-- Create a tabulated version of the datafield above, where
-- the elements are computed on demand

fib' :: Datafield Int Int
fib' = tab fib

-- create a totally evaluated datafield with the first 1000
-- fibonacci numbers:

fib'' :: Datafield Int Int
fib'' = strictTab (fib <\> (1<:>1000))
```

2.4.1 Matrix multiplication

For a longer example, we look at matrix multiplikation. Let A be an $n \times m$ matrix, B be an $m \times l$ matrix, and $C = AB$ be their product. Then

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

This can be expressed rather elegantly as a function on data fields:

```
matrixmult a b =
  forall (i,j) -> foldr1Df (+) (forall k -> a!(i,k) * b!(k,j))
```

Strassen's algorithm

The above algorithm for multiplying matrices has complexity $O(n^3)$. An algorithm with lower complexity is Strassen's algorithm for matrix multiplication. The algorithm works when both the matrices to be multiplied are $n \times n$ matrices. The idea is to split the matrices into blocks and then apply the algorithm recursively, in a clever way. We split the matrices into four similar blocks (we assume $n = 2^k$):

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$. If we perform these computations directly, the algorithm will use 8 matrix multiplies and 4 matrix adds, which will give us 8 recursive calls, which will give us no improvement in complexity. However, it is possible to reformulate the computations such that only 7 multiplies are performed. If we let

$$\begin{aligned}
m_1 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
m_2 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
m_3 &= (A_{11} - A_{21})(B_{11} + B_{12}) \\
m_4 &= (A_{11} + A_{12})B_{22} \\
m_5 &= A_{11}(B_{12} - B_{22}) \\
m_6 &= A_{22}(B_{21} - B_{11}) \\
m_7 &= (A_{21} + A_{22})B_{11}
\end{aligned}$$

then it is easy but tedious to verify that

$$\begin{aligned}
C_{11} &= m_1 + m_2 - m_4 - m_6 \\
C_{12} &= m_4 + m_5 \\
C_{21} &= m_6 + m_7 \\
C_{22} &= m_2 - m_3 + m_5 - m_7
\end{aligned}$$

Thus we can multiply the matrices using 7 recursive calls and 18 matrix adds.

To express Strassen's algorithm as a function on data fields we begin by defining some convenient functions: Since Strassen's algorithm operates on square matrices with upper left corner $(1, 1)$, we need a function `align`, which translates a matrix such that the upper left corner is $(1, 1)$.

```
align d = forall (x,y) -> d!(x+lx-1,y+ly-1)
  where (lx,ly) = lowerBound (bounds d)
```

We also need functions `c11`, `c12`, `c21` and `c22` which selects the four different sub-matrices (and aligns them):

```
c11 d = align (d <\> (lx,ly) <:> (ux‘div‘2,uy‘div‘2))
  where (lx,ly) = lowerBound (bounds d)
        (ux,uy) = upperBound (bounds d)

c12 d = align (d <\> (lx,uy‘div‘2 + 1) <:> (ux‘div‘2,uy))
  where (lx,ly) = lowerBound (bounds d)
        (ux,uy) = upperBound (bounds d)
```

```
c21 d = align (d <\> (ux‘div‘2+1,ly) <:> (ux,uy‘div‘2))
  where (lx,ly) = lowerBound (bounds d)
        (ux,uy) = upperBound (bounds d)

c22 d = align (d <\> (ux‘div‘2+1,uy‘div‘2+1) <:> (ux,uy))
  where (lx,ly) = lowerBound (bounds d)
        (ux,uy) = upperBound (bounds d)
```

We also need to perform the inverse operation, i.e put together four submatrices into a larger matrix. This can be expressed by the functions `concatv` and `concath`, which concatenates two matrices by putting them beside each other vertically and horizontally, respectively:

```
concatv d1 d2 = for x in bounds d1  -> d1!x
                         bounds d2' -> d2'!x
  where (ux,uy) = upperBound (bounds d1)
        d2'     = forall (x,y) -> d2!(x-ux,y)

concath d1 d2 = for x in bounds d1 -> d1!x
                         bounds d2' -> d2'!x
  where (ux,uy) = upperBound (bounds d1)
        d2'     = forall (x,y) -> d2!(x,y-uy)
```

Now we can express Strassen’s algorithm as

```
strassen a b
  | size (bounds a) == 1 = forall x -> (a!x)*(b!x)
  | otherwise            = concath (concatv d11 d21)
                                   (concatv d12 d22)
  where m1 = strassen (forall x -> (c12 a)!x - (c22 a)!x)
                      (forall x -> (c21 b)!x + (c22 b)!x)
        m2 = strassen (forall x -> (c11 a)!x + (c22 a)!x)
                      (forall x -> (c11 b)!x + (c22 b)!x)
        m3 = strassen (forall x -> (c11 a)!x - (c21 a)!x)
                      (forall x -> (c11 b)!x + (c12 b)!x)
        m4 = strassen (forall x -> (c11 a)!x + (c12 a)!x)
                      (c22 b)
        m5 = strassen (c11 a)
                      (forall x -> (c12 b)!x - (c22 b)!x)
        m6 = strassen (c22 a)
                      (forall x -> (c21 b)!x - (c11 b)!x)
        m7 = strassen (forall x -> (c21 a)!x + (c22 a)!x)
                      (c11 b)

        d11 = forall x -> (m1!x) + (m2!x) - (m4!x) + (m6!x)
```

```
d12 = forall x -> (m4!x) + (m5!x)
d21 = forall x -> (m6!x) + (m7!x)
d22 = forall x -> (m2!x) - (m3!x) + (m5!x) - (m7!x)
```

Chapter 3

Haskell implementations

Since implementing Haskell is a large undertaking, our implementation of a Haskell-dialect had to build on an existing implementation of Haskell. In this section, we first give a brief overview of the different Haskell implementations. We then give a more detailed evaluation of the two compilers (GHC and NHC), which we thought were the most promising alternatives.

3.1 Overview of Haskell implementations

The available Haskell implementations were Hugs, GHC, NHC and HBC. Here we give an overview of their features.

3.1.1 Hugs

Hugs (The Haskell Users Gofer[1] System) [20] is a Haskell-interpreter, designed to run on small systems and to be easily portable. It is written in C, and is very fast in compiling, type checking and reading files. It executes Haskell-programs slowly, however (being an interpreter). It is very useful for incremental development and fast prototyping.

3.1.2 GHC

The Glasgow Haskell compiler (GHC) is a compiler for full Haskell 1.4. GHC is developed at the University of Glasgow. It is written in Haskell, and runs on many architectures. It generates the fastest code of all the implementations, and it is designed to be used as a research platform.

GHC is written in Haskell, except for the parser and lexer which are written in bison and flex. GHC is the largest and most advanced of the compilers. Many extensions to Haskell are implemented in GHC, and GHC

[1]Gofer (GOod For Equational Reasoning) is an interpreter for a Haskell-like language. Hugs is based on the Gofer interpreter.

has advanced facilities for optimization. Some extensions implemented in GHC are extensions for writing parallel [49] and concurrent [35] programs, for mutable arrays and for calling C [37]. The philosophy of GHC is to express as much as possibly of the compilation process (including optimizations) as correctness preserving program transformations [34, 41].

GHC generates C-code or assembler.

3.1.3 NHC

NHC (Nearly a Haskell Compiler) [38] is a compiler for almost all of Haskell 1.3 [30]. The original version of NHC was developed by Nicklas Röjemo when at Chalmers, but updated versions have been released by the functional programming group at the University of York, who have added facilities for heap compression and binary I/O [51], facilities for tracing and debugging Haskell programs [42], and an interface for calling C [37].

NHC is a relatively small compiler, and is written entirely in Haskell.

NHC was designed to be memory efficient. More exactly, NHC was designed to minimize the memory usage when NHC compiled by NHC compiles itself. This was achieved by both writing the compiler in a memory efficient way, and by making the compiler generate memory efficient code. To facilitate the development of memory efficient Haskell programs, NHC has extensive facilities for profiling memory usage [38, 39].

NHC does not generate an assembler sequence for each abstract-machine instruction. Instead, NHC generates *byte code*, and links with a *byte code interpreter*. This is detrimental to speed, but makes it possible to create a small binary, which decreases the memory-usage.

3.1.4 HBC

The Haskell B Compiler (HBC) [5] is a combined lazy ML (LML) / Haskell compiler which was originally a LML compiler [4]. It is written in LML. The design goal for HBC is to generate as fast code as possible.

HBC generates assembler.

3.2 Comparison of GHC and NHC

The two compilers which we considered as platforms for implementing our dialect were GHC and NHC. Hugs is an interpreter, and we wanted to use a compiler. In addition, Hugs is written in C, which of course is a minus. We did not consider HBC seriously as a platform for implementing our extensions, since HBC basically has the worst of both worlds in relation to NHC and GHC. It is large and undocumented, while GHC is documented but large, and NHC is undocumented but small. Yet another point against HBC are the sometimes atrociously bad error messages. HBC performs type checking *after* the input has been desugared, which means the the error messages on type errors shows the intermediate code, and not the original program. This can lead to extremely confusing error messages.

We evaluated GHC and NHC according to the following criteria:

- *Documentation and extensibility.* Most importantly, the compiler had to be to be easily extended enough so that our extensions could be implemented with a reasonable effort.

  We thus looked at how well the compiler source was documented, how hard it was to get an overview of the compiler, and how hard the code was to modify.

  To make this more concrete, we decided to experiment by adding a dummy primitive operation to the compilers, in addition to reading the source code and comments.

  We also looked especially at the parts of the compiler we originally deemed most likely to have to be modified: the parser and the type checker. We also thought the easiest way to implement the data fields extensions would be as a program transformation of the compiler's intermediate code, and thus we looked especially at this format.

- *Compiler quality.* Since we would like our compiler to generate good code, to be fast in compiling, and to give good error messages, we also looked at those aspects of the compilers.

- *Portability to parallel architectures.* Parallel extensions implemented by the compiler were a definite plus, since a long term goal is to have a parallel implementation, and it would probably be much easier to adapt parallel extensions already in place than to implement them from scratch.

3.2.1 Documentation and extensibility

The source code of GHC is well-documented, and technical information about GHC is readily available [41, 33, 36]. The comments are relevant, but in spite of this we found it hard to get an overview of the compiler. We found the sheer size of GHC intimidating, and we thus considered the probability of encountering unforeseen problems high. The following quote from [36] gives some justification to our fears:

> We consistently underestimated how long it would take us to do the job, largely because of the scale of the compiler. Apparently-small changes would cause a chain of effects all of which would have to be chased through before the compiler could be rebuilt.

The many extensions implemented in GHC might also be troublesome to get to work with our own extensions.

The source code of NHC is barely documented at all. However, NHC basically follows [32], and is thus fairly standard in its structure, and some parts of NHC where the compiler differs from [32] is at least briefly described in [38] (an example is the implementation of the G-machine whose implementation in NHC differs somewhat from [32]). NHC is also a rather small compiler, which makes it easier to get a more complete overview of the compiler.

Since the design goal of NHC was to minimize memory usage we considered it a risk that some parts of the compiler would be written in a non-obvious way to minimize memory usage. After examining the code we indeed found some parts which were difficult to understand, for example the part of the type-checker responsible for resolving overloaded functions.

We found adding a dummy primitive to NHC and GHC about equally difficult (we had a little more problems with finding all places where the code had to be changed in GHC, but not much). From this we conclude that our chosen modification probably was a bit too trivial to get a good impression of how the compilers are to work with.

The parser and lexer of GHC are written using flex and bison, which interface with the main Haskell program by producing output with a simple syntax which is then parsed by a simple recursive descent parser written in Haskell [36]. We have not actually modified the grammar in GHC, so we do not know how hard it really is to add things to the interface. But it would seem to be a bit more complex than if the entire parser was written in Haskell.

NHC has a lexer and parser written entirely in Haskell. The parser is written using *parser-combinators* [18, 38], which are higher-order functions which build complex parsers by combining simpler ones. Since the parser is written entirely in Haskell, it is probably more easy to make simple modifications to than the GHC-parser. However, with a parser written using bison, we get warnings if our new rules cause conflicts, so for complicated extensions of Haskell syntax it might be easier to use GHC.

The intermediate code of GHC, the *core-language*, is explicitly typed. The intermediate code of NHC is not typed. There are some other differences between the intermediate code of NHC and GHC, for example NHC uses the constructs `fail` and $[\!]$ (described in section 4.4.1) to efficiently express pattern-matching, while GHC avoids using them by using a program transformation which transforms less efficient formulations of pattern matching into more efficient ones [34].

The explicit types of GHC would make it easier to check the correctness of our transformation, since GHC has a *core-lint* which checks the type-consistency of the core language. It also has an well-defined operational semantics, which is nice.

```
module Typeerr where

f z (x,y) = (x,z)

g = snd (f (1,2))
```

Figure 3.1: `Typeerr.hs`

3.2.2 Compiler quality

We have not spent too much effort on measuring the performance of the compilers, but here are some impressions:

Generated code

GHC generates much faster code than NHC. Not only does GHC have the ability to do lots of optimizations, while NHC has none, NHC uses interpreted byte code instead of machine code, which according to [38] costs about a factor four in speed. Since NHC is designed for memory-efficiency, the memory usage of code generated by NHC is probably better than for code generated by GHC. We have not done any actual measurements of either time or memory usage, however.

Compiler speed

NHC compiled with HBC is faster than GHC. NHC is about twice as fast as GHC when compiling a small program (about 250 lines). This is when using GHC with optimizations turned off. If optimizations are turned on, NHC is more than three times as fast as GHC. We have not measured NHC compiled by NHC, but it would probably be even slower than GHC with optimizations turned on.

Error messages

Both GHC and NHC have better error messages than HBC. Especially the error messages on type errors are much better, since both GHC and NHC performs type checking on the original syntax tree, and not on some intermediate code. In fact, the must frustrating errors when programming in Haskell are usually the type errors. Syntax errors are usually much easier to spot, and thus require less help from the compiler. So it is important that a Haskell compiler produce good error messages on type errors.

The error messages from trying to compile the program in Figure 3.1 with GHC and NHC respectively are shown in Figure 3.2 and Figure 3.3. From the examples, it may seem that the error messages from GHC are more

```
Typeerr.hs:5: Couldn't match the type '(,)' against '->'
    Expected: '(taU9, taU8)'
    Inferred: '(taUc, taUd) -> (taUc, taUf)'
    In the first argument of 'snd', namely '(f (1, (2)))'
    In a pattern binding: 'g = snd (f (1, (2)))'

Compilation had errors
```

Figure 3.2: Error message from GHC when compiling `Typerr.hs`.

```
====================================
        Error after type deriving/checking:
Type error type clash between Prelude.2 and Prelude.->
when trying to apply function at 5:5 to its 1 argument at 5:10.
```

Figure 3.3: Error message from NHC when compiling `Typerr.hs`.

useful. In practice, however, we have found that the exact positions (both column and row) given by NHC more than compensate for the brevity of the error messages.

3.2.3 Portability to parallel architectures

GHC can run parallel programs on parallel architectures which support PVM (Parallel Virtual Machine). The parallel extensions are just two new primitives `par` and `seq` [43]. `x 'par' y` *sparks* `x` and returns `y`. Sparked expression are put on queue to be evaluated in parallel. `x 'seq' y` evaluates `x` and returns `y`. `seq` is used to force sequential evaluation.

It would probably be quite easy to implement simple parallel evaluation of data fields in GHC (given that the basic data field constructs have been implemented, of course). To implement good support for parallel evaluation of data fields would probably be more work, since one probably would like to take advantage of the more structured parallelism of data field evaluation.

NHC has no parallel extensions, and it would probably be a lot of work to add parallelism to NHC. It would probably be best to give NHC an entire new back-end, since the point of parallelism in the first place is speed, and it is a bit sub-optimal to have a highly parallelized byte code interpreter.

3.2.4 Conclusions

On a point-by-point basis, GHC might seem to be the choice. But the most important consideration was how likely it seemed to be that the project would be finished in reasonable time. We made the judgment that the sheer size of GHC would increased the likelihood of encountering problems, and that NHC thus would be easier to get an overview of, despite the lack of documentation. Another, more trivial point is that a smaller compiler takes less time to compile. Thus, we figured development speed would be higher if we used NHC.

A point against NHC would be that NHC only implements Haskell 1.3, while GHC implements Haskell 1.4. The differences between Haskell 1.3 and Haskell are minor, however.

In the end we choose to go with NHC[2].

[2]However, if GHC version 4.00 had been available when we made the decision, we might have chosen differently. In GHC 4.00, the intermediate code, and thus much of the compiler, has been simplified, and exceptions (which we added when implementing our extensions) are already available [44, section 1.4].

Chapter 4

Implementation

In this section, we describe how the extensions described in 2 and Appendix A were implemented.

The implementation consists of

- Modifications to the front-end which enable the compiler to parse and type-check `forall` and `for`-abstractions.
- Automatic derivation of instances for the new type class `Pord`, and automatic derivation of instances for the modified `Eval` class (see Appendix A).
- A program transformation which transforms intermediate code with `forall`- and `for`-abstractions into intermediate code without `forall` and `for`-abstractions.
- The abstract data types for `Datafield` and `Bounds` implemented in Haskell.
- Simple exception handling (used to implement $*$), implemented mostly by modifications to the back-end.

4.1 Parsing and lexing

Modifying the parser and lexer to parse the data field syntax as described in Appendix A was mostly straightforward. However, the syntax for `for`-abstractions caused some trouble with the layout rules. The layout rules make it possible to group declarations by using indentation, and avoid using explicit semi-colons, curly-brackets, etc. Layout rules are fully described in [29, section 1.5].

The problem which appeared was that `for`-abstractions and `let`-expressions use the key-word `in` in different ways. We want to be able to use layout when writing `for`-abstractions:

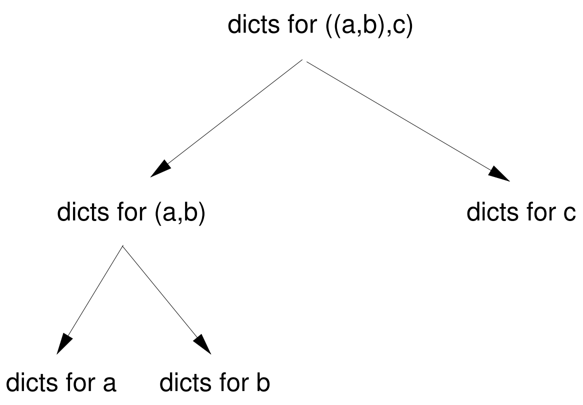

Figure 4.1: Tree of dictionaries for a `forall`-abstraction of type `((a,b),c)`.

```
d = for x in 1 <:> 5  -> 4711
             6 <:> 10 -> 42
```

Which means that `in` should begin a *layout-list*, and the declaration above should be expanded to

```
d = for x in {1 <:> 5 -> 4711 ; 6 <:> 10 -> 42}
```

However, if this is implemented straightforwardly by treating `in` like the other key-words which begin layout-lists (`where`, `let`, `do` and `of`), then `let`-expressions such as

```
e = let x = 4711
    in x*x
```

would be expanded into

```
e = let {x = 4711} in {x*x}
```

which is incorrect (the correct expansion is `e = let {x = 4711} in x*x`).

The solution is to keep track of whether the key word `for` or the key word `let` was the last one encountered and treat `in` accordingly. This makes the lexer a bit more complicated.

4.2 Type checking

Type checking of `forall`- and `for`-abstractions is fairly straightforward. `forall`-abstractions are type checked in a manner which is very close to how lambda-expressions are type checked, the only major difference being the additional requirement of `Ix` and `Pord` contexts for the arguments.

`for`-abstractions are a bit more difficult, having parallels both with lambda-expressions and case-expression, but are still rather straightforward to type check.

However, there is an additional matter which must be taken care of, which has to do with type classes and dictionary arguments. The program transformation from intermediate code with `forall`- and `for`-abstractions to intermediate code without `forall`- and `for`-abstractions will need to use overloaded functions such as `datafield` and `bounds`. We can either make sure the program transformation is performed before the dictionaries are inserted, or make sure the program transformation has access to the dictionaries so it is able to insert them where needed.

NHC inserts dictionary arguments during type checking, so if we want to transform the program before the dictionaries are inserted we can either perform the program transformation before type checking, perform the program transformation during type checking, or move the insertion of dictionaries out of the type checker.

Performing the program transformation before type checking is not a good idea, since it would make it much harder to get good reporting of type errors. Performing the transformation in the type checker would make the type checker messy, and would probably be difficult to implement correctly. Moving the insertion of dictionaries out of the type checker would mean non-trivial changes to the code.

It seems simpler to make sure the program transformation has access to the needed dictionaries, which we achieve by annotating `forall`- and `for`-abstractions with these. The dictionaries we might need are `Pord` and `Ix` dictionaries for the arguments of the `for`- or `forall`-abstraction, as well as `Pord` and `Ix` dictionaries for the components for arguments which are tuples, and for the components of the components if the components are tuples, etc.

This means we in general will have to annotate `forall`-abstractions with a forest of dictionaries, and `for`-abstractions with a tree of dictionaries (since they only have one argument). However, since we later will need `forall`-abstraction to have only one argument as well, we rewrite `forall`-abstractions `forall` $x_1 \ldots x_n$ `->` *exp* to equivalent nested abstractions `forall` x_1 `->` … `->` `forall` x_n `->` *exp*, and annotate the resulting `forall` abstractions with trees of dictionaries.

For an expression `forall x -> ...` with type `((a,b),c)`, the tree of dictionaries created is as shown in Figure 4.1.

4.3 Derivation of `Pord` and `Eval` instances

The derivation of `Pord` and `Eval` instances was straightforward to implement given the infrastructure already in place for deriving instances. The derivation of `Pord` instances follows Figure A.1. The derivation of `Eval` instances is a bit different, since the `seq` class operation can be implemented more efficiently as a primitive operation (which is implemented by using the `EVAL` instruction of the G-machine). Thus `seq` is not derived at all, instead it is provided in the class declaration as a *default method.* A default method for a class operation allows instances to omit methods for this operation. Instead the one provided in the class operation is used. Thus `Eval` is defined as

```
class Eval a where
   ...
   seq a b = seq a b       -- MAGIC
   ...
```

The `seq` on the right-hand side is "magically" replaced by the corresponding primitive operation by the compiler. We make use of this more efficient version of `seq` when deriving `hseq`. We also use the more efficient `handle` primitive operation instead of `isoutofBounds` (for a description of `handle`, see below). For data types where all of the constructors are nullary (i.e have no arguments), the derived method for `hseq` is simply

```
hseq x y = handle (seq x y) y
```

for other data types, the scheme described in Figure A.2 is basically followed (except that `handle` is used there as well).

4.4 The program transformation

The program transformation and the derivation of bounds basically follows the translations in Appendix A. However, there are some differences between the Haskell core and the intermediate code used by NHC, as well as some implementation issues which has to do with the dictionary problem mentioned above.

4.4.1 Pattern matching, `fail` and ▯

While the Haskell core contains `case`- expressions on the form

```
 case v of {K x1 ... xn -> exp ; _ -> exp'}
```

NHC transforms pattern matching in a way similar to the method given in [32], which results in case-expressions containing `fail` and ▯ (alternatively written `fatbar`). This means we must be able to handle `fail` and ▯ in our transformations.

`fail` and ▯ are defined by

$$\begin{array}{llll} \texttt{fail} & \talloblong\ e' & = e' & \\ e & \talloblong\ e' & = e & , e \neq \texttt{fail} \end{array}$$

`fail` may be explicit in the code, but it also represents failed pattern matching. To get an idea of how ▯ and `fail` are used, consider the following example:

```
case v of [x] -> x
            _ -> error "foo"
```

This is transformed by NHC into the intermediate code

```
fatbar
  (case v of (x:v1) ->
     fatbar
       (case v1 of [] -> x)
       fail)
  (error "foo")
```

How should the transformation handle `fail` and $\|$? `fail` represents failed pattern-matching, so the $\mathcal{B}$-scheme (defined in Figure A.4) which defines how bounds are derived, has to be modified so that `fail` is defined as should be handled as `caseNoMatch` in Figure A.4, rule (PFAIL), i.e we should have a rule

$$\mathcal{B}(\texttt{fail}, X, Y) = \texttt{empty}$$

$\|$ chooses between alternatives, so it should be handled like `case`-expressions. We get

$$\mathcal{B}(e_1 \| e_2, \vec{x}, Y) = \mathcal{B}(e_1, \vec{x}, Y) \sqcup \mathcal{B}(e_2, \vec{x}, Y)$$

4.4.2 Dictionaries

As mentioned above, we need to insert `Pord` and `Ix` dictionaries when we transform the program, since we use functions such as `join`, `meet` and `bounds`, as well as the overloaded constants `empty` and `universe`. On some occasions we can steal them from other applications (If we look at the rules in Figure A.4, we see that `bounds`, for example, will only be used when we have an application of the `!` operator, and then we can steal the dictionaries from the application of `!`). But `meet`, `join` and the overloaded constants will be used in situations where it is not possible to find an application of a function which uses the dictionaries. Thus the functions which performs the transformation of a data field expression are given the tree of dictionaries as an extra argument. If the argument of the `forall`-abstraction is a tuple, calls to the function which analyze some sub-expression with respect to some component of the tuple "step down" into the tree, i.e are given the sub-tree corresponding to the component as the extra argument. In this way, we can always find the correct dictionaries in the root of the tree.

4.4.3 Optimizations

When we began the implementation, we decided not to implement any optimizations until the rest of the code was written. When debugging the program transformation, however, we quickly found that the intermediate code was very hard to follow due to the fact that large expressions with lots of `universe` bounds were generated. Simplifying the intermediate code by using the identities

$$\texttt{universe} \sqcap x = x$$

and

$$x \sqcap \texttt{universe} = x$$

was easy to implement and made the intermediate code much easier to read. When this was implemented, the symmetrical simplification using the identities

$$\texttt{empty} \sqcup x = x$$

and

$$x \sqcup \texttt{empty} = x$$

was trivial to implement, so we added it as well.

4.5 `Datafield` and `Bounds` abstract data types

Implementing the abstract data types `Datafield` and `Bounds` in Haskell was a bit troublesome. A simple `Datafield` type is easy to define, given that we have already defined the `Bounds` type:

```
data Datafield a b = Fun (a -> b) (Bounds a)
                   | Tab [(a,b)] (Bounds a)
```

The problem comes when we try to define the `Bounds` type. The problem is how to represent product bounds, created with the `<*>` operator. Product bounds make it possible to have some form of bounds which only are available when the index type (`a` above) is on the form `(b,c)`. An example is the bound `b1 = universe <*> (1<:>10)`

This does not seem to be possible to fulfill if we represent `Bounds` with a data type

```
Bounds a = Dense a a
         | Sparse (Set a)
         | Universe
         | Empty
         | Pred (a -> Bool)
         -- etc ...
```

Here we assume we have an abstract data type `Set a` which represent sets. The implementation of `Set` is described below.

We can not represent products with a constructor `Prod`, since it would have a type on the form `t -> Bounds a`, where the only free type variable in `t` is `a`, and what we want is something with the type `Bounds c -> Bounds b -> Bounds (c,b)`. Another possibility would be to give up, represent products where one of the components is `universe` with `universe`, and represent products where all components are finite by sparse or dense bounds. However, this would give a different semantics from the one given in Appendix A. There is another solution, which admittedly is something of a hack: we implement low-level data types which represents data fields and bounds, but with incorrect types. We then write functions which coerce from the type of these representations to the type we want, and use these coercion functions to implement the functions as defined in section A.1.

Our low-level representations of data fields and bounds are `LowDfield a b c` and `LowBounds a c`, respectively. `c` is an extra type-variable which is used to represent products. It will hopefully become clear how it is used below. `LowDfield` is easy to define once we have `LowBounds`, so we concentrate first on how `LowBounds` and operations on `LowBounds` are defined.

First we define a data type which represents the non-product bounds:

```
data LowBs a = S (Set a)
             | D (a,a)
             | Pr (a -> Bool)
             | Universe
             | Empty
```

then we define

```
data LowBounds a c = B (LowBs a)
                   | P (Bfuns a c) c
```

where the `B` constructor is used to represent non-products, and the `P` constructor is used to represent products. The arguments to the `P` constructor are a *record* of functions of type `Bfuns a c` and something of type `c` (which will typically be a tuple of bounds). The type `Bfuns` is defined as:

```
data Bfuns a c
  = Bfuns { cmeet  :: c -> LowBounds a c -> LowBounds a c,
            cjoin  :: c -> LowBounds a c -> LowBounds a c,
            cinb   :: c -> a -> Bool,
            cenum  :: c -> [a],
            csize  :: c -> Int,
            cupper :: c -> a,
            clower :: c -> a,
            cfin   :: c -> Bool}
```

`Bfuns` could simply be a tuple of functions, but using a record makes the code easier to follow.

The functions in an element of type `Bfuns a c` is used to define functions on the `LowBounds` type when one of the arguments is a product. For example, if we have defined a function `bmeet` which is the ⊓ operation for the non-product bounds in `LowBs`, then we can define the ⊓ operation for `LowBounds` like this:

```
lowMeet (B b1) (B b2) = B (bmeet b1 b2)
lowMeet (P fs c) b    = cmeet fs c b    -- here we get the
                                        -- operation from the
                                        -- record
lowMeet b1 b2         = lowMeet b2 b1
```

The record of functions, `fs` above, is used similarly to how dictionaries are used to implement overloading in Haskell.

The record of functions is created by the product function `lowProd`[1]. `lowProd` has the type

```
lowProd :: (Pord a, Ix a, Pord b, Ix b) =>
           LowBounds a c -> LowBounds b d ->
           LowBounds (a,b) (LowBounds a c, LowBounds b d)
```

and is defined like this:

```
lowProd bx by
  = P (Bfuns cmeet cjoin cinb cenum csize cupper clower cfin)
      (bx,by)
  where cmeet (bx,by) b@(B (S s))
          = lowMeet (lowPred (cinb (bx,by))) b

        cmeet (bx,by) b@(B (Pr p))
          = lowMeet (lowPred (cinb (bx,by))) b

        cmeet (bx,by) (B b)
          = lowProd (lowMeet bx (B (bproj2_1 b)))
                    (lowMeet by (B (bproj2_2 b)))

        cmeet (bx,by) (P _ (bx2,by2))
          = lowProd (lowMeet bx bx2) (lowMeet by by2)
-- etc
```

Here we use *projection functions* `bproj2_1` and `bproj2_2` which, given a non-product bound over a two-dimensional index type, computes a bound

[1]In reality there is a family of functions $\texttt{lowProd}_n$ which contain higher dimensional versions of `lowProd`.

which is the restriction (or an approximation of the restriction) of the original bound to the first and second dimensions, respectively.

With the functions `lowMeet`, `lowJoin`, `lowProd`, etc defined we only need some dummy `Datafield` and `Bounds` types and coercion functions `lowBounds2Bounds`, `lowDfield2Dfield`, `bounds2LowBounds` and `dfield2LowDfield` to define `meet`, `join`, `<*>`, etc.

The coercion functions are easy to define in NHC. All Haskell compilers produce an *interface file* for each compiled module. The interface file contains, among other things (which may differ from compiler to compiler), the types of all exported objects in that module. We put the coercion functions in their own module, defined as identity functions. Then we lie about their type in the interface file (by editing it by hand). Now we can define e.g `meet` as

```
b1 ‘meet‘ b2 = lowBounds2Bounds (lowMeet (bounds2LowBounds b1)
                                         (bounds2LowBounds b2))
```

Representing tabulated data fields

Tabulated data fields are represented either as contiguous arrays or as sparse finite maps. A tabulated data field is represented as an contiguous array if the bounds of the data field are dense, or if the bounds are a product of dense bounds (or a product of a product of dense bounds, etc). Otherwise the tabulated data field is represented as a sparse finite map, the implementation of which is similar to the implementation of sets described below.

4.5.1 Representing sets

When implementing sparse bounds we needed an implementation of sets. We now state what criteria we would like our implementation to fulfill. The operations we need to be able perform efficiently are membership test (for the `inBounds` function), union (for the `join` operation), and intersection (for the `meet` operation). We would also like to have efficient filtering and reduction operations (these are set-versions of the Haskell `filter` and `foldl` functions on lists).

Sorted lists

One possibility is to represent sets as sorted lists. Using sorted lists, we can implement the union and intersection operations with running time $O(n + m)$, where n and m are the sizes of the sets. This is achieved by implementing union and intersection as variations on the classic “merge” function. For example, intersection can be defined like this:

```
intersect [] _              = []
intersect _  []          = []
intersect (x:xs) (y:ys) =
  case compare x y of
     LT -> intersect xs (y:ys)
     EQ -> x : intersect xs ys
     GT -> intersect ys (x:xs)
```

Filtering and reduction operations are also easy to implement. The `filter` operation is just the standard `filter` operation on lists, since an ordered list with some elements removed is still an ordered list. Reduction can be expressed by the standard fold-operation on lists. However, membership test would take time $O(n)$. This is not good, since membership test probably will be a heavily used operation.

Balanced binary trees

Another approach is to use balanced binary trees. Then we would have $O(\log n)$ membership tests. The "obvious" way of implementing intersection and union takes time $O(n \log m)$, but it is possible to convert between sorted lists and balanced trees in linear time, so we can implement union and intersection by converting the binary tree to a list, performing the operation as described above, and convert back. If we implement the filter and reduction operations similarly we do not need any operations which manipulate the tree directly, which means we do not need any explicit balancing information in the tree (as we need if we use e.g Red-Black trees [8]).

How do we convert between binary trees and sorted lists? We assume binary trees are represented by the following data type:

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

Converting from trees to ordered lists is rather easy, we just have to be careful when using the list concatenation operation `++`, since it takes time linear in its left argument. The following definition will not do:

```
toList Empty = []
toList (Node x l r) = toList l ++ (x : toList r)
```

This is not linear, due to the argument `(toList l)` of `++`. Calculating the exact running time is left as an exercise for the reader.

The following definition avoids using `++` by using an accumulating argument:

```
toList t = tl t []
  where tl Empty xs = xs
        tl (Node x l r) xs = tl l (x : tl r xs)
```

Converting a sorted list to a balanced binary tree is a bit harder. We first give a non-linear version:

```
fromOrdList [] = Empty
fromOrdList xs = Node x (fromOrdList l) (fromOrdList r)
  where (l,(x:r)) = splitAt ((length n) ‘div‘ 2)
```

This is non-linear, since we have to traverse the entire list in each recursion when we calculate the length, as well as traverse half the list when using the `splitAt` function. The call to `length` in each iteration can be removed by calling `length` at the beginning of the recursion and pass the length of the sublists as extra arguments to the calls which builds the subtrees. The call to `splitAt` is possible to remove if we take a similar approach as for `toList` above. The idea is to avoid splitting the list by passing the function which builds the left subtree the entire list, and having this function return the unconsumed part of the list. Then the right subtree is built by a similar call, which is passed the returned list. To make this work, we need an additional argument, `n`, which tells the function how much of the list it is allowed to consume. This is the same as the length of the sublists above. The final function looks like this:

```
fromOrdList xs = fst (fl xs (length xs))
  where fl xs 0 = (Empty, xs)
        fl xs n = (Node x l r, xs'')
          where n' = n ‘div‘ 2
                (l, x:xs') = fl xs  n'
                (r, xs'')  = fl xs' (n - n' - 1)
```

Since this way of building trees create a balanced tree where the size of left subtree always is equal to or one greater than the size of the right subtree, we can do clever things like implementing a `size` operation which runs in time $O((\log n)^2)$, as described in [27].

4.6 Implementation of $*$

The constant `outofBounds` is used to represents the error value $*$ in Haskell. Since some functions which operate on data fields need to be able to test if a value is $*$, we also have a function `isoutofBounds` which returns `True` if the given value evaluates to `outofBounds`. The implementation of `outofBounds` and `isoutofBounds` is exception based. `isoutofBounds` is implemented using the operation `handle`, which is defined to follow the following rule:

```
handle x y = x  -- if x does not evaluate to *
handle x y = y  -- if x does evaluate to *
```

Using `handle`, `isoutofBounds` can be expressed as:

```
isoutofBounds = handle (seq x False) True
```

`handle` is implemented by catching exceptions, and `outofBounds` is implemented by throwing them.

When describing the implementation of `handle` and `outofBounds`, we first give a high-level description using the G-machine as described in [32] (this is a simplified variant of the G-machine described in [19]), and we then give a description of how we implemented the primitives on NHC's implementation of (a variant of) the G-machine.

4.6.1 The G-machine

Formally, the G-machine is a tuple $< S, G, C, D >$, where

S	is a stack of *node names* (or *node pointers*).
G	is the graph, i.e a mapping from node names to *nodes.*
C	is the sequence of G-code being executed.
D	is the dump, a stack of pairs (C, S) of code sequences and stacks.

The types of nodes that can exist in the graph are

INT i	an integer node
CONS n_1 n_2	a CONS node (i.e a node representing a list. n_1 is the head of the list and n_2 is the tail.
AP n_1 n_2	an application node
FUN k C	a function with arity k and G-code sequence C
HOLE	a node which is to be filled in later

Some notation: an empty stack or code-sequence is written [], a stack or code-sequence where the first element is x and the rest of the stack or code-sequence is X is written $x : X$. A graph where node n is an application of n_1 to n_2 is written $G[n = \text{AP}\, n_1\, n_2]$.

As mentioned in section 1.4.2, the G-machine expects its input to be a set of super combinators. Each super combinator is compiled into a sequence of G-code instructions. The code sequence compiled for a super combinator assumes the arguments of the super combinator can be found on the stack.

The instructions in the G-code are defined by state-transitions of the machine. A complete description of all G-code instructions is beyond the scope of this thesis (see [32] for details), but to get an idea of how the G-machine works, we describe the two G-machine instructions which are probably the most important to understand how the G-machine performs evaluation, the `EVAL` and `UNWIND` instructions. The definitions of `EVAL` and `UNWIND` in terms of state transitions of the G-machine are shown in Figure 4.2. We now explain the definitions:

$< n : S, G[n = \text{INT}\, i], \texttt{EVAL} : C, D >$
$\Rightarrow < n : S, G, C, D >$
and similarly for CONS and FUN nodes (with $k > 0$ arguments).

$< v : S, G[v = \text{AP}\, v'\, n], \texttt{EVAL} : C, D >$
$\Rightarrow < v : [], G, \texttt{UNWIND} : [], (S, C) : D >$

$< n : S, G[n = \text{FUN}\, 0\, C'], \texttt{EVAL} : C, D >$
$\Rightarrow < n : [], G, C' : [], (S, C) : D >$

$< n : [], G[n = \text{INT}\, i], \texttt{UNWIND} : [], (S, C) : D >$
$\Rightarrow < n : S, G, C, D >$
and similarly for CONS nodes

$< v : S, G[v = \text{AP}\, v'\, n], \texttt{UNWIND} : [], D >$
$\Rightarrow < v' : v : S, G, \texttt{UNWIND} : [], D >$

$< v_0 : v_1 : \ldots : v_k : S, G[v_0 = \text{FUN}\, k\, C\, ;\, v_i = \text{AP}\, v_{i-1}\, n_i], \texttt{UNWIND} : [], D >$
$\Rightarrow < n_1 : n_2 : \ldots : n_k : v_k : S, G, C, D >$
where i goes from 1 to k

$< v_0 : v_1 : \ldots : v_a : [], G[v_0 = \text{FUN}\, k\, C'], \texttt{UNWIND} : [], (S, C) : D >$
$\Rightarrow < v_a : S, G, C, D >$, if $a < k$

Figure 4.2: State transition for `EVAL` and `UNWIND`.

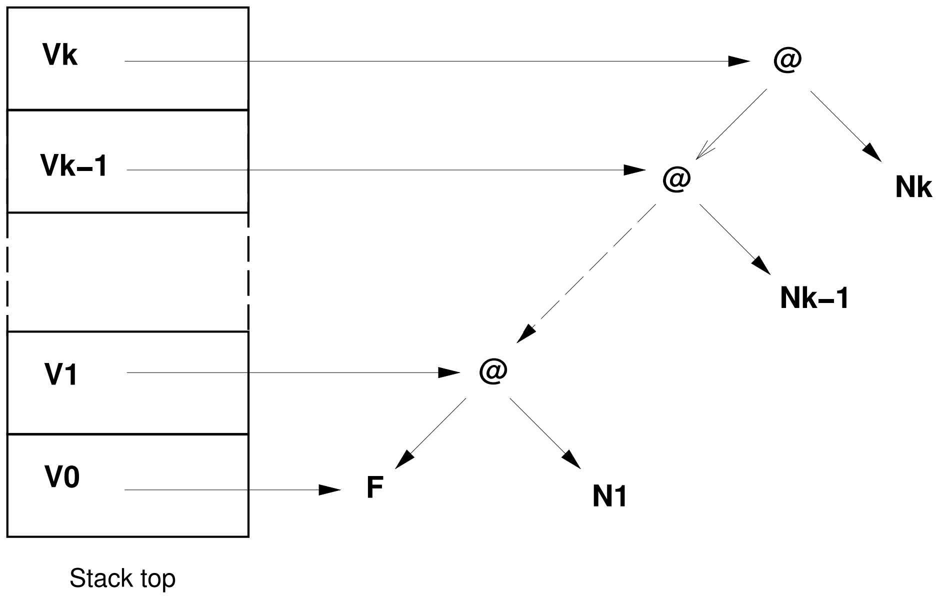

Figure 4.3: The stack before it is rearranged by `UNWIND`.

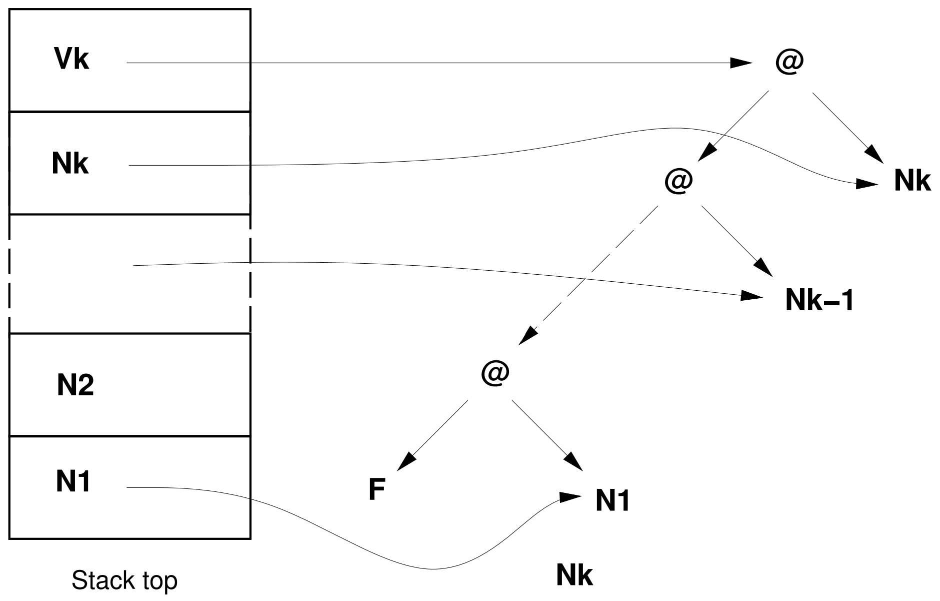

Figure 4.4: The stack after it has been rearranged by `UNWIND`.

`EVAL` is used to initiate evaluation of the node at the top of the stack (actually there is a node name and not a node at the top of the stack. So when we talk about "the node at the top of the stack", we actually mean the node named by the node name at the top of the stack) If the top of the stack is an integer node, a cons node or a function node with non-zero arity, the node is already on whnf, and the execution just continues with the next instruction. If the top of the stack is an application node, the current stack and code are saved on the dump, a new stack with this node as its only element is formed, and `UNWIND` is executed. If the top of the stack is a function node with zero arguments, the current stack and code are saved, a new stack is formed with this node as the only element, and the code associated with the function is executed.

`UNWIND` is used to find the next sub-expression to reduce. `UNWIND` works like this: If the node on the top of the stack is an integer node or a cons node, the node is already on canonical form, so the evaluation is complete. The stack and code are restored from the dump, and the result from the evaluation is put on top of the restored stack. If the node at the top of the stack is an APP node, we just push the left argument of the node onto the top of the stack and execute a new `UNWIND` instruction. If the top of the stack is a function node (as shown in Figure 4.3), we either have enough arguments on the stack for the function to execute or there are too few arguments on the stack. If there are enough arguments, the stack is rearranged so that the arguments of the super combinator are on the stack

$< n_1 : n_2 : S, G, \texttt{HANDLE} : C, D, E >$
$\Rightarrow < n_1 : S, G, \texttt{EVAL} : \texttt{REMOVEHANDLER} : C, D, (n_2, S, C, D) : E >$

$< S, G, \texttt{REMOVEHANDLER} : C, D, t : E >$
$\Rightarrow < S, G, C, D, E >$

$< S, G, \texttt{FAIL} : C, D, (n, S', C', D') : E >$
$\Rightarrow < n : S', \texttt{EVAL} : C', D', E >$

Figure 4.5: State transitions for `HANDLE` , `REMOVEHANDLER` and `FAIL`.

(as shown in Figure 4.4), and we begin executing the code for the function. If there are too few arguments, the expression being evaluated is on whnf (since partial applications are whnfs), so `UNWIND` restores the stack and code from the dump and puts the evaluated value on top of the stack.

4.6.2 Exceptions in the G-machine

To handle exceptions, we add a new component E to the G-machine. E consists of quadruples (n, S, C, D) of a node name, a stack, a code sequence and a dump. We get a new G-machine which is a 5-tuple $< S, G, C, D, E >$. We also need three new instructions: `HANDLE`, `REMOVEHANDLER`, and `FAIL`. The code generated for `outofBounds` is simply

```
FAIL
```

and the code generated for `handle x y` is

<code which puts x on the stack>
<code which puts y on the stack>
`HANDLE`

The idea is to abort the evaluation of `x` if `FAIL` is executed, restore the machine state to what is was before the evaluation of `x` began, and evaluate `y`. To this end, we use the new component E to save the current machine state (which consist of S, C and D) together with y.

The definitions of the instructions as transitions of the modified G-machine are shown in Figure 4.5. We now describe the execution of the instructions:

`HANDLE` sets up n_2 as the expressions to be evaluated on failure by pushing n_2, the current stack (minus the top two elements), the current code sequence and the current dump on the exception stack. Then `HANDLE` makes sure n_1 will evaluated and the post on the exception stack will be removed by adding `EVAL` and `REMOVEHANDLER` to the current code sequence.

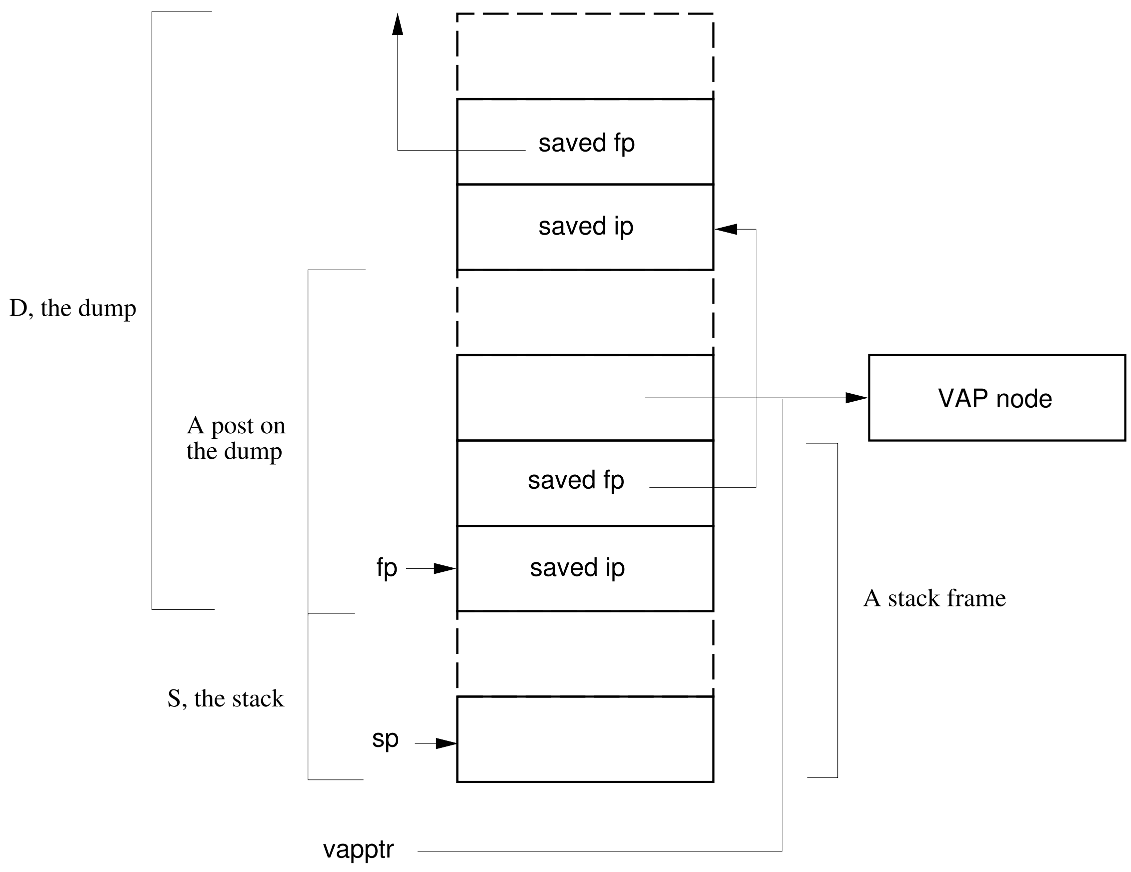

Figure 4.6: The stack in the runtime system for programs compiled with NHC. The stack grows downwards in memory.

`REMOVEHANDLER` removes the top post on the exception stack.

`FAIL` aborts the current execution and continues execution by restoring the code sequence, stack and dump found on the top of the exception stack, and then evaluating the node n found on top of the exception stack.

4.6.3 Exceptions in NHC

NHC makes some modifications to the G-machine as described above. For one thing, there are no binary application (AP) nodes. Instead there are *vector application nodes*, VAP $C\ n_1 \ldots n_k$, where C is a code-pointer and $n_1 \ldots n_k$ are the arguments of the application. Applications of unknown functions are handled by rewriting expressions such as `f x` where `f` is unknown at compile time, to `app f x` where `app` is a function which will (at runtime) evaluate `f` (the result will be a function node with known code) and build a new VAP-node.

The implementation of the G-machine in NHC's run-time system use the stack not only to store arguments to functions and values being operated on (as the G-machine does), but also for the dump (where the state of the machine is saved when executing `EVAL` instructions). The basic layout of the stack is shown in Figure 4.6, as well as how the dump and stack of the G-machine map to the NHC stack. The *frame pointer* `fp` indicates where the G-machine stack begins (the first element is at `fp - 1`). The saved `ip` is a pointer to the code sequence where execution will continue when the current evaluation finished (The C of the top post of the dump in the context of the G-machine). Above this (at $\texttt{fp} + 2$) a pointer to the VAP-node being evaluated can be found. This pointer is also cached in `vapptr`.

With normal compiler terminology, a *stack frame* is a block of the stack which is used for storing the arguments, local variables, and the return address for a function. Thus a stack frame on the NHC stack begins with the saved `fp` and ends either with the top of the stack (if it is the last stack frame) or just before the next saved `fp`.

When implementing exception handling in NHC we change the instruction set slightly. Instead of `HANDLE`, we have an instruction `SETUPHANDLER`, and instead of generating the code

```
<code which puts x on the stack>
<code which puts y on the stack>
HANDLE
```

for the expression `handle x y`, we generate the code

```
<code which puts y on the stack>
SETUPHANDLER
<code which puts x on the stack>
EVAL
REMOVEHANDLER
```

The reason for this is that we do not want to have to insert instructions in the G-code sequence at runtime (which we would have to do if we followed the above description exactly).

To represent the exception stack E we use the NHC stack and a special pointer `failptr` which points to the last *exception frame* on the stack. The exception frame is a new type of stack frame which, together with parts of the previous and next stack frame represents a post on E, as shown in Figure 4.7. The G-machine dump and stack in a post on E is represented by the stack above the exception frame, and the code sequence C in a post on E can be found in the next ordinary stack frame (with the difference that this code sequence includes the `REMOVEHANDLER` instruction).

Note that we save `failptr` in the same position as we save the instruction pointer `ip`. This make the exception frames look just like ordinary frames to the garbage collector. The only thing the garbage collector cares about

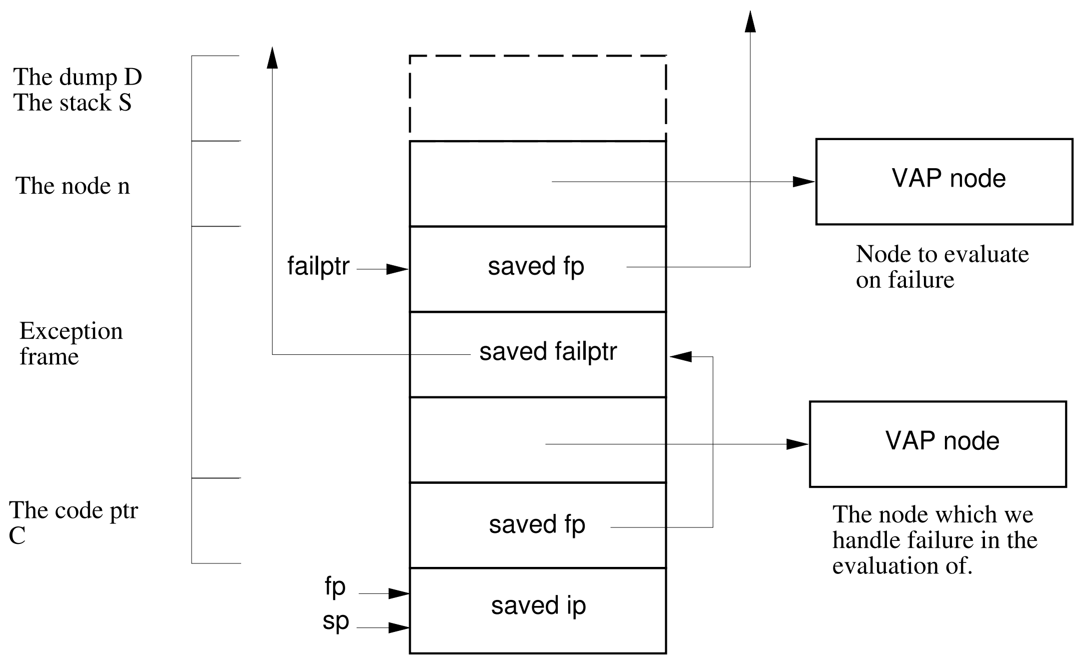

Figure 4.7: Exception frame, next frame, and parts of previous frame just after `EVAL` has been executed.

is that the thing saved in the position above the saved `fp` is not a pointer into the heap.

We now describe the execution of the new instructions.

`SETUPHANDLER`

When `SETUPHANDLER` is executed, the VAP node which should be evaluated on failure is at the top of the stack. A subtle point is that when the exception frame has been created, the node to be evaluated on failure must be a VAP node. This is because a pointer to this node is stored in the same position in the frame as where the system assumes VAP-nodes to be evaluated are stored. Thus the `SETUPHANDLER` has to make sure this node is a VAP node. This is done by creating an application of the identity function `id` on the node, and replacing the node top of the stack with this VAP node. Then `SETUPHANDLER` saves `failptr` and `fp` on the stack and sets `failptr` and `fp` to point to the position of the stack where `failptr` is saved. The stack before the execution of `SETUPHANDLER` is shown in Figure 4.8, and the stack after the execution of `SETUPHANDLER` is shown in Figure 4.9

`REMOVEHANDLER`

When `REMOVEHANDLER` is executed, the evaluation did not fail (as we will see below, the `REMOVEHANDLER` instruction is skipped on failure). So we have

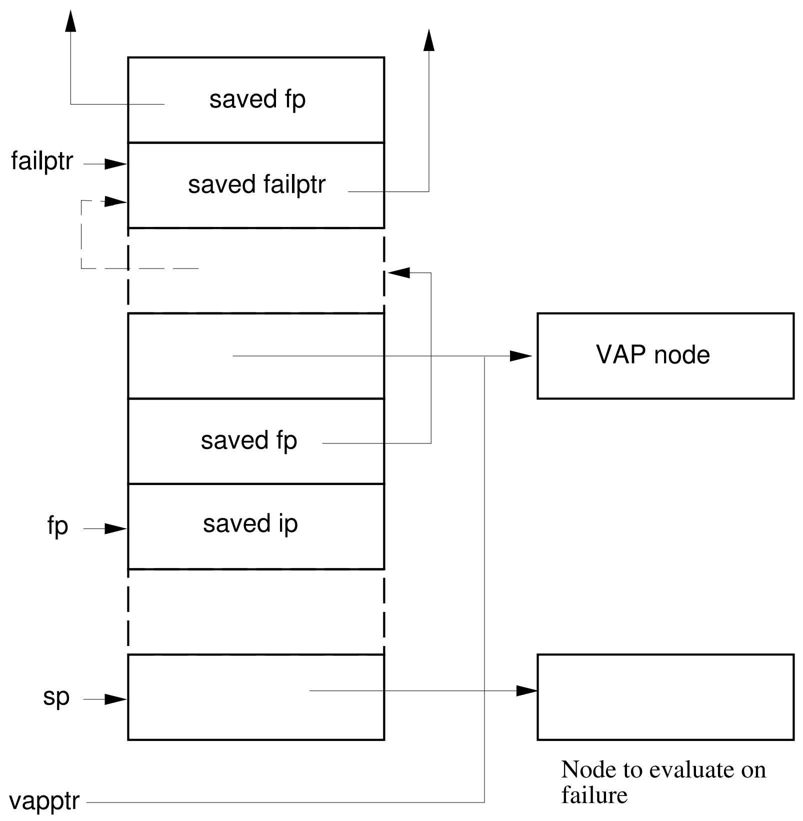

Figure 4.8: The stack before `SETUPHANDLER` is executed.

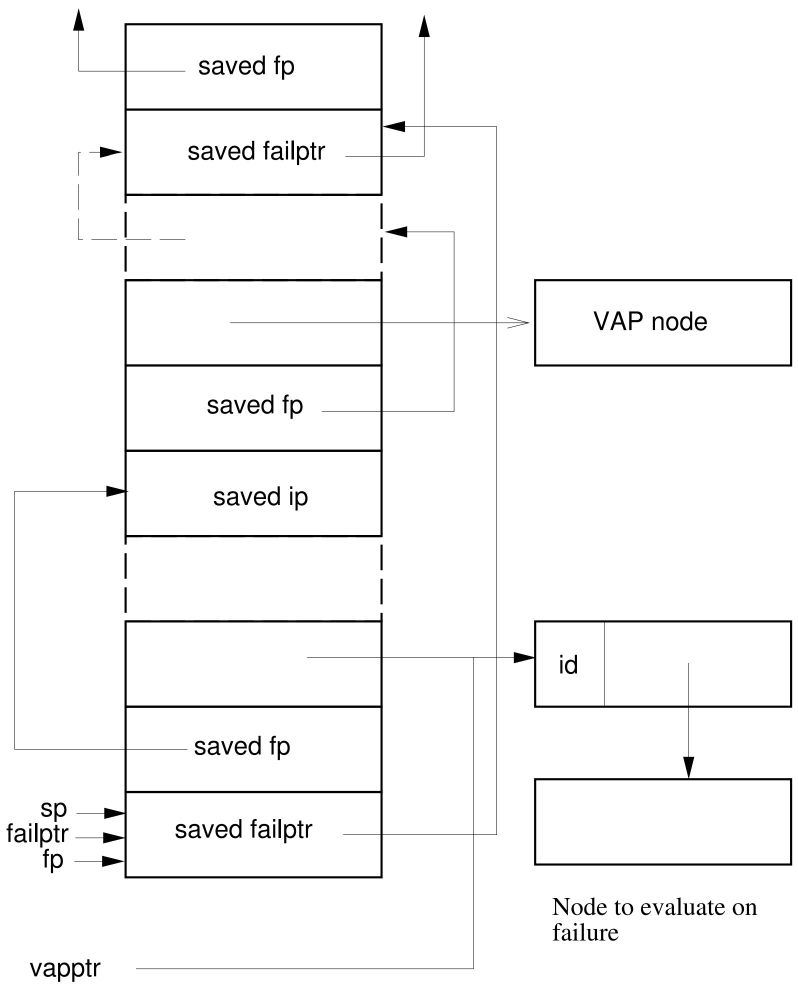

Figure 4.9: The stack after `SETUPHANDLER` is executed.

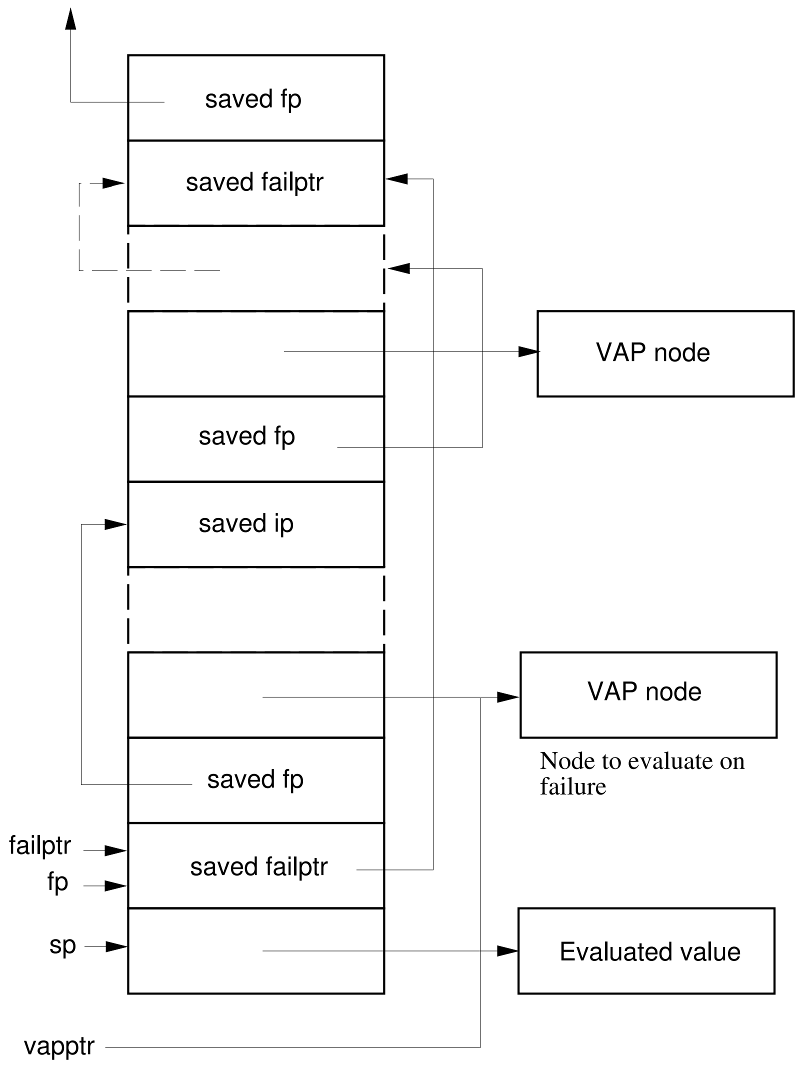

Figure 4.10: The stack before `REMOVEHANDLER` is executed.

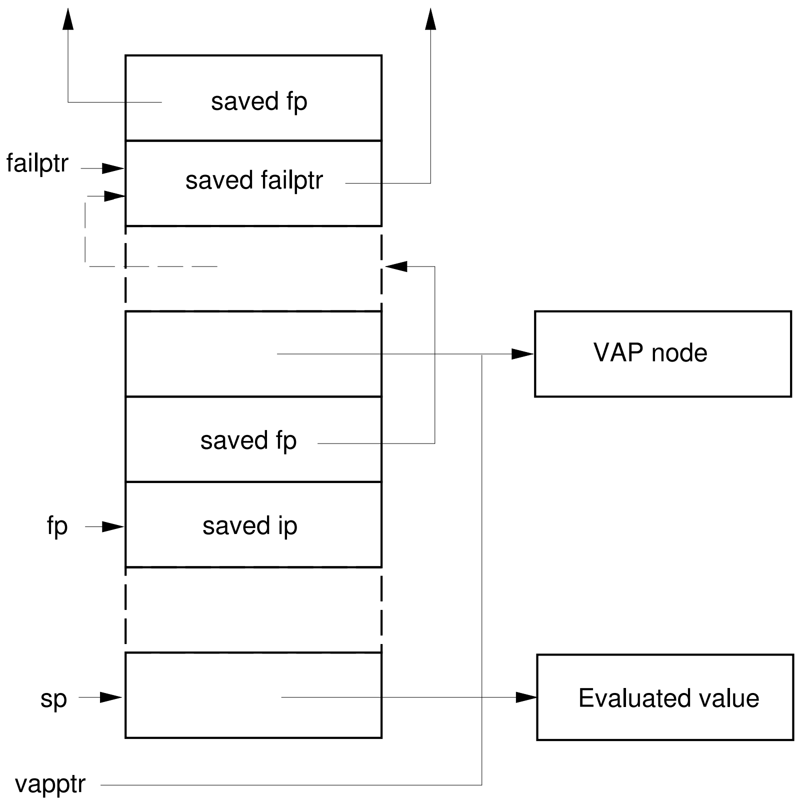

Figure 4.11: The stack after REMOVEHANDLER is executed.

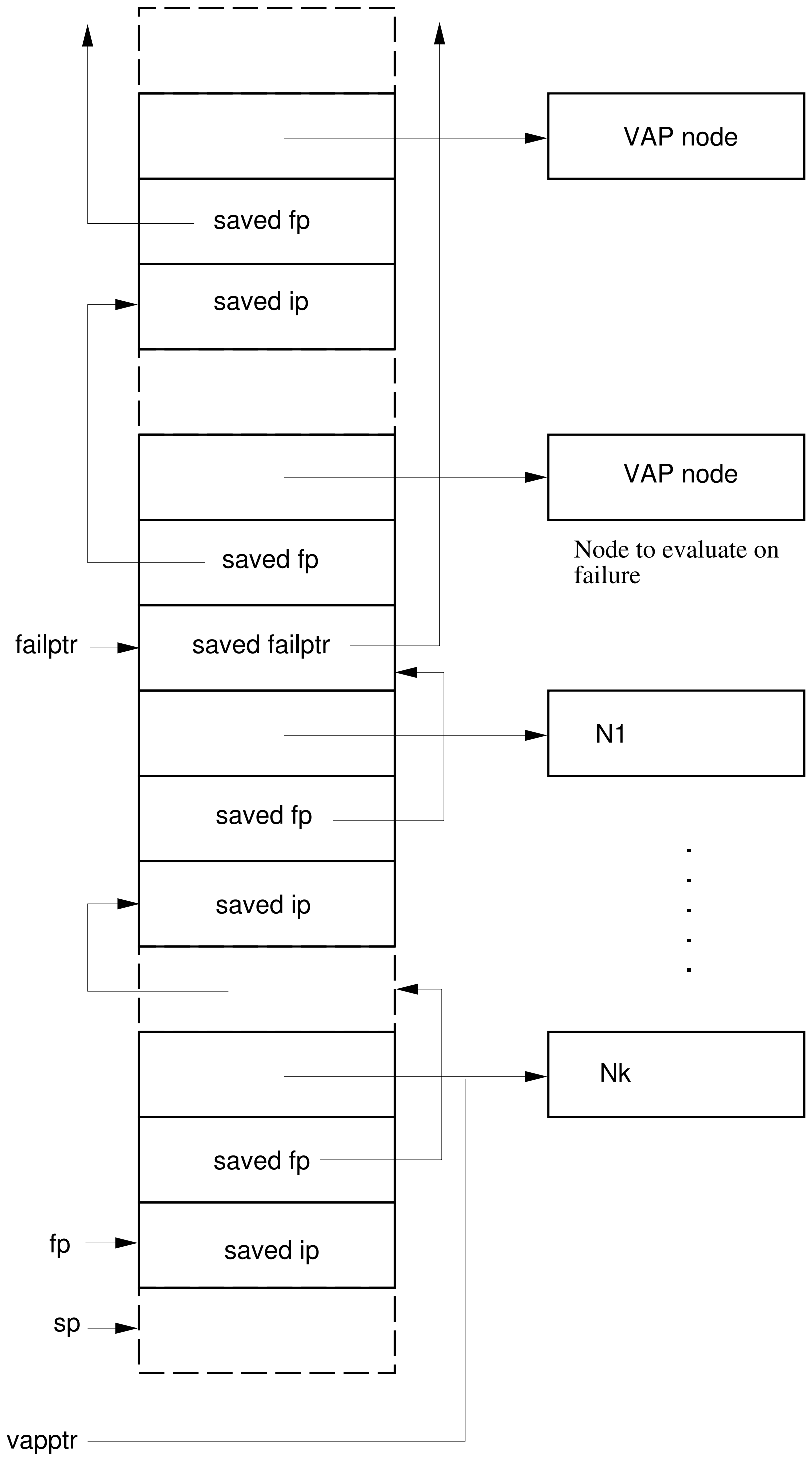

Figure 4.12: The stack before `FAIL` is executed.

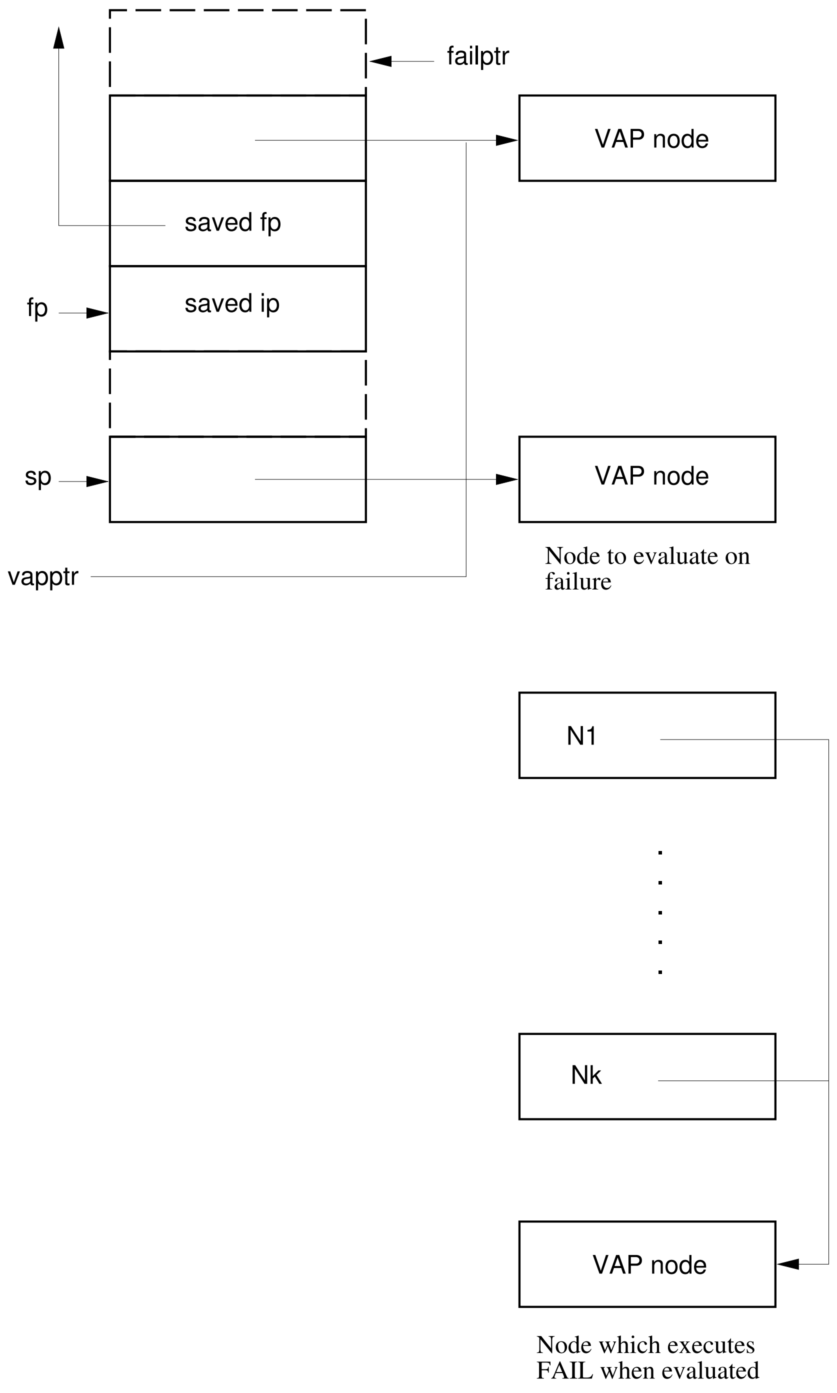
failptr
VAP node
saved fp
saved ip
fp
sp
VAP node
vapptr
Node to evaluate on failure
N1
Nk
VAP node
Node which executes FAIL when evaluated

Figure 4.13: The stack after `FAIL` is executed.

an evaluated value at the top of the stack, and the current stack frame will be an exception frame, as shown in Figure 4.10. `REMOVEHANDLER` removes the exception frame (restoring `failptr` and `fp`) while making sure the evaluated value stays on top of the stack. The stack after the execution of `REMOVEHANDLER` is shown in Figure 4.11.

`FAIL`

`FAIL` is a bit more complicated than the other instructions. This due to the convention of *black holing* VAP-nodes which are being evaluated. When the evaluation of a VAP-node begins, the node is marked as a *black hole.* When the evaluation is finished, the VAP-node is overwritten with the result (which is not a black hole). So far so good. However, if a VAP-node marked as a black hole is evaluated, the compiler issues an error-message and halts, since this means we evaluated a node whose evaluation depends on itself. This is a way of detecting some forms of non-terminating programs at runtime. However, black holed nodes leads to a problem. Figure 4.12 shows the stack and some nodes in the heap before `FAIL` is executed. Here the nodes `N1 ... Nk` are black holes. Basically, what one would like to do is just throw away the part of the stack which lie below `failptr`, and continue evaluation by evaluating the node in the exception frame (making sure `failptr`, `ip` and `fp` are restored correctly, of course). But this will not work, since it would leave black holes in the heap. One solution would be to modify the compiler to that it does not black hole nodes when they are evaluated. We tried this, and the compiled programs worked fine, except that instead of an error message when the space in the heap ran out, we got a segmentation fault. Instead of trying to find the problem, we found a better solution, which is to overwrite the nodes `N1 .. Nk` with indirections to a newly created *failure VAP-node*, a node whose evaluation results in the execution of `FAIL`. The state of the stack (and part of the heap) after the execution of `FAIL` is shown in Figure 4.13.

4.7 Limitations of the implementation

The products and projection functions described above and in Appendix A are hand written, and thus we have to limit the size of tuples for which they are provided. In the present implementation, the size is limited to triples.

Currently the data field extensions will not work with profiling turned on.

Chapter 5

Conclusions and further work

We have presented a dialect of Haskell which makes it possible to program in the data field model. The data field model generalizes the indexed data structures found in many applications, and is based on the intuition of indexed data structures as partial functions. We also gave an overview of the existing Haskell implementations. We came to the conclusion that we should base the implementation of our dialect on either the Glasgow Haskell Compiler (GHC) or Nearly a Haskell Compiler (NHC). In order to decide between them, we compared them more closely, with the result that we decided to base our implementation on NHC. Finally, we described the implementation of our extended version of Haskell. The major parts of the implementation are modifications to the compiler which allows it to parse and type check our extended language, a program transformation which turns intermediate code for our extended language into the original intermediate code for NHC, abstract data types representing the data field and bounds implemented in Haskell, and simple exception handling, used to handle applications of data fields to indices which are out of bounds.

The design of the extensions took rather longer than we anticipated. This might have something to do with the fact that the underlying theoretical model [25] had a tendency to grow more detailed all the time (which on the other hand might have had something to do with the fact that trying to apply the theoretical model raised lots of interesting questions which had to be answered).

Of course, designing a language is not completely divorced from implementing it, since one is reluctant to add language features which one does not have at least a vague idea of how to implement. Thus a lot of the time spent designing the language was really spent thinking of implementation techniques.

It was hard to decide how thoroughly to evaluate the Haskell implementations. We quickly decided to focus on GHC and NHC, but how to proceed was not obvious. A really thorough comparison would include making (or at least trying to make) rather large modifications to the compilers. Understanding a compiler well enough to make such modifications takes a lot of time, and we considered it to be beyond the scope of this Master's project. On the other hand, just looking at the source code does not give a good enough impression of how difficult the program is to modify. We compromised by making a simple modification to the compilers, but the modification might have been to simple.

The actual amount of code required to implement the extensions was rather small (perhaps 3000 lines of Haskell, and under 100 lines of C). The difficult part of writing the code was to make it fit with the existing code and data structures. Learning enough about NHC to be able to write the code was thus a substantial exercise. The most trouble was caused by the implementation of exceptions. The basic principles for how to implement them were not that hard to figure out, but getting the details right took some grueling debugging sessions (as of the writing of this report, we still have some bugs to fix).

Possible improvements of the language

The semantics for our data fields extension became a bit complicated. In particular, the rules for deriving bounds for `forall`-abstractions are complicated. The rules are also a bit too "syntactical". That is, the derived bounds for an expression such as

```
forall x -> f a x
```

will be `universe`, even if `f` is defined as

```
f = (!)
```

It would be nice if the expression `forall x -> f a x` and the expression `forall x -> a!x` would have the same bounds when `f` is defined as above. It is not obvious how to achieve this while keeping the performance of the implementation reasonable, however. One might try to define a new abstract machine which performs some of the transformations at runtime (and thus delay the transformation until we know for sure that `f` is defined as `!`), but this would probably be very inefficient.

Another problem with the derivation of bounds is that the two data fields

```
d1 = forall (x,y) -> a!(x,y)
```

and

```
d2 = forall x -> a!x
```

might get different bounds. For example, if the bound of a is the sparse bound sparse [(1,2),(2,1)], then d1 will have the bound (sparse [1,2]) <*> (sparse [1,2]), while d2 will have the same bound as a. To solve this problem in we need a new kind of sparse bounds, with different semantics (and representation).

We would also like to extend the type system in Haskell such that we could implement the abstract data types for data fields and bounds directly in Haskell. A possible way to represent bounds using type classes has been proposed [22], but to use this approach we need to allow data declarations on the following forms:

```
data T a = T (Class (b a) => b a)
```

and

```
data T (a,b) = ...
```

That is, we basically need to allow the components of data types to be more polymorphic, and to allow the parameter of a polymorphic data type to be a non-variable. Whether these extensions are consistent with the existent type system remains to be determined.

A possible future extension to the language is to incorporate some elemental intrinsic overloading. How to combine this style of overloading with type inference is a research issue, however [48].

Finally, we would like to allow *scaling* as well as translations of data fields. That is, we would like to be able to create a new data field with the even elements of a:

```
evens a = forall x -> a!(2*x)
```

where the bounds of evens are the bounds of a scaled appropriately. In the current language, a would always have the bound universe. This is due to lack of time and to a problem with type classes. The problem is that the type of evens is

```
evens :: (Pord a, Ix a, Num a) =>
         Datafield a b -> Datafield a b
```

and the scaling operation need to perform integer division, which is not an operation of the Num class, but of the Integral class. A simple solution is to require explicit type signatures for the scaling to work, but other and more elegant solutions might be possible.

Possible improvements of the implementation

There are several possibilities for improvement of our implementation. A possibility would be to implement a more advanced simplification of the derived bounds than the one given in section 4.4.3.

It might also be possible to perform more advanced static analysis of data fields, and try to calculate the size of data fields at compile time [24].

It would also be nice to have data fields behave as real memoized functions. In the current implementation, the evaluation of a recursively defined data field will recalculate previously calculated values, instead of using the value already stored in the table. To avoid recalculating the values would probably require a more low-level implementation of the data field evaluators.

Bibliography

[1] Frequently asked questions for comp.lang.functional, September 1998. http://www.cs.nott.ac.uk/Department/Staff/gmh/faq.html.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, April 1989.

[5] Lennart Augustsson. *HBC User's Manual.* Programming Methodology Group, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1993.

[6] Lennart Augustsson. Implementing Haskell overloading. In *Functional Programming and Computer Architecture*, 1993.

[7] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to FORTRAN 90.* Programming Languages. McGraw-Hill, 1990.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[9] A.D. Falkoff and K.E. Iverson. The Design of APL. *IBM Journal of Research and Development*, pages 324–333, July 1973.

[10] Karl-Filip Faxén. *Flow Inference, Code Generation, and Garbage Collection for Lazy Functional Languages.* Licentiate thesis, Dept. of Teleinformatics, KTH, Stockholm, January 1996. Research Report TRITA-IT R 96:01.

[11] Karl-Filip Faxén. *Analyzing, Transforming and Compiling Lazy Functional Programs.* PhD thesis, Dept. of Teleinformatics, KTH, Stockholm, June 1997. Research Report TRITA-IT R 97:08.

[12] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10:349–366, 1990.

[13] Joacim Halén, Per Hammarlund, and Björn Lisper. An experimental implementation of a highly abstract model of data parallel programming. Technical Report TRITA-IT R 97:02, Dept. of Teleinformatics, KTH, Stockholm, March 1997.

[14] Cordelia Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[15] Per Hammarlund and Björn Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 210–222. ACM Press, June 1993.

[16] Kevin Hammond, John Peterson, Lennart Augustsson, Joseph Fasel, Andrew D. Gordon, Simon L. Peyton Jones, and Alastair Reid. Standard libraries for the Haskell programming language, version 1.4, April 1997. http://www.haskell.org/definition/.

[17] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell, version 1.4, March 1997.

[18] Graham Hutton. Higher-order functions for parsing. *J. Functional Programming*, 2(3):323–343, July 1992.

[19] Tomas Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[20] Mark P. Jones and John C. Peterson. Hugs 1.4 user manual. Technical Report NOTTCS-TR-97-1, Department of Computing Science, University of Nottingham, April 1997.

[21] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.

[22] Björn Lisper. Personal Communication.

[23] Björn Lisper. Data parallelism and functional programming. In Guy-Reneé Perrin and Alain Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Vol. 1132 of *Lecture Notes in Comput. Sci.*, pages 220–251, Les Ménuires, France, March 1996. Springer-Verlag.

[24] Björn Lisper and Jean-François Collard. Extent analysis of data fields. In Baudouin Le Charlier, editor, *Proc. International Symposium on Static Analysis*, Vol. 864 of *Lecture Notes in Comput. Sci.*, pages 208–222, Namur, September 1994. Springer-Verlag.

[25] Björn Lisper and Per Hammarlund. The data field model. Submitted, 1998.

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

[27] Chris Okasaki. Three algorithms on braun trees. *J. Functional Programming*, 7(6), November 1997.

[28] L.C Paulson. *ML for the working programmer.* Cambridge University Press, 2 edition, 1996.

[29] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell: A non-strict purely functional language, version 1.4, April 1997. http://www.haskell.org/definition/.

[30] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Simon L. Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell: A non-strict purely functional language, version 1.3, May 1996. http://www.haskell.org/definition/.

[31] John Peterson and Mark Jones. Implementing type classes. In *Proceedings of ACM SIGPLAN Symposium on Programming Language Design and Implementation.* ACM SIGPLAN, June 1993.

[32] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice Hall, 1987.

[33] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *J. Functional Programming*, 2(2):127–202, April 1992.

[34] Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Proc. 6th European Symposium on Programming*, Volume 1058 of *Lecture Notes in Comput. Sci.*, pages 18–44, Linköping, Sweden, April 1996. Springer-Verlag.

[35] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.

[36] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 1993.

[37] Simon L. Peyton Jones and Thomas Nordin. Green card: A foreign-language interface for Haskell. In *Proceedings of the 2nd ACM Haskell Workshop*, Amsterdam, the Netherlands, June 1997.

[38] Niklas Röjemo. *Garbage Collection, and Memory Efficiency, in Lazy Functional Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, 1995.

[39] Niklas Röjemo and Colin Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, Philadelphia, Pennsylvania, May 1996.

[40] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *Functional Programming & Computer Architecture.* ACM, June 93.

[41] André Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Glasgow University, Department of Computing Science, 1995.

[42] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Programming Languages: Implementation, Logics and Programs*, volume 1292 of *Lecture Notes in Computer Science*, pages 291–?? Springer-Verlag, 1997.

[43] The GHC team. The Glasgow Haskell compiler user's guide, version 2.10, 1997.

[44] The GHC team. The Glasgow Haskell compiler user's guide, version 4.00, 1998.

[45] Thinking Machines Corporation, Cambridge, MA. *Connection Machine Model CM-2 Technical Summary*, 1989.

[46] Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in C**, 6.1 edition, 1991.

[47] Thinking Machines Corporation, Cambridge, MA. *Connection Machine: Programming in *Lisp*, 6.1 edition, 1991.

[48] Claes Thornberg and Björn Lisper. Type inference with elemental function overloading. Submitted, 1998.

[49] P. W. Trinder, K. Hammond, S. J. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. GUM : A portable parallel implementation of Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 79–88, New York, May 21–24 1996. ACM Press.

[50] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *Proc. Functional Programming Lang. and Comput. Arch.*, Volume 201 of *Lecture Notes in Comput. Sci.*, pages 1–16. Springer-Verlag, September 1985.

[51] Malcolm Wallace and Colin Runciman. Heap compression and binary I/O in Haskell. In *Proceedings of the 2nd ACM Haskell Workshop*, Amsterdam, the Netherlands, June 1997.

[52] Glynn Winskel. *The Formal Semantics of Programming Languages—An Introduction.* MIT Press, 1993.

[53] J. Allan Yang and Young-il Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structures*, Montreal, Canada, June/July 1992.

Appendix A

Formal definition of data field extensions to Haskell

A.1 The Data Field module

```
module Datafield(Datafield, Bounds, datafield, (!), bounds, (<\>),
                 translate, (<:>), (<*>), universe,  empty, sparse,
                 predicate, finite, enumerate, inBounds, size,
                 lowerBound,  upperBound, transBound, join, meet,
                 prod_2, prod_3, prod_4, --etc
                 tab, strictTab, hstrictTab, foldrDf, foldr1Df,
                 foldlDf, foldl1Df, scanl1Df, scanr1Df,
                 matrix, unmatrix, assoctoDf, outofBounds,
                 isoutofBounds, module Pord) where
infixl 9 !
infixl 3 <*>
infix  2 <:>
infixr 1 <\>

-- Creation of data fields
datafield :: (Pord a,Ix a) => (a -> b) -> Bounds a -> Datafield a b

-- Basic operations on data fields
(!)         :: (Pord a,Ix a) => Datafield a b -> a -> b
bounds      :: (Pord a,Ix a) => Datafield a b -> Bounds a
(<\>)       :: (Pord a,Ix a) =>
               Datafield a b -> Bounds a -> Datafield a b
translate   :: (Pord a,Ix a,Num a) =>
               a -> (Datafield a b) -> (Datafield a b)
```

```
-- Creation of bounds
(<:>)    :: (Pord a,Ix a) => a -> a -> Bounds a
(<*>)    :: (Pord a,Ix a, Pord b, Ix b) =>
           Bounds a -> Bounds b -> Bounds (a,b)
universe  :: (Pord a,Ix a) => Bounds a
empty     :: (Pord a,Ix a) => Bounds a
sparse    :: (Pord a,Ix a) => [a] -> Bounds a
predicate :: (Pord a,Ix a) -> (a -> Bool) -> Bounds a

-- Basic operations on bounds
finite     :: (Pord a,Ix a) => Bounds a -> Bool
enumerate  :: (Pord a,Ix a) => Bounds a -> [a]
inBounds   :: (Pord a,Ix a) => a -> Bounds a -> Bool
size       :: (Pord a,Ix a) => Bounds a -> Int
lowerBound :: (Pord a,Ix a) => Bounds a -> a
upperBound :: (Pord a,Ix a) => Bounds a -> a
transBound :: (Pord a,Ix a,Num a) => a -> Bounds a -> Bounds a

-- join/meet - used by semantics

join :: (Pord a,Ix a) => Bounds a -> Bounds a -> Bounds a
meet :: (Pord a,Ix a) => Bounds a -> Bounds a -> Bounds a

-- Products

prod_2  :: (Pord a,Ix a,Pord b,Ix b) =>
           Bounds a -> Bounds b -> Bounds (a,b)
prod_3  :: (Pord a,Ix a,Pord b,Ix b,Pord c,Ix c) =>
           Bounds a -> Bounds b -> Bounds c -> Bounds (a,b,c)
prod_4  :: (Pord a,Ix a,Pord b,Ix b,Pord c,Ix c,Pord d,Ix d) =>
           Bounds a -> Bounds b -> Bounds c -> Bounds d ->
           Bounds (a,b,c,d)
-- etc

-- Parallel evaluators
tab        :: (Pord a,Ix a) => Datafield a b -> Datafield a b
strictTab  :: (Pord a,Ix a) => Datafield a b -> Datafield a b
hstrictTab :: (Pord a,Ix a) => Datafield a b -> Datafield a b
```

```
-- folds, scans, etc
foldrDf  :: (Pord a,Ix a,Eval c) =>
            (b -> c -> c) -> c -> (Datafield a b) -> c

foldr1Df :: (Pord a,Ix a,Eval a) => (b -> b -> b) -> (Datafield a b) -> b

foldlDf  :: (Pord a,Ix a,Eval b) =>
            (b -> c -> b) -> b -> (Datafield a c) -> b

foldl1Df :: (Pord a,Ix a,Eval a) => (b -> b -> b) -> (Datafield a b) -> b

scanr1Df :: (Pord a,Ix a,Eval a) =>
            (b -> b -> b) -> (Datafield a b) ->  (Datafield a b)

scanl1Df :: (Pord a,Ix a,Eval a) =>
            (b -> b -> b) -> (Datafield a b) -> (Datafield a b)

-- foldDf and scanDf are only guaranteed to work correctly functions
-- which are associative
foldDf :: (Pord a,Ix a,Eval b) => (b -> b -> b) -> (Datafield a b) -> b
scanDf :: (Pord a,Ix a,Eval b) =>
          (b -> b -> b) -> (Datafield a b) -> (Datafield a b)

-- misc

outofBounds   :: a
isoutofBounds :: Eval a => a -> Bool

-- utilites

matrix    :: [[a]] -> Datafield (Int,Int) a
unmatrix  :: Datafield (Int,Int) a -> [[a]]
assoctoDf :: (Pord a,Ix a) -> [(a,b)] -> Datafield a b
```

This module provides the abstract data types `Data Field` and `Bounds` and operations on them. Data Fields can be constructed by applying `datafield` to a function and a expression of the `Bounds`-type.

We give the semantics of the operations as equations between different Haskell-expression. These equations mean that, if we ignore the bounds derived for `forall`-abstractions, the left-hand side can be substituted for the right-hand side, and vice versa. For example, the bound for derived for

```
   forall x -> (datafield f b)!x
```

will not be the same as the bound derived for

```
   forall x -> if inBounds x b then f x else outofBounds
```

but in all other contexts one can be substituted for the other.
The following holds for the basic operations on data fields: :

```
bounds (datafield f b) = b

(datafield f b) ! x =
  if inBounds x b then f x else outofBounds

(datafield f b1) <\> b2 = data field f (b2 ‘meet‘ b1)

(translate a d) ! x = d ! (x-a)
```

Data fields with finite bounds can be tabulated by the `tab`, `strictTab` and `hstrictTab` functions, where `tab` tabulates the data field, but does not evaluate it (as the `array` function in Haskell), `strictTab` tabulates the data field, and evaluates the elements to weak-head normal form (i.e evaluate them to the outermost constructor), and `hstrictTab` tabulates the data field, and evaluates the elements to the innermost constructor.

The following equations hold for the basic operations on bounds:

```
inBounds x universe        = True
inBounds x empty           = False
inBounds x (l <:> u)       = lt l x && lt x u
inBounds x (sparse l)      = x `elem` l
inBounds x (predicate p)   = p x
inBounds (x,y) (b1 <*> b2) = inBounds x b1 && inBounds y b2

finite universe      = False
finite empty         = True
finite (predicate p) = False
finite (sparse l)    = True
finite (l <:> u)     = True
finite (b1 <*> b2)   = finite b1 && finite b2

enumerate (l<:>u)      = range (l,u)
enumerate (sparse l)   = (nub . sort) l
enumerate (b1 <*> b2)
  = [(x,y) | x <- enumerate b1, y <- enumerate b2]
enumerate empty        = []
-- enumerate of an infinite data field is illegal

size b                 = length (enumerate b)
-- size of an infinite data field is illegal

lowerBound (l <:> u)    = l
lowerBound (sparse l)   = foldr1 glb l
lowerBound (b1 <*> b2)  = (lowerBound b1, lowerBound b2)

upperBound (l <:> u)    = u
upperBound (sparse l)   = foldr1 lub l
upperBound (b1 <*> b2)  = (upperBound b1, upperBound b2)

transBound a (l <:> u)     = (l-a) <:> (u-a)
transBound a (sparse l)    = sparse (map (subtract a) l)
transBound a (predicate p) = predicate (p . \x -> x+a)
transBound a universe      = universe
transBound a empty         = empty
```

The following holds for `meet` and `join`:

```
universe ‘meet‘ b  = b
b ‘meet‘ universe  = b
empty ‘meet‘ b     = empty
b ‘meet‘ empty     = empty

(l1 <:> u1) ‘meet‘ (l2 <:> u2)
 = (glb l1 l2) <:> (lub u1 u2)

(bx1 <*> by1) ‘meet‘ (bx2 <*> by2)
 = (bx1 ‘meet‘ bx2) <*> (by1 ‘meet‘ by2)

(bx1 <*> by1) ‘meet‘ ((lx,ly) <:> (ux,uy))
 = (bx1 <*> by1) ‘meet‘ ((lx <:> ux) <*> (ly <:> uy))

((lx,ly) <:> (ux,uy)) ‘meet‘ (bx2 <*> by2)
 = ((lx <:> ux) <*> (ly <:> lx)) ‘meet‘ (bx2 <*> by2)

-- if none of the above equations apply,
-- the following holds:

-- if b1 finite:
b1 ‘meet‘ b2
  = sparse [x | x <- enumerate b1, inBounds x b2]

-- if b2 finite:
b1 ‘meet‘ b2
  = sparse [x | x <- enumerate b2, inBounds x b1]

-- otherwise:
b1 ‘meet‘ b2
  = predicate (\x -> inBounds x b1 && inBounds x b2)

universe ‘join‘ b  = universe
b ‘join‘ universe  = universe
empty ‘join‘ b     = b
b ‘join‘ empty     = b

(l1 <:> u1) ‘join‘ (l2 <:> u2)
  = (lub l1 l2) <:> (glb u1 u2)

(bx1 <*> by1) ‘join‘ (bx2 <*> by2)
 = (bx1 ‘join‘ bx2) <*> (by1 ‘join‘ by2)

(bx1 <*> by1) ‘join‘ ((lx,ly) <:> (ux,uy))
 = (bx1 <*> by1) ‘join‘ ((lx <:> ux) <*> (ly <:> uy))

((lx,ly) <:> (ux,uy)) ‘join‘ (bx2 <*> by2)
 = ((lx <:> ux) <*> (ly <:> uy)) ‘join‘ (bx2 <*> by2)

-- if none of the above equations apply, the following holds:

-- if b1,b2 finite:
b1 ‘join‘ b2
  = sparse (enumerate b1 ++ enumerate b2)

-- otherwise:
b1 ‘join‘ b2
  = predicate (\x -> inBounds x b1 || inBounds x b2)
```

The following holds for outofBounds and isoutofBounds

```
-- outofBounds represent the error value *,
-- which behaves as _|_, except that it can be tested for
-- by isoutofBounds

isoutofBounds x = True  -- if x evaluates to *
isoutofBounds x = False -- otherwise
```

The following holds for the folds

```
foldlDf f a df = foldl f' a (enumerate (bounds df))
  where f' z x  = if isoutfBounds (f z (df!x))
                  then z
                  else (f z (df!x))
-- Similar equations hold for the other folds
```

A.2 Partial orders

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

class Pord a where
   lub   :: a -> a -> a
   glb   :: a -> a -> a
   lt    :: a -> a -> Bool

instance                          Pord Char     where ...
instance                          Pord Int      where ...
instance                          Pord Integer  where ...
instance (Pord a, Pord b) => Pord (a,b)    where ...
-- et cetera
instance                          Pord Bool     where ...
instance                          Pord Ordering where ...
```

The Pord class is used to provide least upper bounds, lub, and greatest lower bounds, glb, according to a *partial order*, lt. That is, lt should be reflexive, antisymmetric and transitive. In addition, the following laws should hold for instances of Pord:

```
x `lt` (lub x y)
y `lt` (lub x y)

x  `lt` u && y `lt` u implies (lub x y) `lt` u
```

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

-- Instances for other tuples are obtained from this scheme:
--
--  instance (Pord a1,..., Pord ak) => Pord (a1,...,ak) where
--      lub (x1,...,xk) (x1',...,xk')
--        = (lub x1 x1',...,lub xk xk')
--      glb (x1,...,xk) (x1',...,xk')
--        = (glb x1 x1',...,lub xk xk')
--      (x1,...,xk) `lt` (x1',...,xk')
--        = x1 `lt` x1' && ... &&  xk `lt` xk'
```

Figure A.1: Derivation of `Pord` instances.

```
(glb x y) `lt` x
(glb x y) `lt` y
```

`` `lt` x && l `lt` y `` implies `` l `lt` (glb x y) ``

For types which are instances of both `Ix` and `Pord` the following should hold:

```
foldr1 glb (range (l,u)) = l
foldr1 lub (range (l,u)) = u
```

The `Pord` class is used in the data field extensions to define pointwise `lub`, `glb` and `lt` for tuples and user defined data types.

A.2.1 Deriving instances of `Pord`

Derived instance declarations for the class `Pord` are possible for enumerations (data types having only nullary constructors), and single constructor data types with constituent types which are instances of `Pord`. For enumeration types, `lub`, `glb` and `lt` are derived in the same way as `max`, `min` and `<=` for the `Ord` class. For single-constructor data types, the derived instance declarations are as shown for tuples in Figure A.1.

A.3 The Show class

```
class Show a  where
  showsPrec        :: Int -> a -> ShowS
  showsPrec'       :: Int -> a -> ShowS
  showList         :: [a] -> ShowS
  showsType        :: a -> ShowS
  showsOutofbounds :: a -> ShowS

  showList xs = handle (showLs xs) (showsOutofbounds (head xs))
    where showLs []     = showString "[]"
          showLs (x:xs) = showChar '[' . shows x .
                           (handle (showl xs)
                                   (showString ", " . showsOutofbounds x))

          showl []     = showChar ']'
          showl (x:xs) = showString ", " . shows x .
                           (handle (showl xs)
                                   (showString ", " . showsOutofbounds x))

  showsPrec p x = handle (showsPrec' p x) (showsOutofbounds x)

  showsOutofbounds _ = showString "<OUB>"
```

To make the printing of the value $*$ configurable for each type, we add a method `showsOutofbounds` to the `Show` class. To minimize the necesary modifications of the instances for the basic types, we add a method `showsPrec'` which should work as `showsPrec` in the Haskell report. The new `showsPrec` is, as default, a wrapper which checks is the result of applying `showPrec'` is $*$, and if so, calls `showsOutofbounds`.

`showsOutofbounds` is, as default, defined as printing `<OUB>`.

```
instance (Eval a1, ..., Eval ak) => Eval (T a1 ... ak) where
  x `seq` y = case x of
                C1 _ ... _ -> y
                        _ -> y

  x `hseq` y =
    let x' = case x of
                 C1 x1 ... xk1 ->
                   x1 `hseq` (x2 `hseq` ... (xk1 `hseq` y)...)
                 C2 x1 ... xk2 ->
                   x1 `hseq` (x2 `hseq` ... (xk2 `hseq` y)...)
                 .
                 .
                 .
                 Cn x1 ... xkn ->
                   x1 `hseq` (x2 `hseq` ... (xkn `hseq` y)...)
    in if isoutofBounds x' then y else y
```

Figure A.2: Derivation of `Eval` instances.

A.4 The `Eval` class

```
infixr 0 `seq`
infixr 0 `hseq`

class Eval a where
   seq         :: a -> b -> b
   strict      :: (a -> b) -> a -> b
   hseq        :: a -> b -> b
   hyperstrict :: (a -> b) -> a -> b

   strict f x  =  x `seq` f x
   hyperstrict f x = x `hseq` f x
```

We add methods `hseq` and `hyperstrict` to the `Eval` class. They are hyperstrict analogs to the `seq` and `strict` methods. However, while `seq` and `strict` are strict both in the error value $*$ and $\perp$, `hseq` and `hyperstrict` are hyperstrict only in $\perp$.

The new `Eval` instance derived by the compiler for a type `T` with constructors `C1`, ..., `Cn` is as shown in Figure A.2

A.5 `forall`-abstraction

`forall`-abstraction is a construct for specifying data field in a implicit way. It is the same as the φ-abstraction in [25]. Syntactically it works as λ-abstraction:

`forall` $apat_1$ ... $apat_n$ `->` exp

As for λ-abstractions, the general form above is equivalent to

```
forall  x1 ... xn -> case  ( x1, ..., xn ) of
                          ( pat1, ..., patn ) -> exp
```

where $x_1, \ldots, x_n$ are fresh identifiers. The types of the identifiers being abstracted over must be instances of the `Pord` and `Ix` classes. So if the type of

`\`x_1 ... x_n `->` $expr$

is

c `=>` a_1 `-> ... ->` a_n `->` t

where the a_i are type variables and c is a context, i.e a list $(C_1 u_1, \ldots C_n u_n)$ where the C_i are class identifiers and the u_i are type variables, then the type of

`forall` x_1 ... x_n `->` $expr$

is

c' `=> Datafield` a_1 `(Datafield` a_2 `(... Datafield` a_n t`))`

where c' is c with type assertions (`Pord` a_1, `Ix` a_1, ... `Pord` a_n, `Ix` a_n) added.

If we have a type of the above form where the a_i are not type variables, the situation is a bit more complicated, since only type variables may appear in contexts. If $a_i = C\, v_1 \ldots v_n$, where C is a type constructor and v_i are type variables, then C must be an instance of `Pord` and `Ix`, and we must add contexts (`Pord` v_1, `Ix` v_1, ..., `Pord` v_n, `Ix` v_n) to c. The above applies recursively if the v_i are not type variables. Thus the expression `forall ((x,y),z) -> x` will have the type

```
(Pord a, Ix a, Pord b, Ix b, Pord b, Ix c) =>
Datafield ((a,b),c) -> a
```

A.5.1 Other modifications to the prelude

The *putChar* and *putStr* functions are modifiued so that they print `<OUB>` if $*$ is encountered. That is:

```
putChar c = if isoutofBounds c then
                putStr "<OUB>"
            else
                primPutchar c

putStr xs = if isoutofBounds xs then
              putStr "<OUB>"
            else
              case xs of
                []     -> return ()
                (x:xs) -> do putChar x
                             putStr xs
```

A.5.2 Semantics of `forall`-abstraction

In [29], the semantics of advanced syntactic structures are given as translation into the Haskell kernel, a very simple subset of Haskell. In this tradition, we give the semantics of `forall`-abstraction as a translation from the Haskell kernel with `forall`-abstractions into the Haskell kernel without `forall`-abstractions. However, for convenience we do not assume that `let`-expressions have been removed. We do assume all `case`-expressions have been "flattened", and written on the form

```
case v of  { K x1 ... xn  -> e ;
             _            -> e' }
```

Where v and $x_1, \ldots, x_n$ are variables.

Semantics for pattern-matching

The formal semantics for `case`-expressions in [29, section 3.17.3] is changed slightly. The rule

```
(b)  case v of { p1  match1;  ... ; pn  matchn }
       =  case v of { p1  match1 ;
                        _  -> ... case v of {
                                    pn  matchn
                                    _  -> error "No match" }...}
```

where each $match_i$ has the form:

```
  | g_{i,1}  -> e_{i,1} ; ... ; | g_{i,m_i} -> e_{i,m_i} where { decls_i }
```

is replaced by

(FORALL1) `forall`$x_1 ... x_n$`->`e
= `forall`x_1`->...->` `forall`x_n`->`e

(FORALL2) `forall`x`->`e
= `datafield(\`x`->`e`)`$\mathcal{B}(e, \{x\}, \emptyset)$

Figure A.3: Translation of `forall`-abstractions.

(b') `case` v `of {` p_1 $match_1$`;` ... `;` p_n $match_n$ `}`
= `case` v `of {` p_1 $match_1$ `;`
`_ -> ... case` v `of {`
p_n $match_n$
`_ -> caseNoMatch }...}`
where each $match_i$ has the form:
`|` $g_{i,1}$ `->` $e_{i,1}$ `; ... ; |` g_{i,m_i} `->` e_{i,m_i} `where {` $decls_i$ `}`

We need `caseNoMatch` to formally distinguish pattern-matching errors from other errors in the semantics for `forall`-abstractions. `caseNoMatch` can be implemented by the compiler as

```
caseNoMatch = error "No match"
```

so this change does not change the semantics of pattern matching in practice.

Translation of `forall`-abstractions

The translation of `forall`-abstractions is given in Figure A.3. A `forall`-abstraction with multiple arguments is first translated into nested `forall`-abstractions with one argument each. The `forall`-abstractions with single arguments are translated into an application of the `datafield` constructor on a λ-abstraction and a bound which is derived from the expression. The rules for how the bounds are derived is based on the rules for how explicit restrictions can be propagated for partial functions. Some examples of this can be found in section 1.2, but for a complete treatment, see [25]. The derivation of the bound is given by the $\mathcal{B}$-scheme, shown in Figure A.4. $\mathcal{B}$ is closely based on the corresponding scheme for φ-abstractions in [25]. There are some differences, however, since the B-scheme given there is more of a operational description of how a φ-expression should be reduced, and requires that part of the expression has been reduced already. We do not want to do any reduction (except translating into the Haskell kernel) before applying the $\mathcal{B}$-scheme. The scheme given in [25] also has a problem with recursive user-defined functions, in that some expressions which intuitively should work as definitions of data fields has no normal form (i.e the evaluation of them will loop forever). We handle this by viewing all functions as strict with respect to the propagation of bounds. Remember that we for strict functions g have

$$\begin{aligned}&\lambda x.g(f_1\backslash b_1\,x)\dots(f_n\backslash b_n\,x) = \\ &\quad(\lambda x.g\,(f_1 x)\dots(f_n x))\backslash(\lambda y.(b_1\,y)\wedge\dots\wedge(b_n\,y))\end{aligned}$$

For propagation of bounds, we apply a similar rule for all functions, regardless of whether they are strict or not. This might give us a bound which is tighter than the domain for the corresponding partial function.

We also handle `case`-expressions, and we give a more explicit definition of what happens when data fields are applied to tuples.

Below, and in Figure A.4, x and v stand for variables, while e and t stand for Haskell core-expressions.

Some notes on the translation: The parameters in $\mathcal{B}(e,\vec{x},Y)$ are an Haskell core expression e, a tuple $\vec{x}$, alternatively written $(x_1,...,x_n)$ (where n might be 1), and a set Y. e is the expression being analyzed. $\vec{x}$ is the argument which we analyze e as a data field over. At the beginning, this is the argument of the `forall`-abstraction, and is thus a single variable, but since `case`-expressions may bind new variables to the components of a tuple, we also need to find the applications of data fields to those variables. This is done by analyzing the sub expression where the binding has effect with respect to the tuple which contains the new variables. The set Y is used to keep track of variables which are bound after the variable being abstracted over. These are needed since we can consider variables which are bound earlier as constants (i.e they can occur in the derived bound).

By abuse of notation, we will write $Y\cup\vec{x}$ for $Y\cup\{x_1,...,x_n\}$.

To keep the description more readable the function `meet` is denoted by $\sqcap$, `join` by $\sqcup$, and `prod_`$n\,e_1...e_n$ is written either as $e_1\times\dots\times e_n$, as $\times_{i=1}^{n}e_i$, or, if all factors are identical, as e^n. We assume that all bound variables are distinct.

We also define a family of *projection functions* on bounds, pr_k^m. Let ρ be a (set-theoretic) partial function from $[1,m]$ to Haskell expressions, and b be a m-dimensional bound (i.e a bound which represents a set of m-tuples). The projection $pr_k^m(\rho,b)$ is the projection of the bound b in the k:th dimension, with additional constraints in the dimensions for which the partial function ρ is defined. We first define pr_k^m for product bounds. Let π_k^m be a family of functions with the property $\pi_k^m(b_1\times\dots\times b_k\times\dots\times b_m)=b_k$, and

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

where

$$\begin{aligned}cond &= v_{i_1}\ \ \texttt{`inBounds`}\ \ \pi_{i_1}^m(b)\ \ \texttt{\&\& ... \&\&}\ \ v_{i_l}\ \ \texttt{`inBounds`}\ \ \pi_{i_l}^m(b)\\ \rho &= \{(i_1,v_{i_1}),...,(i_l,v_{i_l})\}\end{aligned}$$

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

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

For sparse bounds we can define π_k^m as

$$\pi_k^m(\texttt{sparse l}) = \texttt{sparse (map (\textbackslash(}x_1, ..., x_k, ..., x_m\texttt{) ->}x_k\texttt{) l)}$$

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

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

where

$$\texttt{p} = \texttt{(\textbackslash(}x_1, ..., x_m\texttt{) ->}\ x_{i_1}\ \texttt{==}\ v_{i_1}\ \texttt{\&\& ... \&\&}\ x_{i_l}\ \texttt{==}\ v_{i_l}))$$

For predicate bounds, we have

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

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

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

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

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

and

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

We now explain the rules in Figure A.4.

- The (LAM)-rule simply keeps track of variables bound by λ-abstractions.
- The (CASE1)-rule handles the fact that case-expressions can be used to bind variables to the components of a tuple which is a component of the tuple $\vec{x}$. That is, we get a new representation $\vec{v} = (v_1, ..., v_m)$ of the component x_i in $\vec{x}$. This means that we need to consider data field being applied to the variables $v_1, ..., v_n$ as well as the original variables. This is handled by analyzing both over $\vec{v}$ and over $\vec{x}$ and applying $\sqcap$ to the results. Since $\vec{v}$ is a representation of a single component x_i of $\vec{x}$, the expression derived by $\mathcal{B}(e, \vec{v}, Y \cup \vec{x})$ only restricts the bound in the dimension i. This is the reason for the `universe` bounds in the other dimensions.

  Since matching of tuples never fail, we do not need to bother with the other branch of the `case`-expression.
- The (CASE2)-rule handles `case`-expressions where the pattern is not a tuple. This means that (in general) any branch could be taken, which means that the bound derived for the `case`-expression should be $\sqcup$ applied to the bounds of the branches.

- The (APP1)-rule handles applications of data fields, on tuples or non-tuples (a non-tuple is simply considered a tuple of arity 1). One should note that this rule matches syntactically on the `!`-operator. Thus the rule does *not* hold if we replace `!` with `f`, even if `f` is defined as `f = (!)`. The details of data field application is given in the (TUPLE)-rule.

- The (APP2)-rule handles applications of other functions than `!`. Application is strict in the function being applied, so the bounds of the application will depend on the bounds of data fields occurring in the expression which we apply. The bounds of the application may or may not depend on data fields in the argument (for the corresponding rule for partial functions it depends on whether or not the function applied is strict), but for the purpose of the propagation of bounds we assume that all functions are strict, which means bounds from the argument should be propagated.

- The (LET)-rule handles `let` expressions. The (LET)-rule can be seen as a theorem following from the transformations of `let` to λ- and `case`-expressions given in [30] and the other rules given here. But since this is not obvious, we give the rule for (LET) here.

- The (PFAIL)-rule handles pattern-matching failure. We need to distinguish pattern-matching failure from other errors since we otherwise would get the bound `universe` for all `case`-expressions.

- The (AFAIL)-rule should be self-explanatory (an expression which is out of bounds is defined nowhere).

- (DEFAULT) takes care of all cases which do not match any other rule.

- (TUPLE) defines the $\mathcal{T}$-scheme which is used to define data field application on tuples. The bound $\mathcal{T}(b, \vec{t}, \vec{x})$ calculated from the bound b is a product where the i:th component is restricted by the occurrences of x_i in $\vec{t}$. Basically, if x_i occurs in t_k, then the k:th dimension of b might restrict the i:th dimension of the resulting bound. Exactly how depends in what context x_i occurs. The details are given by the (COMP)-rule.

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

A.6 `for`-abstraction

`for`-abstraction specifies the bounds of data fields more explicitly than `forall`-abstraction. The syntax is

```
        for  $pat$ in  { $e_1$ -> $e'_1$ ; ... ; $e_n$ -> $e'_n$ }
```

which is equivalent to

```
(forall $pat$ ->            if inBounds $pat$ ($e_1$) then $e'_1$
                            else if inBounds $pat$ ($e_2$) then $e'_2$
                            ...
                            else if inBounds $pat$ ($e_n$) then $e'_n$
                            else outofbounds) <\>
($e_1$) $\sqcup$ ($e_2$) $\sqcup$ ... $\sqcup$ ($e_n$)
```

(LAM) $\mathcal{B}(\texttt{\textbackslash}\ v_1\ ...\ v_n\ \texttt{->}\ e,\ \vec{x}, Y)$
$= \mathcal{B}(e, \vec{x}, Y \cup \{v_1, ..., v_n\})$

(CASE1) $\mathcal{B}(\texttt{case}\ x_i\ \texttt{of \{(}\ v_1\texttt{,}\ \texttt{...}\texttt{,}\ v_n\ \texttt{) ->}\ e\ \texttt{;}\ \texttt{\_ ->}\ e'\ \texttt{\}},\ \vec{x},\ Y)$
$= \texttt{((universe}\ ^{i-1} \times \mathcal{B}(e, \vec{v}, Y \cup \vec{x}) \times\ \texttt{universe}^{m-i}\texttt{)}$
$\sqcap \mathcal{B}(e, \vec{x}, Y \cup \vec{v}))$

(CASE2) $\mathcal{B}(\texttt{case}\ x\ \texttt{of \{}\ K\ v_1\ ...\ v_n\ \ \texttt{->}\ \ \text{e}\ \texttt{;}\ \texttt{\_ ->}\ \text{e'}\texttt{\}},\ \vec{x}, Y)$
$= \mathcal{B}(e, \vec{x}, Y \cup \{v_1, ..., v_n\}) \sqcup \mathcal{B}(e', \vec{x}, Y)$

(APP1) $\mathcal{B}(\texttt{(!)}\ e\ (t_1, ..., t_m), \vec{x}, Y)$
$= \mathcal{T}(\texttt{bounds}\ e,\ (t_1, ..., t_m),\ \vec{x},\ Y)$, if $\text{FV}(e) \cap (Y \cup \vec{x}) = \emptyset$

(APP2) $\mathcal{B}(e_1\, e_2, \vec{x}, Y)$
$= \mathcal{B}(e_1, \vec{x}, Y) \sqcap (e_2, \vec{x}, Y)$

(LET) $\mathcal{B}(\texttt{let \{}v_1\texttt{=}e_1\texttt{;}...\texttt{;}v_n\texttt{=}e_n\texttt{\} in}\ e, \vec{x}, Y)$
$= \mathcal{B}(e_1, \vec{x}, Y \cup \{v_1, ..., v_n\}) \sqcap ... \sqcap \mathcal{B}(e_n, \vec{x}, Y \cup \{v_1, ..., v_n\})$
$\sqcap \mathcal{B}(e, \vec{x}, Y \cup \{v_1, ..., v_n\})$

(PFAIL) $\mathcal{B}(\texttt{caseNoMatch}, \vec{x}, Y)$
$= \texttt{empty}$

(AFAIL) $\mathcal{B}(\texttt{outofBounds}, \vec{x}, Y)$
$= \texttt{empty}$

(DEFAULT) $\mathcal{B}(e, \vec{x}, Y)$
$= \texttt{universe}$, if none of the other rules apply

(TUPLE) $\mathcal{T}(b, (t_1, ..., t_m), (x_1, ..., x_n), Y)$
$= \times_{i=1}^{n} \sqcap_{k=1}^{m} \mathcal{C}(b, k, (t_1, ..., t_m), x_i, Y \cup \vec{x})$

(COMP) $\mathcal{C}(b, k, (t_1, ..., t_m), x_i, Y)$

$= \texttt{transBound}(pr_k^m(\rho, b), a)$	if $t_k \equiv x_i + a$, where $\text{FV}(a) \cap Y = \emptyset$
$= pr_k^m(\rho, b)$	if $t_k \equiv x_i$
$= \mathcal{T}(pr_k^m(\rho, b), (t'_1, ..., t'_l), x_i, Y \setminus \{x_i\})$	if $t_k \equiv (t'_1, ..., t'_m)$, and $x_i \in \text{FV}(t_k)$
$= \texttt{universe}$	otherwise

where $\rho = \{(j, t_j) \mid \text{FV}(t_j) \cap Y = \emptyset\}$

Figure A.4: Derivation of bounds.