University of Edinburgh
Division of Informatics

Borel - A Bounded Resource Language

4th Year Project Report
Computer Science

Christian Kirkegaard

May 29, 2001

Abstract: This thesis presents a first-order functional language with a linear type system and a special resource type. The language is formally defined together with templates for a translation into effective c-code. It is shown how to design and implement a compiler for the language which generates c-code that manipulates heap-allocated datastructures in an in-place manner. A notion of tail recursion is introduced and this leads to iterative behaviour which optimises the execution time of the generated c-code.

The main feature of the language is that it provides the programmer with explicit control of heap-space manipulations. This introduces the possibility of statically proving a bound on the resources needed by a program. Other state-of-the-art functional languages uses garbage collection to manage heap-space and this results in an unpredictable behaviour making statical proofs impossible.

The proposed language is combining the best of two worlds. It has the strong type system and syntax known from functional languages, and it provides effective heap-space usage known from imperative languages.

Acknowledgements

A big thanks go to Kenco Coffee[1] for producing the excellent instant coffee sold from the JCMB Common Room coffee machine.

An even bigger thanks go to Martin Hofmann[2] for supervising the project. This thesis would not have existed without his help and inspiration.

[1] Kenco web: www.kenco.co.uk
[2] Hofmann web: www.dcs.ed.ac.uk/home/mxh

Contents

1. Introduction

The principal goal of this project is to design and implement a compiler for a functional language with resource control. The functional language will be named Borel (BOunded REsource Language) and it is to be designed as a part of the project. Borel is a first-order functional language with a linear type system and a special resource type. It is meant to extend and improve the language described in [4, 6, 7]. The goal is to define a type system for Borel which will support user defined recursive datatypes and have all the first-order features known from SML and Ocaml. Also, it is the goal to investigate how Borel programs can be optimised for tail recursion.

1.1 Motivation

Functional languages such as SML and Ocaml are generally known to provide simple and easy implementations of programs that manipulate heap-allocated data structures. Functional syntax is nice when it comes to pattern matching and structure building. Also the strong type system of functional compilers will prevent the programmer from making errors. An example illustrating this point is a tree mirror function written in Ocaml syntax:

```
type tree = Leaf of int
          | Node of int * tree * tree

let rec mirror t =
  match t with
    Leaf(i) -> Leaf(i)
  | Node(i,t1,t2) -> Node(i,mirror(t2),mirror(t2))
```

The mirror function is easy to read and understand. Further it is verifiable because of the logical foundation of functional programming.

Now, consider how a functional program will manage heap-allocated datastructures. While executing the program it will sometimes need extra heap-space to hold temporary results and it will sometimes be able to reclaim heap-space which is no longer occupied by data. All this heap-space management is done by a garbage collector. The main problems with this is that garbage collection is quite expensive and completely unpredictable. It is not possible to predict when heap-space is allocated and it is not possible to ensure that all the unoccupied memory is free'ed for further use.

Turning to traditional imperative languages such as C it is much more explicit when heap-space is being used and when it can be reclaimed. This means that most programs containing heap-space allocated datastructures can be implemented with only in-place manipulations (i.e. recycling the heap-space). Given definitions:

```
typedef enum {TREE_LEAF, TREE_NODE} tree_kind_t;

typedef struct {
   tree_kind_t kind;
   union {
      int tree_leaf;
      struct {
         struct tree_t *i;
         struct tree_t *right;
         int left;
      } tree_node;
   } val;
} tree_t;
```

then the following c-code will implement a tree mirror function:

```
tree_t omirror(tree_t t) {
  tree_t left,right;
  void *d1, *d2;
  if (t.kind==TREE_NODE) {
    d1 = (void *)t.val.tree_node.left;
    d2 = (void *)t.val.tree_node.right;
    *(tree_t *)d1 = mirror(*(tree_t *)t.val.tree_node.left);
    *(tree_t *)d2 = mirror(*(tree_t *)t.val.tree_node.right);
    (tree_t *)t.val.tree_node.left = (tree_t *)d2;
    (tree_t *)t.val.tree_node.right = (tree_t *)d1;
  }
  return t;
}
```

There is no allocation of heap-space in the code so it is doing the manipulations in-place. This means that the code is more efficient and that the behaviour when executing is much more predictable than the similar functional code above.

The major problem with imperative functions like mirror is that the code is hard to read and almost impossible to verify. Also, having to do the memory management by hand is a great source of errors. These are serious drawbacks and makes the programming process more tedious and sometimes very frustrating.

Having seen how two known programming languages will fail to provide predictable and efficient code for in-place updates of heap-allocated datastructures leads to look for alternative methods. It would be beneficial if we could have a functional language that guaranteed to modify its arguments in-place, or a method for producing readable and verifiable c-code that manipulates the heap-space in an in-place manner.

The aim of this thesis is to show that there is an alternative with the characteristics mentioned. By developing a compiler which will translate a linearly typed functional language into malloc-free c-code it is possible to have the best of two worlds. The functional programs are easy to write and verify, and since the generated c-code will not allocate heap-space it provides efficient memory management.

1.2 Background

In the paper [7] Martin Hofmann proposes a method for translating a simple linearly typed first-order functional language into c-code which manipulates heap-allocated lists and trees in an in-place manner. It is shown how a linear type system together with a resource type will imply correctness of the translation. There are some hand compiled examples that show how the generated c-code has a functional-like structure. The main result of the paper is showing how to achieve a bound on the amount of heap-space used by a linear functional programming language.

In his thesis [4] Nicholas Brown describes how to design a real programming language that captures the properties proven by Martin Hofmann in [6, 7]. He designed and build a prototype compiler for the language and showed how a number of well known algorithms could be written and compiled. The prototype compiler accepts a language which is limited by the fact that list and tree are the only heap-allocated datastructures. The prototype compiler can be taken as a proof-of-concept implementation of the ideas described above.

We will now take a more detailed look at the proposed method. The in-place heap-space modifications are achieved by introducing a special resource type, $\diamond$, called diamond. A diamond is thought to hold a piece of free heap-space which can be occupied by any heap-allocated value occurring in the program. The clever idea is now that whenever a recursive constructor function is called then it must explicitly be passed enough diamonds to allocate the needed heap-space. And similarly, whenever a recursive datatype is destructed then it will free the diamonds it occupied. The linear typing is needed to ensure that a diamond is only occupied by one heap-allocated value at the time.

A possible representation of the diamond type in c is as a void-pointer which points to a chunk of heap-space. It must of course be ensured that the chunks are appropriately sized so that a diamond can hold any heap-allocated value occurring in a program. The use of a void-pointer as place holder for heap-space was used by the mirror function in the previous section, and it is indeed a technique generally known by c-programmers. However, looking at the mirror function it becomes clear that the use of void-pointers introduce a lot of dangerous c-hacking (eg. type-casting and indirection of pointers). The proposed type system will allow the programmer to focus more on the semantics of the code and then leave it to the compiler to deal with the modification of pointers.

The following extension of the mirror example will illustrate a method for systematically using the diamond type to generate c-code with the wished in-place heap-space manipulations. First define constructor and destructor functions:

```
typedef void *dia_t;

typedef struct {
   tree_kind_t kind;
   union {
      int tree_leaf;
      struct {
         dia_t d1;
         dia_t d2;
         int i;
         tree_t left;
         tree_t right;
      } tree_node;
   } val;
} tree_destr_t;
```

```
tree_t leaf(int i) {
   tree_t res;
   res.kind = TREE_LEAF;
   res.val.tree_leaf = i;
   return res;
}

tree_t node(dia_t d1, dia_t d2, int i
   tree_t left, tree_t right) {
   tree_t res;
   res.kind = TREE_NODE;
   *(tree_t *)d1 = left;
   *(tree_t *)d2 = right;
   res.val.tree_node.i = i;
   res.val.tree_node.left = (struct tree_t *)d1;
   res.val.tree_node.right = (struct tree_t *)d2;
   return res;
}
```

```
tree_destr_t tree_destr(tree_t tree) {
   tree_destr_t res;
   switch (tree.kind) {
      case TREE_LEAF:
         res.kind=TREE_LEAF;
         res.val.tree_leaf=tree.val.tree_leaf;
         break;
      case TREE_NODE:
         res.kind=TREE_NODE;
         res.val.tree_node.i=tree.val.tree_node.i;
         res.val.tree_node.d1=(dia_t )tree.val.tree_node.left;
         res.val.tree_node.d2=(dia_t )tree.val.tree_node.right;
         res.val.tree_node.left=*(tree_t *)tree.val.tree_node.left;
         res.val.tree_node.right=*(tree_t *)tree.val.tree_node.right;
         break;
   }
   return res;
}
```

The definition of `dia_t` is the diamond type represented by a void-pointer. The definition of `leaf(-)` is a constructor function which will take an integer as argument and return a tree consisting of a single leaf. The definition of `node(-)`

is another constructor function. It will take the usual integer and two trees as arguments together with two diamonds which will be used to construct a new node. It is this explicit passing of diamonds to the function which allow it to construct a new node without allocating memory. The definition of `tree_destr_t` is a type which is used to temporarily hold the content of a tree node when it is decomposed. The definition `tree_destr(-)` is a function which will take a tree as argument and then return it as a decomposed tree. It should be noted that decomposing a tree will make two diamonds available for other purposes.

With the definitions above it is straightforward to write a piece of c-code for the tree mirror function. Note how the following c-code has the same overall structure as the Ocaml program in section 1.1:

```
tree_t mirror(tree_t t) {
   tree_t res, t1, t2;
   dia_t d1, d2;
   int i;
   tree_destr_t destr;
   destr = tree_destr(t);
   switch(destr.kind) {
   case TREE_NODE:
      i=destr.val.tree_node.i;
      left=mirror(destr.val.tree_node.left);
      right=mirror(destr.val.tree_node.right);
      d1 = destr.val.tree_node.d1
      d2 = destr.val.tree_node.d2
      res = cons_node(d1,d2,i,right,left);
   case TREE_LEAF:
      i = destr.val.leaf;
      res = cons_leaf(i);
   }
   return res;
}
```

The defined c-code does not allocate heap-space so it is performing the mirror operation by using only in-place manipulations. It should be clear by looking at the code that the definitions that lead to the nice result is not unique for trees. It can be repeated for any recursive datatype, thereby providing a set of guidelines to be used for c-programmers.

Having a method as the one described it is possible to design a system which will take care of automatically generating types and constructor functions. At the same time such a system will guarantee the required in-place constraints. The automated process will relieve the programmer from a lot of tedious work and help him/her to focus on more important aspects of the programming.

The functional language and the compiler described in this thesis is an extension of the work by Martin Hofmann and Nicholas Brown. The language will include user defined datatypes and more functional operators. The compiler will translate the extended language into c-code and optimise for tail-recursion.

1.3 Synopsis

Chapter 1 The introduction sets the goal of the project and gives the motivation for implementing a prototype compiler. Also it presents the previous work which this thesis is improving.

Chapter 2 The Borel language is defined with type rules, and it is shown how to translate it into malloc-free c-code. Finally it is shown formally how to optimise tail recursive functions.

Chapter 3 This chapter outlines the design of a compiler for the Borel language. Each of the stages of the compiler is described together with datastructures and algorithms.

Chapter 4 It is described how the designed compiler was turned into an Ocaml implementation. The explanation includes a more concrete design and all coding problems are described together with their solution.

Chapter 5 The chapter is partly concerned with how the functionally was tested but more important is the comparison with other state-of-the art functional compilers.

Chapter 6 The conclusion where all result are summarised.

2. The Borel Language

On page 1 it is claimed that:

> *Borel is a first-order functional language with a linear type system and a special resource type.*

and now it is time to find out exactly what this means, thereby formalising the ideas from chapter 1.

The term *first-order* refers to *first-order logic*, which is the foundation of the type system and of the language. A first-order functional language is build by a set of zero-order types T which can be thought of as sets of data, and a set of first-order types F which can be thought of as a set of functions. Each type of function in F takes zero-order values as arguments and returns a zero-order value as result.

The term *linear* refers to *affine linear typing* which is a restriction put on the type system. The linearity means that arguments to a function can be used only once in the body of the function. It is this restriction which will guarantee that the programs in-place modifications of datatypes are sound.

The term *resource type* means that the diamond type, $\diamond$, is a build-in zero-order type. The diamond values are the abstractions of free heap-space and such values will be passed around Borel programs allowing the heap-space to be recycled.

In this chapter the Borel language will be represented by a metalanguage. This will be used to formalise typing rules and code templates for a translation into c. The metalanguage will also be used to define tail recursion and it will be used to find optimisations of tail recursive function.

The metalanguage has five zero-order types (see section 2.1) and the first-order types (or functions) are defined by a signature and an expressions of the following kind:

$$\begin{array}{rcl} E & ::= & () \\ & | & c \\ & | & x \\ & | & f(E_1, \ldots, E_n) \\ & | & E[x/y] \\ & | & E_1 \star E_2 \\ & \ldots & \end{array}$$

$$
\begin{array}{ll}
\ldots & \\
| & \mathbf{nil}_T \\
| & \mathbf{cons}(E_h, E_t, E_d) \\
| & \mathbf{snoc}(E_h, E_t, E_d) \\
| & \mathbf{concat}(E_1, E_2) \\
| & d(E_1, \ldots, E_n) \\
| & \mathbf{let}\ x = E_0\ \mathbf{in}\ E_1 \\
| & \mathbf{if}\ E_0\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \\
| & \mathbf{match}\ E_0\ \mathbf{with} \\
 & \quad |\ \mathbf{nil}_T \ \rightarrow\ E_1 \\
 & \quad |\ \mathbf{cons}(x_h, x_t, x_d) \ \rightarrow\ E_2 \\
| & \mathbf{match}\ E_0\ \mathbf{with} \\
 & \quad |\ d_1(x_1, \ldots, x_{m_1}) \ \rightarrow\ E_1 \\
 & \quad |\ \ldots \\
 & \quad |\ d_n(x_1, \ldots, x_{m_n}) \ \rightarrow\ E_n
\end{array}
$$

where the symbol c ranges over constants, x and y over variables, f over functions and d over constructors. The symbol $\star$ is representing the infix operators $+, -, *, /, =, \mathbf{and}, \mathbf{or}, \mathbf{not}$.

The semantics (or meaning) of the language will not be formalised in this report. It seems plausible to claim that the semantics can be defined in the usual set-theoretic way as a fix-point of a suitable operator over E, where zero-order types are interpreted as sets and first-order types as partial functions. An example of how to formalise the semantics of a similar (but simpler) metalanguage can be seen in [7].

In chapter 3 it will be shown how to convert the metalanguage into a real functional programming language.

2.1 Typing rules

The zero-order types are defined by the following grammar:

$$
\begin{array}{rcl}
T & ::= & U \\
 & | & N \\
 & | & \diamond \\
 & | & L(T) \\
 & | & D(d_1.S_1, \ldots, d_n.S_n) \ \text{ where } S_i = (T_{i,1}, \ldots, T_{i,m_i})
\end{array}
$$

The type U is unit, N is number, $\diamond$ is diamond, $L(T)$ is list of type T and $D(d_1.S_1, \ldots, d_n.S_n)$ is datatype.

The heap-free zero-order types are U, N and the datatypes $D(d_1.S_1, \ldots, d_n.S_n)$ where type $T_{i,j}$ is heap-free for all $1 \leq i \leq n$ and $1 \leq j \leq m_i$.

The first-order types are defined to be $F = (T_1, \ldots, T_n) \to T$ where all of $T_1, \ldots, T_n, T$ are zero-order types.

A datatype $D(d_1.S_1 \ldots, d_n.S_n)$ defines n constructors of type:

$$d_i : (T_{i,1}, R_1, \ldots, T_{i,m_i}, R_{m_i}) \to D(S_1 \ldots, S_n)$$

where $T_{i,j}$ is a zero-order type and R_j is either U or $\diamond$ for $1 \leq i \leq n$ and $1 \leq j \leq m_i$.

A signature environment Σ is a partial function from function names to first-order types and a constructor environment Λ is a partial function from constructor names to first order types. It is implicitly assumed that the set of function names and the set of constructor names are disjoint, i.e. $\mathrm{dom}(\Sigma) \cap \mathrm{dom}(\Lambda) = \emptyset$.

A typing context Γ is a finite function from identifiers to zero order types. Whenever $x \notin \Gamma$ then $\Gamma, x : T$ is defined to mean Γ extended with $x \mapsto T$. Whenever $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then Γ, Δ is defined to mean the disjoint union of Γ and Δ. When writing $\Gamma, x : T$ and Γ, Δ it is always implicitly assumed that the disjointness conditions are met.

A Borel program can now be defined to consist of a signature environment Σ, a constructor environment Λ and for each symbol $f : (T_1, \ldots, t_n) \to T$ a defining expression:

$$x_1 : T_1, \ldots, x_n : T_n \vdash_{\Sigma,\Lambda} e_f : T$$

In the following let Γ be a typing context, let Σ be a signature environment and let Λ be a constructor environment. The typing judgement $\Gamma \vdash_{\Sigma,\Lambda} e : T$ is read "the expression e has type T in typing context Γ, signature environment Σ and constructor environment Λ". For the Borel metalanguage $\vdash_{\Sigma,\Lambda}$ is defined by the following typing rules:

$$\frac{\text{c the unit constant}}{\Gamma \vdash_{\Sigma,\Lambda} c : U} \qquad (\textsc{Const}_1)$$

$$\frac{\text{c an integer constant}}{\Gamma \vdash_{\Sigma,\Lambda} c : N} \qquad (\textsc{Const}_2)$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_{\Sigma,\Lambda} x : \Gamma(x)} \qquad (\textsc{Var})$$

$$\frac{\Sigma(f) = (T_1, \dots, T_n) \to T \qquad \Gamma_i \vdash_{\Sigma,\Lambda} e_i : T_i \quad \text{for } i = 1 \dots n}{\Gamma_1, \dots, \Gamma_n \vdash_{\Sigma,\Lambda} f(e_1, \dots, e_n) : T} \tag{\textsc{Sig}}$$

$$\frac{\Gamma, x : T_1, y : T_1 \vdash_{\Sigma,\Lambda} e : T_2 \qquad \mathrm{T}_1 \text{ is heap-free}}{\Gamma, x : T_1 \vdash_{\Sigma,\Lambda} e[x/y] : T_2} \tag{\textsc{Contr}}$$

$$\frac{\Gamma_1 \vdash_{\Sigma,\Lambda} e_1 : N \qquad \Gamma_2 \vdash_{\Sigma,\Lambda} e_2 : N}{\Gamma_1, \Gamma_2 \vdash_{\Sigma,\Lambda} e_1 \star e_2 : N} \tag{\textsc{Infix}}$$

$$\Gamma \vdash_{\Sigma,\Lambda} \mathbf{nil}_T : L(T) \tag{\textsc{Nil}}$$

$$\frac{\Gamma_d \vdash_{\Sigma,\Lambda} e_d : \diamond \qquad \Gamma_h \vdash_{\Sigma,\Lambda} e_h : T \qquad \Gamma_t \vdash_{\Sigma,\Lambda} e_t : L(T)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_{\Sigma,\Lambda} \mathbf{cons}(e_h, e_t, e_d) : L(T)} \tag{\textsc{Cons}}$$

$$\frac{\Gamma_d \vdash_{\Sigma,\Lambda} e_d : \diamond \qquad \Gamma_h \vdash_{\Sigma,\Lambda} e_h : T \qquad \Gamma_t \vdash_{\Sigma,\Lambda} e_t : L(T)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_{\Sigma,\Lambda} \mathbf{snoc}(e_h, e_t, e_d) : L(T)} \tag{\textsc{Snoc}}$$

$$\frac{\Gamma_1 \vdash_{\Sigma,\Lambda} e_1 : L(T) \qquad \Gamma_2 \vdash_{\Sigma,\Lambda} e_2 : L(T)}{\Gamma_1, \Gamma_2 \vdash_{\Sigma,\Lambda} \mathbf{concat}(e_1, e_2) : L(T)} \tag{\textsc{Concat}}$$

$$\frac{\begin{array}{c} \Lambda(d) = (T_1, R_1, \dots, T_m, R_m) \to T \qquad \Delta_i \vdash_{\Sigma,\Lambda} e_i : T_i \\ \Gamma_i \vdash_{\Sigma,\Lambda} r_i : R_i \quad \text{where } R_i = \begin{cases} U & \text{if } T_i \text{ heap-free} \\ \diamond & \text{otherwise} \end{cases} \end{array}}{\Delta_1, \dots, \Delta_m, \Gamma_1, \dots, \Gamma_m \vdash_{\Sigma,\Lambda} d(e_1, r_1, \dots, e_m, r_m) : T} \tag{\textsc{Constr}}$$

$$\frac{\Delta \vdash_{\Sigma,\Lambda} e_1 : T_1 \qquad \Gamma, x : T_1 \vdash_{\Sigma,\Lambda} e_2 : T_2}{\Delta, \Gamma, x : T_1 \vdash_{\Sigma,\Lambda} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : T_2} \tag{\textsc{Let}}$$

$$\frac{\Delta \vdash_{\Sigma,\Lambda} e_0 : N \qquad \Gamma \vdash_{\Sigma,\Lambda} e_1 : T \qquad \Gamma \vdash_{\Sigma,\Lambda} e_2 : T}{\Delta, \Gamma \vdash_{\Sigma,\Lambda} \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : T} \tag{\textsc{If}}$$

$$\frac{\begin{array}{c} \Delta \vdash_{\Sigma,\Lambda} e_0 : L(T_1) \\ \Gamma \vdash_{\Sigma,\Lambda} e_1 : T_2 \qquad \Gamma, d : \diamond, h : T_1, t : L(T_1) \vdash_{\Sigma,\Lambda} e_2 : T_2 \end{array}}{\Delta, \Gamma \vdash_{\Sigma,\Lambda} \begin{array}{l} \mathbf{match}\ e0\ \mathbf{with} \\ \mid \mathbf{nil}_{T_1} \to e_1 \\ \mid \mathbf{cons}(h, t, d) \to e_2 \end{array} : T_2} \tag{\textsc{Lst-Elim}}$$

$$\frac{\begin{array}{c}\Lambda(d_i) = (T_{i,1}, R_{i,1}, \ldots, T_{i,m_i}, R_{i,m_i}) \to T \\ \Delta \vdash_{\Sigma,\Lambda} e_0 : D(S_1, \ldots, S_n) \\ \Gamma, x_{i,j} : T_{i,j}, r_{i,j} : R_{i,j} \vdash_{\Sigma,\Lambda} e_i : T_2\end{array}}{\Delta, \Gamma \vdash_{\Sigma,\Lambda} \begin{array}{l}\textbf{match } e0 \textbf{ with} \\ \mid \mathbf{d}_1(x_{1,1}, r_{1,1}, \ldots, x_{1,m_1}, r_{1,m_1}) \to e_1 \\ \mid \cdots \\ \mid \mathbf{d}_n(x_{n,1}, r_{n,1}, \ldots, x_{n,m_n}, r_{n,m_n}) \to e_2\end{array} : T_2} \quad (\textsc{Dt-Elim})$$

It can be seen that the disjointness condition on typing contexts ensures linearity of the type system. The linearity comes from the fact that free variables in an arbitrary expression can not in general occur as arguments to more than one operation. The Contr rule allows free variables of heap-free type to break the linearity and thereby occur as argument more than once. Also the If, Lst-Elim and Dt-Elim rules allows the same free variable to occur once in each branch of the **if** or **match** expressions.

2.1.1 Typing example

Typing of expressions is done by building a typing tree where each level corresponds to a rule and where all the leaves are trivially true. If such a typing tree can not be build then it is a type error.

Assume we have typing environments $\Gamma = \{y : N\}$ and $\Delta = \{x : N\}$. The following shows how to type an **if-then-else** expression:

$$\frac{\dfrac{\begin{array}{c}\Delta \vdash_{\Sigma,\Lambda} x : N \\ \Delta \vdash_{\Sigma,\Lambda} 0 : N\end{array}}{\Delta \vdash_{\Sigma,\Lambda} x <> 0 : N} \quad \dfrac{\dfrac{\begin{array}{c}\Gamma \vdash_{\Sigma,\Lambda} y : N \\ z : N \vdash_{\Sigma,\Lambda} z : N\end{array}}{\Gamma \vdash_{\Sigma,\Lambda} y * z[y/z] : N}}{\Gamma \vdash_{\Sigma,\Lambda} y * y : N} \quad \dfrac{1 \text{ integer constant}}{\Gamma \vdash_{\Sigma,\Lambda} 1 : N}}{\Delta, \Gamma \vdash_{\Sigma,\Lambda} \textbf{if } x <> 0 \textbf{ then } y * y \textbf{ else } 1 : N}$$

The use of rule If force Δ and Γ to be disjoint and since y is heap-free the use of rule Contr allows us to have variable y occurring twice in subexpression $y * y$.

2.2 Code templates

For each Borel program $B = \big(\Sigma, \Lambda, (e_f)_{f \in \mathrm{dom}\Sigma}\big)$ the aim is to generate a piece of c-code which will compute the meaning. This is done by first associating c-code to every zero-order type occurring in B and next by associating c-code to all the first-order types f occurring in dom(Σ). Finally the pieces are concatenated into a syntactically correct c-program.

Let Types(B) be the finite set of zero-order types that are mentioned in program B. Assume each type $T \in \text{Types}(B)$ has been assigned an unique identifier $[T]^t$. Also assume all the constructors $d_i \in \text{dom}(\Lambda)$ have been assigned a unique identifier $[d_i]^d$. The piece of c-code associated to a type $T \in \text{Types}(B)$ is referred to by $[\![T]\!]^c$ and it is defined by the code templates below. The c-code for all zero-order types is $[\![\text{Types}(B)]\!]^c$ which is the concatenation of $[\![T]\!]^c$ for $T \in \text{Types}(B)$ ordered so that the generated it is valid (i.e. if a type T_1 occurs as a subexpression of a type T_2 then $[\![T_1]\!]^c$ must occur before $[\![T_2]\!]^c$). For types U, N and $\diamond$ define:

$[\![U]\!]^c =$

```
typedef int [U]^t;
```

$[\![N]\!]^c =$

```
typedef int [N]^t;
```

$[\![\diamond]\!]^c =$

```
typedef void *[N]^t;
```

Let $T = D(S_1, \dots, S_n)$ be a datatype. For each of the n constructors

$$d_i : (T_{i,1}, R_1, \dots, T_{i,m_i}, R_{m_i}) \to D(S_1 \dots, S_n)$$

define $[\![d_i]\!]^c$ to be the following piece of c-code:

$[\![d_i]\!]^c =$

```
[T]^t cons_[d_i]^d
([T_{i,1}]^t val_1, [R_{i,1}]^t r_1, ... ,
[T_{i,m_i}]^t val_m_i, [R_{i,m_i}]^t r_m_i) {
  [T]^t res;
  res.kind = [d_i]^d;
  *([T_{i,1}]^t *)r_1 = val_1;
  ...
  *([T_{i,m_i}]^t *)r_m_i = val_m_i;
  res.val.[d_i]^d_val.val_1 = ([T_{i,1}]^t *)r_1;
  ...
  res.val.[d_i]^d_val.val_m_i = ([T_{i,m_i}]^t *)r_m_i;
  return res;
}
```

The c-code $[\![T]\!]^c$ for datatype $T = D(S_1, \ldots, S_n)$ is defined to be:

```
typedef struct {
  [T]^t_kind kind;
  union {
    struct {
      [T_{1,1}]^t val_1;
      ...
      [T_{1,m_1}]^t val_m_1;
    } [d_1]^d_val;
    ...
    struct {
      [T_{n,1}]^t val_1;
      ...
      [T_{n,m_n}]^t val_m_n;
    } [d_n]^d_val;
  } val;
} [T]^t;

typedef struct {
  [T]^t_kind kind;
  union {
    struct {
      [T_{1,1}]^t val_1;
      [R_{1,1}]^t r_1;
      ...
      [T_{1,m_1}]^t val_m_1;
      [R_{1,m_1}]^t r_m_1;
    } [d_1]^d_val;
    ...
    struct {
      [T_{n,1}]^t val_1;
      [R_{n,1}]^t r_1;
      ...
      [T_{n,m_n}]^t val_m_n;
      [R_{n,m_n}]^t r_m_n;
    } [d_n]^d_val;
  } val;
} [T]^t_destr;
```

```
typedef enum {
  [d_1]^d, ... , [d_n]^d
} [T]^t_kind;

[[d_1]]^c, ... , [[d_n]]^c

[T]^t_destr destr_[T]^t
([T]^t t) {
  [T]^t_destr res;
  res.kind = t.kind;
  switch (t.kind) {
  case [d_1]^d :
    res.val.[d_i]^d_val.val_1 =
      t.val.[d_i]^d_val.val_1;
    res.val.[d_i]^d_val.r_1 =
      ([◇]^t)t.val.[d_i]^d_val.r_1;
    ...
    res.val.[d_i]^d_val.val_m_1 =
      t.val.[d_i]^d_val.val_m_1;
    res.val.[d_i]^d_val.r_m_1 =
      ([◇]^t)t.val.[d_i]^d_val.r_m_1;
    break;
  ...
  case [d_n]^d :
    res.val.[d_i]^d_val.val_1 =
      t.val.[d_i]^d_val.val_1;
    res.val.[d_i]^d_val.r_1 =
      ([◇]^t)t.val.[d_i]^d_val.r_1;
    ...
    res.val.[d_i]^d_val.val_m_n =
      t.val.[d_i]^d_val.val_m_n;
    res.val.[d_i]^d_val.r_m_n =
      ([◇]^t)t.val.[d_i]^d_val.r_m_n;
    break;
  }
  return res;
}
```

For a list $L(T)$ define $[\![L(T)]\!]^c$ to be:

```
typedef enum {
  LIST_NIL, LIST_CONS
} [L(T)]^t_kind;

typedef struct {
  [T]^t head;
  struct [L(T)]^t *tail;
  struct [L(T)]^t *last;
} [L(T)]^t;

[L(T)]^t *lnil_[L(T)]^t() {
  return NULL;
}

[L(T)]^t *lcons_[L(T)]^t
([◇]^t d, [T]^t head, [L(T)]^t *tail) {
  [L(T)]^t *res;
  res = ([L(T)]^t *)d;
  res → head = head;
  res → tail = tail;
  if(tail==NULL)
    res → last = res;
  else
    res → last = res;
  return res;
}

[L(T)]^t *lconcat_[L(T)]^t
([L(T)]^t *l1, [L(T)]^t *l2) {
  if (l1==NULL) return l2;
  if (l2==NULL) return l1;
  l1 → last = l2 → last
  (l1 → last) → tail = l2
  (l2 → last) → tail = NULL
  return l1;
}

[L(T)]^t *lsnoc_[L(T)]^t
([◇]^t d, [T]^t head, [L(T)]^t *tail) {
  [L(T)]^t *tmp;
  tmp = ([L(T)]^t *)d;
  tmp → head = head;
  tmp → tail = NULL;
  tmp → tail = tmp;
  if(tail!=NULL) {
    (tail → last) → tail = tmp;
    (tail → last) = tmp;
  } else
    tail = tmp;
  return tail;
}

typedef struct {
  [L(T)]^t_kind kind;
  [◇]^t dia;
  [T]^t head;
  struct [L(T)]^t *tail;
} [L(T)]^t_destr;

[L(T)]^t_destr *ldestr_[L(T)]^t
([L(T)]^t *l) {
  [L(T)]^t_destr res;
  if (l==NULL)
    res.kind = LIST_NIL;
  else {
    res.kind = LIST_CONS;
    res.head = l → head;
    res.d = (void *)l;
    res.tail = l → tail;
  }
  return res;
}
```

The piece of c-code associated to a first order type $f : (T_1, \ldots, T_n) \to T$ occurring in the signature environment Σ is referred to by $[\![f]\!]^c$. For each $f \in \mathrm{dom}(\Sigma)$ there is a corresponding Borel expression e_f and therefore the c-code $[\![f]\!]^c$, defined by the code-templates below, is generated partly from the signature $\Sigma(f)$ and partly from the expression e_f.

The c-code associated to the signature environment Σ is $[\![\Sigma]\!]^c$ and it is the concatenation of $[\![f]\!]^c$ for $f \in \mathrm{dom}(\Sigma)$ ordered so that the it is valid (i.e. if a first-order name f occurs in a first-order expression e_g then $[\![g]\!]^c$ must occur before $[\![f]\!]^c$, and if f and g are mutual recursive functions then $[\![f]\!]^c$ and $[\![g]\!]^c$ are preceded by function prototypes).

Define the valuation π of typing context Γ to be a function from identifiers $x \in \mathrm{dom}(\Gamma)$ to syntactically correct c-expressions. The meaning of π, π' is the the valuation obtained by extending π with the mappings from π'. Whenever the mappings overlap (i.e. $x \in \mathrm{dom}(\pi) \cap \mathrm{dom}(\pi')$) then x is mapped to the c-expression defined by π'.

Let e be a typed Borel expression so that $\Gamma \vdash_{\Sigma,\Lambda} e : T$. Let π be a valuation on Γ. The c-code associated with expression e is always assigned to a c-variable of type T after evaluation. By writing $[\![e, \mathtt{x}]\!]^c_\pi$ is meant the piece of c-code that first computes the meaning of e under valuation π and then assigns the result to x.

From now on it is assumed that all the c-variables named `v` are fresh-variables. By fresh is meant that inside the c-code $[\![f]\!]^c$ for a function f there is only one occurrence of each `v`. The freshness of variables can easily be obtained by relabelling them with a unique name, but in order to keep the notation for expressions simple this relabelling is done implicitly.

The first-order type $f : (T_1, \ldots, T_n) \to T$ with defining expression e_f is associated to the following c-code:

$$[\![f]\!]^c \quad = \quad \boxed{\begin{array}{l} [T]^t\ \mathtt{f}([T_1]^t\ \mathtt{x\_1}, \ldots, [T_n]^t\ \mathtt{x\_n})\ \{ \\ \quad [T]^t\ \mathtt{res}; \\ \quad [\![e_f, \mathtt{res}]\!]^c_{[x_1 \mapsto \mathtt{x\_1}, \ldots, x_n \mapsto \mathtt{x\_n}]}; \\ \quad \mathtt{return\ res}; \\ \} \end{array}}$$

The following is an inductive definition of $[\![e, \mathtt{x}]\!]^c_\pi$ where it is assumed that all expressions have been typed by the rules in section 2.1.

$$[\![() : U, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\mathtt{v} = 0;}$$

$$[\![c : N, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\mathtt{v} = \mathtt{c};}$$

$$[\![x : T, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\mathtt{v} = \pi(x);}$$

$$[\![f(e_1, \ldots, e_n) : T, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [T_1]^t\ \mathtt{v\_1}; \\ \quad \cdots \\ \quad [T_n]^t\ \mathtt{v\_n}; \\ \quad [\![e_1, \mathtt{v\_1}]\!]^c_\pi \\ \quad \cdots \\ \quad [\![e_n, \mathtt{v\_n}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{f(v\_1,\ \ldots\ ,v\_n)}; \\ \} \end{array}}$$

$$[\![e_1 \star e_2 : N, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [N]^t\ \mathtt{v\_left}; \\ \quad [N]^t\ \mathtt{v\_right}; \\ \quad [\![e_1, \mathtt{v\_left}]\!]^c_\pi \\ \quad [\![e_2, \mathtt{v\_right}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{v\_left} \star \mathtt{v\_right}; \\ \} \end{array}}$$

$$[\![\mathbf{nil}_T : L(T), \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\mathtt{v} = \mathtt{lnil\_}[L(T)]^t\mathtt{()};}$$

$$[\![\mathbf{cons}(e_h, e_t, e_d) : L(T), \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [T]^t\ \mathtt{v\_h}; \\ \quad [L(T)]^t\ \mathtt{v\_t}; \\ \quad [\diamond]^t\ \mathtt{v\_d}; \\ \quad [\![e_h, \mathtt{v\_h}]\!]^c_\pi \\ \quad [\![e_t, \mathtt{v\_t}]\!]^c_\pi \\ \quad [\![e_d, \mathtt{v\_d}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{lcons\_}[L(T)]^t\mathtt{(v\_d,v\_h,v\_t)}; \\ \} \end{array}}$$

$$[\![\mathbf{snoc}(e_h, e_t, e_d) : L(T), \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [T]^t\ \mathtt{v\_h}; \\ \quad [L(T)]^t\ \mathtt{v\_t}; \\ \quad [\diamond]^t\ \mathtt{v\_d}; \\ \quad [\![e_h, \mathtt{v\_h}]\!]^c_\pi \\ \quad [\![e_t, \mathtt{v\_t}]\!]^c_\pi \\ \quad [\![e_d, \mathtt{v\_d}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{lsnoc\_}[L(T)]^t\mathtt{(v\_d,v\_h,v\_t)}; \\ \} \end{array}}$$

$$[\![\mathbf{concat}(e_1, e_2) : L(T), \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [L(T)]^t\ \mathtt{v\_1}; \\ \quad [L(T)]^t\ \mathtt{v\_2}; \\ \quad [\![e_1, \mathtt{v\_1}]\!]^c_\pi \\ \quad [\![e_2, \mathtt{v\_2}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{lconcat\_}[L(T)]^t\mathtt{(v\_1,v\_2)}; \\ \} \end{array}}$$

$$[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : T, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [T_1]^t\ \mathtt{x}; \\ \quad [\![e_1, \mathtt{x}]\!]^c_\pi \\ \quad [\![e_2, \mathtt{v}]\!]^c_{\pi,[x \mapsto \mathtt{x}]} \\ \} \end{array}}$$

$$[\![d(e_1,\ldots,e_n) : T, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [T_1]^t\ \mathtt{v\_1;} \\ \quad \cdots \\ \quad [T_n]^t\ \mathtt{v\_n;} \\ \quad [\![e_1, \mathtt{v\_1}]\!]^c_\pi \\ \quad \cdots \\ \quad [\![e_n, \mathtt{v\_n}]\!]^c_\pi \\ \quad \mathtt{v} = \mathtt{cons\_}[d]^d\mathtt{(v\_1,\ \ldots\ ,v\_n);} \\ \} \end{array}}$$

$$[\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 : T, \mathtt{v}]\!]^c_\pi \quad = \quad \boxed{\begin{array}{l} \{ \\ \quad [N]^t\ \mathtt{v\_0;} \\ \quad [\![e_0, \mathtt{v\_0}]\!]^c_\pi \\ \quad \mathtt{if(v\_0)}\ \{ \\ \quad\quad [\![e_1, \mathtt{v}]\!]^c_\pi \\ \quad \}\ \mathtt{else}\ \{ \\ \quad\quad [\![e_2, \mathtt{v}]\!]^c_\pi \\ \quad \} \\ \} \end{array}}$$

$$
\left[\!\!\left[\begin{array}{l}\textbf{match } e_0 \textbf{ with}\\ \mid \textbf{nil}_T \;\rightarrow\; e_1\\ \mid \textbf{cons}(x_h, x_t, x_d) \;\rightarrow\; e_2\end{array} \quad : T,\; v\right]\!\!\right] =
\boxed{\begin{array}{l}
\{\\
\quad [L(T)]^t \;\; *\texttt{v\_0};\\
\quad [L(T)]^t\texttt{\_destr v\_1};\\
\quad [\![e_0, \texttt{v\_0}]\!]^c_\pi\\
\quad \texttt{v\_1} = \texttt{ldestr\_}[L(T)]^t\texttt{(v\_0);}\\
\quad \texttt{switch(v\_1.kind)}\ \{\\
\quad \texttt{case LIST\_NIL:}\\
\qquad [\![e_1, \texttt{v}]\!]^c_\pi\\
\quad \texttt{break;}\\
\quad \texttt{case LIST\_CONS:}\\
\qquad [\![e_2, \texttt{v}]\!]^c_{\pi,\pi'}\\
\quad \texttt{break;}\\
\quad \}\\
\}
\end{array}}
$$

where the valuation π' is:

$$\pi_i = [x_h \mapsto \texttt{v\_1.head}, \\ x_t \mapsto \texttt{v\_1.tail}, x_d \mapsto \texttt{v\_1.dia}]$$

$$
\left[\!\!\left[\begin{array}{l}\textbf{match } e_0 \textbf{ with}\\ \mid d_1(x_1, \ldots, x_{m_1}) \;\rightarrow\; e_1\\ \mid \cdots\\ \mid d_n(x_1, \ldots, x_{m_n}) \;\rightarrow\; e_n\end{array} \quad : T,\; v\right]\!\!\right] =
\boxed{\begin{array}{l}
\{\\
\quad [T_0]^t \;\; *\texttt{v\_0};\\
\quad [T_0]^t\texttt{\_destr v\_1};\\
\quad [\![e_0, \texttt{v\_0}]\!]^c_\pi\\
\quad \texttt{v\_1} = \texttt{destr\_}[T]^t\texttt{(v\_0);}\\
\quad \texttt{switch(v\_1.kind)}\ \{\\
\quad \texttt{case}\ [d_1]^d :\\
\qquad [\![e_1, \texttt{v}]\!]^c_{\pi,\pi_1}\\
\quad \texttt{break;}\\
\quad \cdots\\
\quad \texttt{case}\ [d_n]^d :\\
\qquad [\![e_2, \texttt{v}]\!]^c_{\pi,\pi_n}\\
\quad \texttt{break;}\\
\quad \}\\
\}
\end{array}}
$$

where the valuations π_i are:

$$\pi_i = [x_1 \mapsto \texttt{v\_1.val.}[d_i]^d\texttt{\_val.val\_1}, \cdots, \\ x_{m_i} \mapsto \texttt{v\_1.val.}[d_i]^d\texttt{\_val.val\_m}_i]$$

2.3 Tail recursion

For a typed Borel program $B = \big(\Sigma, \Lambda, (e_f)_{f\in \mathrm{dom}\Sigma}\big)$ it is now the goal to find a set of tail recursive first-order types in B and then show how these can be translated into iterative pieces of c-code.

Let $f : (T_1, \dots, T_n) \to T$ be a first-order type and let e_f be the defining expression for f. We say that f is linearly tail-recursive if and only if e_f has the form:

$$\textbf{if } a(\overline{x}) \textbf{ then } b(\overline{x}) \textbf{ else } c\big(\overline{x}, f(d_1(\overline{x}), \dots, d_n(\overline{x}))\big) : T \tag{2.1}$$

where $\overline{x} = (x_1 : T_1, \dots, x_n : T_n)$ are free-variables,

$$c(u, v) : ((T_1, \dots, T_n), T) \to T$$

is a map using v only once (i.e. c is linear in v) and $a : N$, $b : T$, and $d_i : T_i$ are typed Borel expressions. It is further required that each non-heap-free x_i is used by at most one of the sub-expressions and that neither $a(\overline{x})$, $b(\overline{x})$ nor $d_i(\overline{x})$'s have recursive calls to f.

A formal description of how to translate linearly tail-recursive first-order types into iterative c-code is described by Theorem 1. It states that it is not always possible to generate iterative code for a linearly tail-recursive first-order type. Only when K, update and apply exist can the meaning be computed by the c-code in 2.2.

A valid transformation on Borel expressions is one which preserves meaning (i.e if $\phi : E \to E$ is a transformation on expressions then e and $\phi(e)$ have the same meanings).

A typed Borel expression $e : T$ is said to be a tail-expression with regard to first-order type $f : (T_1, \dots, T_n) \to T$ if and only if there is a valid transformation of $e : T$ into a linearly tail-recursive expression $e' : T$. An example of a tail-expression wrt. type $f : (N) \to N$ is "$f(x) : N$" since it can be validly transformed into "**if** 0 **then** 42 **else** $f(x) : N$". If further expression $e : T$ meet the conditions in Theorem 1 then it is said to be a computable tail-expression wrt. first-order type f which is written $C_f(e)$.

For a computable tail-expression $C_f(e)$ the following rules shows how the expressions $a(\overline{x})$, $b(\overline{x})$, $d_i(\overline{x})$, the type K and maps $c(u, v)$, $\mathit{update}()$, apply() can be found recursively. Some rules are defined with a condition that should hold before they can be used. This is usually a restriction on the type and identity

Theorem 1 *Let $f : (T_1, \ldots, T_n) \to T$ be a linearly tail-recursive first-order type with defining expression $e_f(\overline{x})$ and $a(\overline{x})$, $b(\overline{x})$, $d_i(\overline{x})$, $c(u, v)$ as in 2.1. Assume there exist a zero-order type K with identity i_K and maps:*

$$\text{apply} : K \times T \to T, \quad \text{update} : K \times (T_1, \ldots, T_n) \to K$$

that satisfies the following predicates:

$$\begin{array}{rl} \forall v \in T. & \text{apply}(i_K, v) = v \\ \forall u \in (T_1, \ldots, T_n),\ v \in T,\ k \in K. & \text{apply}(\text{update}(k, u), v) = \text{apply}(k, c(u, v)) \end{array}$$

The meaning of $e_f(\overline{x})$ can be computed by the following c-code:

$$\begin{array}{l} [N]^t\ \texttt{ctrl}; \\ [T]^t\ \texttt{b} \\ [K]^t\ \texttt{k} = i_K; \\ [\![a(\overline{x}), \texttt{ctrl}]\!]^c_\pi \\ \textbf{while}(\texttt{ctrl})\ \{ \\ \quad \text{Inner} \\ \quad [\![a(\overline{x}), \texttt{ctrl}]\!]^c_\pi \\ \} \\ [\![b(\overline{x}), \texttt{b}]\!]^c_\pi \\ [\![apply(\texttt{k}, \texttt{b}), \texttt{res}]\!]^c_\pi \end{array} \tag{2.2}$$

where the valuation is $\pi = [x_1 \mapsto \texttt{x\_1}, \ldots, x_n \mapsto \texttt{x\_n}]$ and Inner *is the concatenation:*

$$[\![\text{update}(\texttt{k}, \overline{x}), \texttt{k}]\!]^c_\pi\ [\![d_1(\overline{x}), \texttt{x\_1}]\!]^c_\pi\ \ldots\ [\![d_n(\overline{x}), \texttt{x\_n}]\!]^c_\pi$$

of c-statements which updates `k` *and the* `x_i`*'s.*

Proof: Traditional correctness proof. By introducing suitable predicates on the variables it is possible to show by induction that the meaning of f is computed by 2.2. Since the meaning of functions is not defined formally the proof will not be elaborated further.

of a sub-expression. If a rule is written without a condition then it can be used with any sub-expression. The optimisation rules are:

$$C_f\left(f(e_1,\dots,e_n)\right) \Rightarrow \begin{cases} K = \texttt{undef} \\ i_K = \texttt{undef} \\ a(\overline{x}) = 1 \\ b(\overline{x}) = \texttt{undef} \\ c(u,v) = v \\ d_i(\overline{x}) = e_i \\ \text{update}(k,u) = \texttt{undef} \\ \text{apply}(k,v) = \texttt{undef} \end{cases}$$

$$\begin{matrix} C_f\left(e_1 \cdot C'_f(e_2)\right) \Rightarrow \\ \\ K' = N \\ i'_K = 1 \end{matrix} \qquad \begin{cases} K = N \\ i_K = 1 \\ a(\overline{x}) = a'(\overline{x}) \\ b(\overline{x}) = e_1(\overline{x}) \\ c(u,v) = e_1 \cdot c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = e_1(u) \cdot \text{update}'(k,u) \\ \text{apply}(k,v) = k \cdot \text{apply}'(k,v) \end{cases}$$

$$\begin{matrix} C_f\left(f, e_1 \cdot C'_f(e_2)\right) \Rightarrow \\ \\ K' = \texttt{undef} \\ i'_K = \texttt{undef} \end{matrix} \qquad \begin{cases} K = N \\ i_K = 1 \\ a(\overline{x}) = a'(x) \\ b(\overline{x}) = e_1(\overline{x}) \\ c(u,v) = e_1(u) \cdot c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = e_1(u) \cdot k \\ \text{apply}(k,v) = k \cdot v \end{cases}$$

$$\begin{matrix} C_f\left(e_1 + C'_f(e_2)\right) \Rightarrow \\ \\ K' = N \\ i'_K = 0 \end{matrix} \qquad \begin{cases} K = N \\ i_K = 0 \\ a(\overline{x}) = a'(x) \\ b(\overline{x}) = e_1(\overline{x}) \\ c(u,v) = e_1(u) + c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = e_1(u) + \text{update}'(k,u) \\ \text{apply}(k,v) = k + \text{apply}'(k,v) \end{cases}$$

$$C_f\left(f, e_1 + C'_f(e_2)\right) \Rightarrow \quad \begin{cases} K = N \\ i_K = 0 \\ a(\overline{x}) = a'(x) \\ b(\overline{x}) = e_1(\overline{x}) \\ c(u,v) = e_1(u) + c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = e_1(u) + k \\ \text{apply}(k,v) = k + v \end{cases}$$

$$K' = \texttt{undef}$$
$$i'_K = \texttt{undef}$$

$$C_f\left(\mathbf{cons}(e_h, C'_f(e_t), e_d)\right) \Rightarrow \quad \begin{cases} K = L(T) \\ i_K = \mathbf{nil}_T \\ a(\overline{x}) = a'(\overline{x}) \\ b(\overline{x}) = b'(\overline{x}) \\ c(u,v) = \mathbf{cons}(e_h(u), c'(u,v), e_d(u)) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \mathbf{snoc}(e_h(u), \text{update}'(k,u), e_d(u)) \\ \text{apply}(k,v) = \mathbf{concat}(k, \text{apply}(k,v)) \end{cases}$$

$$K' = L(T)$$
$$i'_K = \text{nil}_T$$

$$C_f\left(\mathbf{cons}(e_h, C'_f(e_t), e_d)\right) \Rightarrow \quad \begin{cases} K = L(T) \\ i_K = \mathbf{nil}_T \\ a(\overline{x}) = a'(\overline{x}) \\ b(\overline{x}) = b'(\overline{x}) \\ c(u,v) = \mathbf{cons}(e_h(u), c'(u,v), e_d()u) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \mathbf{snoc}(e_h(u), k, e_d(u)) \\ \text{apply}(k,v) = \mathbf{concat}(k,v) \end{cases}$$

$$K' = \texttt{undef}$$
$$i'_K = \texttt{undef}$$

$$C_f\left(\mathbf{snoc}(e_h, C'_f(e_t), e_d)\right) \Rightarrow \quad \begin{cases} K = L(T) \\ i_K = \mathbf{nil}_T \\ a(\overline{x}) = a'(\overline{x}) \\ b(\overline{x}) = b'(\overline{x}) \\ c(u,v) = \mathbf{snoc}(e_h(u), c'(u,v), e_d(u)) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \mathbf{cons}(e_h(u), \text{update}'(k,u), e_d(u)) \\ \text{apply}(k,v) = \mathbf{concat}(k, \text{apply}(k,v)) \end{cases}$$

$$K' = L(T)$$
$$i'_K = \text{nil}_T$$

$$C_f\left(\mathbf{snoc}(e_h, C'_f(e_t), e_d)\right) \Rightarrow \begin{cases} K = L(T) \\ i_K = \mathbf{nil}_T \\ a(\overline{x}) = a'(\overline{x}) \\ b(\overline{x}) = b'(\overline{x}) \\ c(u,v) = \mathbf{snoc}(e_h(u), c'(u,v), e_d(u)) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \mathbf{cons}(e_h(u), k, e_d(u)) \\ \text{apply}(k,v) = \mathbf{concat}(k,v) \end{cases}$$

$K' = \texttt{undef}$
$i'_K = \texttt{undef}$

$$C_f\left(\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } C'_f(e_2)\right) \Rightarrow \begin{cases} K = T \\ i_K = i_T \\ a(\overline{x}) = (\neg e_0(\overline{x})) \,\text{and}\, a'(\overline{x}) \\ b(\overline{x}) = \textbf{if } (\neg e_0(\overline{x})) \textbf{ then } b'(\overline{x}) \textbf{ else } e_1(\overline{x}) \\ c(u,v) = c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \text{update}'(k,u) \\ \text{apply}(k,v) = \text{apply}'(k,v) \end{cases}$$

$$C_f\left(\textbf{if } e_0 \textbf{ then } C'_f(e_1) \textbf{ else } e_2\right) \Rightarrow \begin{cases} K = T \\ i_K = i_T \\ a(\overline{x}) = e_0 \,\text{and}\, a'(\overline{x}) \\ b(\overline{x}) = \textbf{if } e_0(\overline{x}) \textbf{ then } b'(\overline{x}) \textbf{ else } e_2(\overline{x}) \\ c(u,v) = c'(u,v) \\ d_i(\overline{x}) = d'_i(\overline{x}) \\ \text{update}(k,u) = \text{update}'(k,u) \\ \text{apply}(k,v) = \text{apply}'(k,v) \end{cases}$$

The next two cases are **match**-expressions and since these bind new identifiers it is necessary to redefine Inner so that it creates an extended valuation π, π' which maps the new identifiers to c-expressions. The Inner is in both cases defined to wrap the update of `k` and `x_i`'s in a way corresponding to the code template for `match`. The following shows how to redefine Inner as:

$$\begin{array}{l} [T]^t \;\texttt{v\_0}; \\ [T]^t\texttt{\_destr v\_1}; \\ [\![e_0, \texttt{v\_0}]\!]^c_\pi \\ \texttt{v\_1} = \texttt{ldestr\_}[L(T)]^t\texttt{(v\_0)}; \\ \text{Inner}' \end{array}$$

where Inner$'$ is the old definition. By defining the valuation π' as in section 2.2 it is ensured that the result stated by Theorem 1 (i.e. that the iterative c-code computes the meaning) is still correct. If there are nested **match**-expressions then Inner is redefined recursively in the obvious way.

$$C_f\left(\begin{array}{l}\textbf{match } e_0 \textbf{ with}\\ \mid \textbf{nil}_T \to e_1\\ \mid \textbf{cons}(h,t,d) \to C'_f(e_2)\end{array}\right) \Rightarrow \left\{\begin{array}{lcl} K & = & T\\ i_K & = & i_T\\ a(\overline{x}) & = & \textbf{match } e_0(\overline{x}) \textbf{ with}\\ & & \mid \textbf{nil}_T \to 0\\ & & \mid \textbf{cons}(-,-,-) \to a'(\overline{x})\\ b(\overline{x}) & = & \textbf{match } e_0(\overline{x}) \textbf{ with}\\ & & \mid \textbf{nil}_T \to e_1\\ & & \mid \textbf{cons}(-,-,-) \to b'(\overline{x})\\ c(u,v) & = & \textbf{match } e_0(u) \textbf{ with}\\ & & \mid \textbf{nil}_T \to ()\\ & & \mid \textbf{cons}(-,-,-) \to c'(\overline{x})\\ d_i(\overline{x}) & = & d'_i(\overline{x})\\ \text{update}(k,u) & = & \text{update}'(k,u)\\ \text{apply}(k,v) & = & \text{apply}'(k,v) \end{array}\right.$$

$$C_f\left(\begin{array}{l}\textbf{match } e_0 \textbf{ with}\\ \mid p_1(x_1,\dots,x_{m_1}) \to e_1\\ \mid \cdots\\ \mid p_i(x_1,\dots,x_{m_i}) \to C'_f(e_i)\\ \mid \cdots\\ \mid p_1(x_1,\dots,x_{m_n}) \to e_n\end{array}\right) \Rightarrow \left\{\begin{array}{lcl} K & = & T\\ i_K & = & i_T\\ a(\overline{x}) & = & \textbf{match } e_0(\overline{x}) \textbf{ with}\\ & & \mid p_1(x_1,\dots,x_{m_1}) \to 0\\ & & \mid \cdots\\ & & \mid p_i(x_1,\dots,x_{m_i}) \to a'(\overline{x})\\ & & \mid \cdots\\ & & \mid p_1(x_n,\dots,x_{m_n}) \to 0\\ b(\overline{x}) & = & \textbf{match } e_0(\overline{x}) \textbf{ with}\\ & & \mid p_1(x_1,\dots,x_{m_1}) \to 0\\ & & \mid \cdots\\ & & \mid p_i(x_1,\dots,x_{m_i}) \to b'(\overline{x})\\ & & \mid \cdots\\ & & \mid p_1(x_n,\dots,x_{m_n}) \to 0\\ c(u,v) & = & \textbf{match } e_0(u) \textbf{ with}\\ & & \mid p_1(x_1,\dots,x_{m_1}) \to 0\\ & & \mid \cdots\\ & & \mid p_i(x_1,\dots,x_{m_i}) \to c'(u,v)\\ & & \mid \cdots\\ & & \mid p_1(x_n,\dots,x_{m_n}) \to 0\\ d_i(\overline{x}) & = & d'_i(\overline{x})\\ \text{update}(k,u) & = & \text{update}'(k,u)\\ \text{apply}(k,v) & = & \text{apply}'(k,v) \end{array}\right.$$

It can look like the non-heap-free expression e_0 in both **match**-rules is being used in a way which will destroy the linearity of the type system. It can easily be argued that this is not the case. The contents of e_0 is never used in any match-branch of the sub-expression $a(\overline{x})$. Therefore we will not manipulate any heap-allocated values and this leads to the conclusion $a(\overline{x})$ is harmless. We are in other words allowed to use e_0 twice within the same rule since one of the occurrences is guaranteed not to change any values.

There are more examples of infix operations which could define a suitable continuation as above. The reason for not including them here is partly that it would take up too many pages to list them and also that the transformations involved are similar to those already shown.

The proofs that update(k, x) and apply(k, y) meet the conditions in Theorem 1 are not shown. To see how such a proof go look at the case for $C_f\big(e_1 \cdot C'_f(e_2)\big)$ with conditions $K' = \texttt{undef}$ and $i'_K = \texttt{undef}$:

$$\begin{array}{lcl}
\text{apply}(i_K, v) & = & i_K \cdot v \\
& = & 1 \cdot v \\
& = & v \\
\\
\text{apply}(\text{update}(k, u), v) & = & \text{update}(k, u) \cdot v \\
& = & e_1(u) \cdot k \cdot v \\
& = & k \cdot (e_1(u) \cdot v) \\
& = & \text{apply}(k, e_1(u) \cdot v)
\end{array}$$

For each Borel program it is now possible to find the set of computable tail-recursive first-order types by recursively applying the rules above. The rules used will define a piece of iterative c-code which by Theorem 1 will compute the meaning of the type.

2.3.1 Tail recursion example

The expression $e = \textbf{if } 0 \textbf{ then } 1 \textbf{ else } 2 \cdot f(x-1)$ is computable tail recursive wrt. $f : (N) \rightarrow N$. Applying the rules above lead to:

$$\begin{array}{lcl}
K & = & N \\
i_K & = & 1 \\
a(x) & = & 1 \textbf{ and } 1 \\
b(x) & = & \textbf{if } \neg(1 \textbf{ and } 1) \textbf{ then } 2 \textbf{ else } 1 \\
c(u, v) & = & 2 \cdot v \\
d_i(x) & = & x - 1 \\
\text{update}(k, u) & = & 2 \cdot k \\
\text{apply}(k, v) & = & k \cdot v
\end{array}$$

and it can easily be checked that the sub-expression will generate a piece of iterative c-code which computes the meaning of e.

2.4 Borel examples

The following are examples of algorithms which can be expressed by the Borel language:

Tree mirror:

$$
\begin{aligned}
&\mathrm{Tree}(\texttt{leaf}.(U),\ \texttt{node}.(N, \mathrm{Tree}, \mathrm{Tree}))\\
&\texttt{leaf} : (U) \to \mathrm{Tree}\\
&\texttt{node} : (N, \mathrm{Tree}, \diamond, \mathrm{Tree}, \diamond) \to \mathrm{Tree}\\
&\texttt{mirror} : (\mathrm{Tree}) \to \mathrm{Tree}\\
&e_{\texttt{mirror}}(t) =\\
&\quad \textbf{match } t \textbf{ with}\\
&\quad \mid \texttt{leaf}(u) \to \texttt{leaf}(u)\\
&\quad \mid \texttt{node}(i, t1, r1, t2, r2) \to\\
&\qquad \texttt{node}(i, \texttt{mirror}(t2), r1, \texttt{mirror}(t1), r2)
\end{aligned}
$$

Breadth-first ordering:

$$
\begin{aligned}
&\mathrm{Tree}(\texttt{leaf}.(U),\ \texttt{node}.(N, \mathrm{Tree}, \mathrm{Tree}))\\
&\texttt{leaf} : (U) \to \mathrm{Tree}\\
&\texttt{node} : (N, \mathrm{Tree}, \diamond, \mathrm{Tree}, \diamond) \to \mathrm{Tree}\\
&\texttt{breadth} : (L(\mathrm{Tree})) \to L(N)\\
&e_{\texttt{breadth}}(l) =\\
&\quad \textbf{match } l \textbf{ with}\\
&\quad \mid \mathbf{nil}_{\mathrm{Tree}} \to \mathbf{nil}_N\\
&\quad \mid \mathbf{cons}(h, t, d) \to\\
&\qquad \textbf{match } t \textbf{ with}\\
&\qquad \mid \texttt{leaf}() \to \mathbf{nil}_N\\
&\qquad \mid \texttt{node}(i, t1, r1, t2, r2) \to\\
&\qquad\quad \mathbf{cons}(i, \texttt{breadth}(\mathbf{snoc}(t2, \mathbf{snoc}(t1, t, r1), r2), d)
\end{aligned}
$$

3. Compiler Design

This chapter presents the details required to turn the Borel metalanguage into a real programming language. The metalanguage was defined mathematically and this is useful when treating subjects such as typing rules, code templates and tail optimisation. A Borel program has until now been considered to be a tuple of signatures, constructors and expressions, but no detail has been given about the textual representation of such a tuple. The following will define the concrete syntax and show a representation of all zero- and first-order types. The goal of the chapter is to fill in missing specifications and provide the design of a compiler for the Borel language.

3.1 Structure

The Borel compiler can be characterised as a program that accepts a Borel program text as input and produces a c-program text as output, while preserving the meaning of the input. The translation is carried out by a series of phases each representing a logically distinct task. The structure of the compiler is shown by a diagram in figure 3.1 and it follows closely the conceptual structures presented in [1] and [5].

The main reason for breaking the compilation process into phases is that it becomes more explicit what the tasks are, thereby allowing for a correct covering of each task. Another reason is that changes to the language will typically effect only one phase and this allows adding new features without redesigning the entire compiler. Finally the separation of phases makes it possible to use program-generating tools (fx. Ocamllex and Ocamlyacc) which given a high level specification of a task will generate a program to solve it.

The first four phases of the compiler form the *front-end*. It is the part concerned with reading the input, checking for syntactical correctness, building abstract syntax tree, performing semantical analysis and building intermediate representation. The lexical analyser and syntactical analyser reads and checks the input for syntactical correctness, while building an abstract syntax tree. The semantical analyser and resource analyser will validate the abstract syntax tree and build a static environment. The last two phases form the *back-end*, which is the part concerned with synthesis of optimised c-code from the static environment.

In figure 3.1 is also shown an error handler. This is not a compiler phase but a set of support tools that interacts with the front-end and reports errors. The success

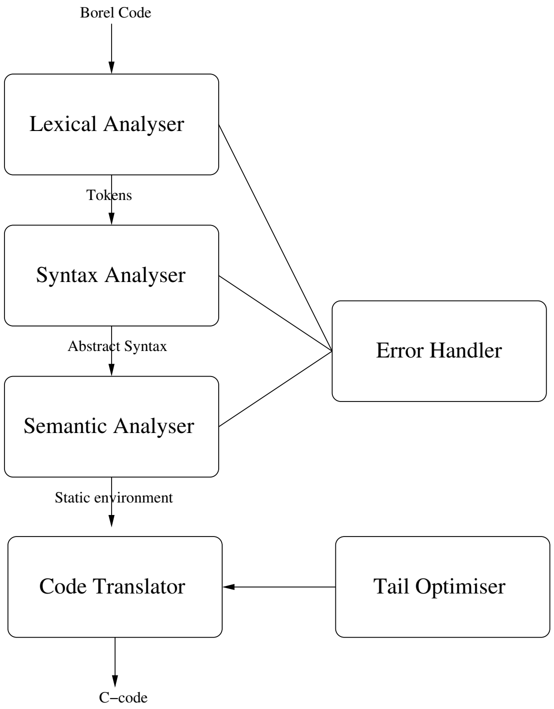


Figure 3.1: Compiler structure

of a compiler (or even a programming language) is often decided by the quality of the error handler and that is the reason for emphasising it in the diagram.

3.2 Lexical analysis

Lexical analysis is the phase that group characters of the input text into recognised words (called tokens). The analyser will accept a stream of characters as input and generate a stream of tokens as output. The tokens of the Borel language is described in the following together with information about their lexical values. The relation between tokens and constructs of the metalanguage should be fairly obvious.

All the Borel tokens will be associated to a lexical value indicating its position in the input stream. This value is used by the error handler and will be described in section 4.1. Some tokens have additional lexical values which will be described below.

The reserved words of the Borel language are:

prog, **end**, **unit**, **int**, <>, **list**, **datatype**, **of**, **not**, **and**, **or**,
nil, **cons**, **snoc**, **concat**, **let**, **in**, **if**, **then**, **else**, **match**, **with**

The identifier token is an arbitrarily long sequence of lowercase letters, uppercase letters, digits, underscore and dot starting with a lowercase letter. An identifier can not be one of the reserved words listed above. The additional lexical value of an identifier is the sequence of characters that formed it. A regular expression for identifiers is:

$$[a-z][a-zA-Z0-9\_.]^*$$

A constructor identifier is like the normal identifier except that it must start with an uppercase letter. The regular expression is:

$$[A-Z][a-zA-Z0-9\_.]^*$$

The number token is a sequence of digits. There is an additional lexical value which is the integer represented by the digits. A number is described by:

$$[0-9]^+$$

The punctuation tokens are representing the symbols:

, @ ( ) | -> $= \quad < \quad > \quad + \quad - \quad * \quad /$

None of them carry an additional lexical value.

The whitespace token is representing white space, newline and tab. A whitespace will not have any semantical meaning so the token is ignored and will not occur in the output stream. In order to allow free formatting of the input stream the whitespace token is allowed anywhere in the input stream.

The comment token is a sequence of ASCII characters starting with (∗ and ending with ∗). For simplicity reasons there can not be nested comments in the input stream. Since a comment should not have any effect on the generated code the token is ignored and will not occur in the output stream. The comment has regular expression:

$$(*\ [\text{any ascii character}]^*\ *)$$

The verbatim token is a sequence of ASCII characters starting with (# and ending with #). As for comments it is not possible to have nested verbatims.

The additional lexical value of a verbatim token is the sequence of characters occurring between the (# and #) marker. The verbatim is thought to represent syntactically correct c-code which can be passed all the way through the compiler and copied directly into the c-program generated by the back-end. The value of a verbatim token is never checked and should therefore be used with care. A regular expression for verbatim is:

$$(\# \text{ [any ascii character]}^* \#)$$

Any other sequence of characters in the input stream will be rejected so that the above defines all the legal Borel tokens.

3.3 Syntax analysis

The syntax analysis is the phase where tokens are matched with a grammar to check that the input is syntactically correct. The analyser receives a stream of tokens, matches these with the grammar and builds up an abstract syntax tree. The concrete syntax of a Borel program is represented by the BNF grammar in figure 3.2. The terminals or the grammar are the tokens listed in section 3.2.

The concrete syntax is designed to follow the style of functional programming languages such as SML and Ocaml (see [2] and [8]). This means that the signature of a function is given together with its name and defining expression. Since the Borel language is still experimental there is a bit of extra syntax involved with defining functions. It is necessary to give return and parameter types for the function which will allow for easy type checking. The extra syntax could be avoided by introducing a type inference mechanism that automatically finds return and parameter types. Since type inference is outside the scope of this project it was decided that well-chosen extra syntax is the best alternative.

The grammar productions for expressions defines a syntax very close to that of the metalanguage. It can be noted that the concrete syntax allows any expression to be enclosed in braces (). This is useful for structuring the expressions and for overruling precedence of operators. Also note that **nil** is annotated by a type which will simplify the type checking. Again the extra annotation could be avoided by introducing a type inference mechanism.

It should be noted that the syntax for datatype allow to declare a constructor with only a name and no additional types. A type declared in that way can be thought of as holding the unit type. Since the unit type has only one value there is no idea in forcing the programmer to write it explicitly.

The order of datatype and function definitions is not restricted by the grammar. This will allow the programmer to organise the code in any way he or she may

$$
\begin{array}{lcl}
\text{Program} & ::= & \{\text{verbatim}\}_{opt}\ \textbf{prog}\ \{\text{Datatype} \mid \text{Definition}\}_{0+}\ \textbf{end} \\
\text{Datatype} & ::= & \textbf{datatype}\ \text{identifier} = \text{Constructor}\ \{\mid \text{Constructor}\}_{0+} \\
\text{Constructor} & ::= & \{\text{constructor}\ \{\textbf{of}\ \text{TypeProd}\}_{opt}\}_{1+} \\
\text{TypeProd} & ::= & \text{Type}\ \{*\text{Type}\}_{1+} \\
\text{Type} & ::= & \textbf{unit} \mid \textbf{int} \mid\ <>\ \mid \textbf{list}(\text{Type}) \mid \text{identifier} \\
\text{Definition} & ::= & \textbf{def}\ \text{Type identifier}(\{\text{ParList}\}_{opt}) = \text{Expression} \\
\text{ParList} & ::= & \text{Parameter}\ \{,\ \text{Parameter}\}_{0+} \\
\text{Parameter} & ::= & \text{Type Variable} \\
\text{Variable} & ::= & \text{identifier} \\
\text{Expression} & ::= & () \\
 & \mid & \text{number} \\
 & \mid & \text{Variable} \\
 & \mid & (\text{Expression}) \\
 & \mid & \text{PrefixOp Expression} \\
 & \mid & \text{Expression InfixOp Expression} \\
 & \mid & \text{identifier}(\text{ExprList}) \\
 & \mid & \text{constructor}\ \{(\text{CArgList})\}_{opt} \\
 & \mid & \textbf{nil}(\text{Type}) \\
 & \mid & \textbf{cons}(\text{Expression},\ \text{Expression @ Expression}) \\
 & \mid & \textbf{snoc}(\text{Expression},\ \text{Expression @ Expression}) \\
 & \mid & \textbf{concat}(\text{Expression},\ \text{Expression}) \\
 & \mid & \textbf{let}\ \text{Variable} = \text{Expression}\ \textbf{in}\ \text{Expression} \\
 & \mid & \textbf{if}\ \text{Expression}\ \textbf{then}\ \text{Expression}\ \textbf{else}\ \text{Expression} \\
 & \mid & \textbf{match}\ \text{Expression}\ \textbf{with}\ \text{PatLst} \\
 & \mid & \textbf{match}\ \text{Expression}\ \textbf{with}\ \text{Pattern}\ \{'|'\ \text{Pattern}\}_{0+} \\
\text{PrefixOp} & ::= & \textbf{not} \\
\text{InfixOp} & ::= & + \mid - \mid * \mid / \mid = \mid < \mid > \mid \textbf{and} \mid \textbf{or} \\
\text{ExprList} & ::= & \text{Expression}\ \{,\ \text{Expression}\}_{0+} \\
\text{CArgList} & ::= & \text{CArg}\ \{,\ \text{CArg}\}_{0+} \\
\text{CArg} & ::= & \text{Expression}\ \{@\ \text{Expression}\}_{opt} \\
\text{PatLst} & ::= & \textbf{nil}(\text{Type})\ {-}{>}\ \text{Expression} \\
 & & '|'\ \textbf{cons}(\text{Variable}, \text{Variable@Variable})\ {-}{>}\ \text{Expression} \\
\text{Pattern} & ::= & \text{constructor}\ \{(\text{PatVarList})\}_{opt}\ {-}{>}\ \text{Expression} \\
\text{PatVarList} & ::= & \text{PatVariable}\ \{,\ \text{PatVariable}\}_{0+} \\
\text{PatVariable} & ::= & \text{Variable}\ \{@\ \text{Variable}\}_{opt}
\end{array}
$$

Figure 3.2: Concrete syntax for the Borel language.

wish. It should be noted that a later phase of the compiler ensures that the a datatype is defined before its use.

The concrete syntax given is not the complete definition of the Borel language. The grammar is full of ambiguities which will have to be resolved before it can be used by a syntax analyser. The most serious ambiguity is caused by the unary and binary operators. It is nowhere defined how one operator has precedence over another and it is not defined how associativity should work. Since this can lead to semantical ambiguities in later phases it is important to decide on precedence order and associativity rules. The decreasing precedence order of Borel operators is:

$$\mathbf{not}, *, /, +, -, <, >, =, \mathbf{and}, \mathbf{or}$$

and they are all left associative. This means that the expression $1 + 2 * 3$ is understood as $1 + (2 * 3)$.

As mentioned above the syntax analyser will build up an abstract syntax tree (AST). The tree is an internal representation of a Borel program so it should be simple and easy to maintain. There is no need to represent anything else but the pure structure in the AST, since possible ambiguities are resolved before building it. The AST used in the Borel compiler is defined by the grammar in figure 3.3. The terminals of the grammar should not need further explanation except that **verbatim**, **identifier** and **variable** are associated with a string constant and that **number** is associated with an integer constant.

3.4 Semantic analysis

Semantic analysis is the phase where the abstract syntax tree (AST) is validated. The main concern of semantic analysis is to type check all expressions using the type rules from section 2.1. The semantic analyser accepts an AST as input and builds up a static environment which is an intermediate representation of the datatypes and functions in the program.

The static environment is a compound structure with the following five components:

- A datatype environment which is a list of pairs. The pairs will have an identifier field containing the name of a datatype and a type field. The environment can be thought of as a mapping from datatype names to types.
- A constructor environment which is a list of pairs. Each pair has an identifier field containing the name of a constructor and a field which hold the signature of the constructor. The environment can be thought of as a mapping from constructor names to constructor signatures.

Program	::=	Verbatim Datatypes Definitions
Verbatim	::=	**verbatim** | ϵ
Datatypes	::=	Datatype Datatypes | ϵ
Datatype	::=	**identifier** DtSignature
DtSignature	::=	Signature DtSignature | ϵ
Signature	::=	**identifier** Types
Types	::=	Type Types | ϵ
Type	::=	**int** | **diamond** | **list** Type | **datatype identifier**
Definitions	::=	Definition Definitions | ϵ
Definition	::=	**identifier** FnSignature Expression
FnSignature	::=	Parameters Type
Parameters	::=	Parameter Parameters | ϵ
Parameter	::=	Type **variable**
Expression	::=	**unit**
	|	**number**
	|	**variable**
	|	**operation** Operator Expressions
	|	**constructor identifier** Expressions
	|	**nil** Type
	|	**cons** Expressions
	|	**snoc** Expressions
	|	**concat** Expressions
	|	**let variable** Expressions
	|	**if** Expressions
	|	**match** Expression Patterns
Expressions	::=	Expression Expressions | ϵ
Operator	::=	**plus** | **minus** | **times** | **divide** | **equals** |
		lt | **gt** | **and** | **or** | **not** | **fun identifier**
Patterns	::=	Pattern Patterns | ϵ
Pattern	::=	**identifier** Variables Expression
Variables	::=	**variable** Variables

Figure 3.3: Abstract Syntax Tree for the Borel language.

- A signature environment is a list of triples. The first element of each pair is an identifier containing the name of a function. The second element will hold a signature of the function. The third element is a typed expression. The environment can be thought of as a mapping from function names to pairs of a function signature and a defining expression.
- A typing context which is a list of pairs. The pairs will have a variable field containing the name of a variable and a type field. The typing context can be seen as a mapping from variable names to types. Nested scope will be handled by a first-in-last-out mechanism on pairs with same variable name.
- A list-type environment which is a list of types. The list-type environment is thought to hold the type of all lists occurring in an AST.

The semantic analyser will recurse through the AST while type checking the nodes. The type checking of **datatype**-nodes is done by updating the static environment with a new datatype and with one or more new constructors. The type checking of **definition**-nodes will first update the static environment with a new signature and then type check the defining expression for the function. The **definition**, **let** and **match** nodes will bind new variables and this is handled by updating the typing context with names and types of the variables. Whenever a **list**, **nil**, **cons** or **snoc** node has been type checked then the static environment is updated with a new type of list.

When adding a constructor to the static environment it is done in a slightly different way than described by the type rules. Remember that a constructor was defined to by a first-order type with signature:

$$d_i : (T_{i,1}, R_1, \ldots, T_{i,m_i}, R_{m_i}) \to D(S_1 \ldots, S_n)$$

where R_i is unit type U if T_i is heap-free and diamond type $\diamond$ otherwise. Since unit U is a singular type (i.e. it has only one value) it is redundant to keep track of it in the signature. In other words, the programmer should be relieved from passing around the unit value since it will not carry any information. It is easy to see that the type checker should add a modified signature with all R_i's of unit type removed. This will change the number of arguments to constructors, but since it simplifies the code it is a very reasonable change.

An important aspect of the semantic analysis is to check that functions and operations uses their arguments at most once. In other words it must be ensured that the program meet the linearity constraint introduced be the type system. When recursing the AST it is easy to build a list of free variables occurring in an expression. This is simply done by merging the free variable occurring in sub-expressions and then remove the variables bound by the expression. The linearity constraint can be met by disallowing a variable to occur twice in any list of free variables. For **if** and **match** nodes it is allowed for a variable to occur once in each branch. The free variable method can easily be modified to take care of these special cases.

For some nodes there are simple extra requirements that has to be checked. An example of this is the match-node which should be complete in the sense that the number of cases is same as the number of constructors for the datatype which is matched.

3.5 Code translation

The code translation is the first phase of the back-end. It will take a static environment as input and generate an abstract code tree (ACT) which is an intermediate representation of c-code. When translation is complete (i.e. all information has been substracted from the static environment) the ACT will be printed into an output string of c-code.

The ACT should have the same properties as the AST described in section 3.3 except that it represents c-syntax and not Borel-syntax. The ACT used by the Borel compiler is shown by the grammar in figure 3.4.

The translation process will make use of the code templates as they are defined in section 2.2. It is straightforward to do since all the constructs used by code templates can be translated into a node in the ACT.

There is another important design decision which is concerned with ensuring uniqueness of variable names in the generated program. First it should be noted that the variable names need only be unique within a function. The uniqueness is achieved by prefixing all the variable names occurring in Borel syntax with the string `var_` and by postfixing all variables within a function by a unique number (i.e. a string of letters $0-9$). A variable `x` occurring in Borel syntax could get the name `var_x7`. A variable `v` introduced by one of the code templates could get the name `v13`.

3.6 Tail optimising

The tail optimising phase is a part of the back-end. The optimiser will repeatedly receive AST-nodes from the code translator, each node representing a function, and it will return tuples of expressions. If a function is not tail recursive then the optimiser outputs an empty tuple. Otherwise it outputs a tuple of expressions that define a piece of iterative c-code.

Program	::=	ProgBit Program | ϵ
ProgBit	::=	**include** | **comment** | TypeDef | GlobalVar | FuncDef
TypeDef	::=	Type **identifier**
Type	::=	**void** | **int** | **named identifier** | **pointer** |
		enum Identifiers | **union** Fields | **struct** Fields
Identifiers	::=	**identifier** Identifiers | ϵ
Fields	::=	Field Fields | ϵ
Field	::=	Type **identifier**
GlobalVar	::=	Type **variable**
FuncDef	::=	Type **identifier** Parameters Statements
Parameters	::=	Parameter Parameters | ϵ
Parameter	::=	Type **variable**
Statements	::=	Statement Statements | ϵ
Statement	::=	**nop**
	|	**break**
	|	**stm_comment**
	|	**block** Statements
	|	**decl** Type **variable**
	|	**expr** Expression
	|	**assign** Expression Expression
	|	**cond** Expression Statement Statment
	|	**switch** Expression Cases
	|	**while** Expression Statment
	|	**return** Expression
Expression	::=	**const**
	|	**null**
	|	**var variable**
	|	**cast** Expression
	|	**infix** InfixOp Expressions
	|	**prefix** PrefixOp Expression
	|	**func_call identifier** Expressions
	|	**arrow** Expressions
	|	**dot** Expressions
	|	**malloc** Expression
Expressions	::=	Expression Expressions | ϵ
InfixOp	::=	**plus** | **minus** | **times** | **divide** | **equals** | **lt** | **gt** | **and** | **or**
PrefixOp	::=	**not**
Cases	::=	Case Cases | ϵ
Case	::=	Expression Statements

Figure 3.4: Abstract Code Tree for c-syntax.

The rules in section 2.3 define a tuple of expressions for every tail recursive function. This provide a straightforward design of the tail optimiser.

3.7 Examples

The language has now been designed and it should be easy to understand and write Borel programs. The following shows the two examples from section 2.4 written in concrete Borel syntax:

Tree mirror:

```
prog

  datatype tree = Leaf | Node of int * tree * tree

  def tree mirror(tree t) =
    match t with
      Leaf -> Leaf
    | Node(i, t1@d1, t2@d2) ->
          Node(i, mirror(t2)@d1, mirror(t1)@d2)
end
```

Breadth-first ordering:

```
prog

  datatype tree = Leaf | Node of int * tree * tree

  def list(int) breadth(list(tree) l) =
    match l with
      nil(tree) -> nil(int)
    | cons(h, t@d) ->
        match t with
          Leaf -> nil(int)
        | Node(i, t1@r1, t2@r2) ->
            cons(i, snoc(t2, snoc(t1, t@r1)@r2)@d)

end
```

4. Compiler Implementation

The implementation language for the Borel compiler is Ocaml. The reason is mainly that Ocaml is a fast and reliable functional language with recursive datatypes and pattern matching. Also there is a nice standard library implementing useful functionality needed within the compiler. For more information on Ocaml see [8].

Another reason for choosing Ocaml as implementation language was that parts of existing code could be used. Before starting this project work had been done[1] on implementing a syntax analyser and a type checker for a linear functional language very much like Borel. The existing code could easily be modified to fit into the Borel compiler.

The compiler is build up using the structure described in section 3.1. The phases are located in separate files and so are the definitions of abstract syntax tree (AST) and abstract code tree (ACT). In order to combine the phases there is a main program file which will control the compilation process and maintain the input and output.

4.1 Error Handler

The error handler is a central part of the compiler. It is designed to report errors found during the analysis phases and thereby help the programmer find and correct errors within a program. Error handling in the Borel compiler is done by a set of utilities developed by Benjamin Pierce at University of Indiana. The utilities are not documented elsewhere so the following is a quick overview of the functionality.

The main idea is that every AST-node should be associated with information about its position in the input string (i.e. filename, line and character numbers). Whenever a syntax error or type error is found this makes it possible to refer directly to its position. The information is defined by the following type:

```
type info = FI of string * int * int | UNKNOWN
```

Together with the definition of the info type is a function to print its contents. The format for printing is [filename : line : character].

In some situations it is useful to carry an additional value together with the the info type. For this there is a polymorfic type definition:

[1] By Martin Hofmann (mxh@dcs.ed.ac.uk) and David Aspinell (da@dcs.ed.ac.uk)

```
type 'a withinfo = {i: info; v: 'a}
```

which will hold an info type and a value of any other type. The additional value can be used to generate more precise error reports than those generated by an `info` value alone.

The utility functions provided by the error handler will take one argument of type info and one of type string. The functionality is to print the info value to standard error followed by a message produced by prefixing the string value with a suitable description. The most important utility function is `errAt` which on input FI("text.brl", 7, 19) and "Variable foo unbound" generates the following message:

```
[test.brl:7:19] Error: Variable foo unbound.
```

The utility functions will raise an exception which will usually terminate the compilation process.

4.2 Lexical Analyser

In the Ocaml library is implemented an automated scanner generator called ocamllex. Its main functionality is to generate a lexical analyser from a specification file. For detailed information on the use of ocamllex see [8]. The Ocaml scanner generator can be seen as a twin brother of the well known GNU Flex tool. The main difference being that it uses and generates Ocaml-code instead of c-code.

The syntax analyser generated by ocamllex is a piece of Ocaml-code simulating a finite automata for recognising strings. The specification file for ocamllex is build by a list of regular expressions and actions. Every time the finite automata matches a string with a regular expression it will execute the corresponding action part. The syntax analyser for the Borel language will recognise the regular expressions given in section 3.2 and output the tokens they define.

The lexing state used by the compiler is the one defined by ocamllex (last substring matched, position of last substring, etc.) together with information about filename and line number. The filename is set globally when invoking the lexical analyser and the line number is updated whenever the lexical analyser matches a newline character.

In order to pass information about line number and character position to the parser there is a utility function called `infoof` defined in the preamble of the specification file. The functionality is to extract the needed values from the current lexing state and produce a value of type `info`. Another utility function is `text` which will extract the last substring that was matched by the the finite automata.

As an example of the format of the specification file is the line that matches numbers:

```
[0-9]+ {INTEGER{i=(info lexbuf); v=int_of_string(text lexbuf)}}
```

The regular expression is the one earlier defined and the action part will build an integer-token. As it can be seen the integer-token is associated with a value of type withinfo holding an info and an integer value.

If the lexical analyser fails to match a substring with any regular expression then it will report an error by calling the `errAt` function. The specification of this is:

```
_ {Support.Error.errAt (info lexbuf) "Illegal character"}
```

The regular expression will match any character. Since the rule is placed after every other rule it is only invoked if all other regular expressions have failed to match.

4.3 Syntax analyser

The implementation of a syntax analyser for the Borel language will make use of ocamlyacc. This is an automated parser generator from the Ocaml library which will generate a syntax analyser from an annotated grammar. Detailed information about ocamlyacc can be found in [8]. The Ocaml parser generator can be seen as a twin sister to the GNU Bison tool the only difference being that ocamlyacc uses and generates Ocaml-code instead of c-code.

Given an annotated LALR(1)[2] grammar as input ocamlyacc will generate a piece of Ocaml-code that simulates a deterministic push-down automata. The push-down automata will recognise the language specified by the input grammar thereby providing a bottom-up traversal of the grammar tree. The bottom-up traversal will make use of the annotations to build up an AST.

Productions of the grammar used for generating the Borel syntax analyser is very much like the ones shown in section 3.3. Only a few changes were made to remove ambiguities. A LALR(1) grammar can usually be obtained by splitting productions into sub-productions and this his is also the case for the Borel grammar. The operator precedence and associativity is specified by the following lines in the preamble of the grammar:

[2]LALR is an acronym for Look-Ahead Left Recursive and the one refers to number of look-ahead symbols.

```
%left NOT
%left TIMES DIVIDE
%left PLUS MINUS
%left LESS GREATER
%left EQUALS
%left AND OR
```

where precedence is given by the order of lines and associativity is defined by the keyword prefixing each line.

The annotations of the grammar will build up an AST. Each annotation defines an action which will be executed when a production matches a string. As an example of the format of the actions is the grammar production for datatype:

```
datatype: DATATYPE IDENT EQUALS tysig
    {LProgDatatype($1, $2.v, $4)};
```

The action part will build an AST-node named `LProgDatatype` by calling the constructor function of that name.

A few remarks should be made on the Ocaml-type representing the AST for Borel syntax. The grammar given in section 3.3 is easily turned into an equivalent Ocaml-type. Each production in the grammar will be represented by a type constructor. The values of each type constructor are as specified by the corresponding production. Since each type constructor will generate a function (like `LProgDatatype`) it is easy to build the AST-nodes.

In order to debug the syntax analyser and later phases of the compiler it is useful to have a pretty printer which will print the AST. The Ocaml library provides a good formatting module so this is simple to implement. A pattern-match on the AST-nodes and some simple printing commands in each case is enough to print and indent the Borel code. This shows the strength of Ocaml as implementation language.

4.4 Type checker

The type checker will recursively traverse the AST while building a static environment as described in section 3.4. The structure of the type checker is rather complicated so it will need a bit of explanation.

The algorithm used is a typical two-pass strategy. The first traversal of the AST will build up a static environment of datatype names, constructor signatures, and function signatures. It is required that a datatype must be declared before its use and this has the effect of permitting mutually recursive datatypes. The second traversal of the AST will perform the actual type checking of expressions. It is done recursively by first type checking sub-expressions, then by checking that the structure of the node is as expected (i.e. the correct number of sub-expressions, expected types of sub-expressions, etc), and finally by returning an annotated AST-node with type information. The static environment will be updated whenever the defining expression for a function has been checked. The two pass strategy will allow functions to be recursive without adding extra syntax.

When the static environment has been build a last concern is to find all the types of list occurring in a program. This is done by a simple traversal of the constructor signatures and the annotated expressions. The pass is postponed till after type checking because the annotations are needed.

If an error is found at any point during the type checking then it will be reported by using the `errAt` function from the support library. The error messages are reasonably informative but it must be admitted that some could be improved. The error handler is really useful in this situation since it is possible to report exactly what has caused the error together with information about position in the input file.

The concrete implementation of static environment is as a record containing the five environments for datatypes, constructors, lists, functions and variables. Together with the definition are update and lookup functions which will maintain the information in each environment. It is the obvious representation and should not need further explanation.

One of the most important concerns when type checking Borel code is performing a disjointness test on sets of variables. This is the main feature of the linear type system for the language. The following will take a closer look at a concrete implementation.

The disjointness condition can be met by merging lists of free variables as described in section 3.4. The type checker will build the list of free variables occurring in each expression and pass the information backwards through the recursion. Typically it is the case that a non-heap-free variable must occur only once in an expression but there are a few exceptions. The full description of legal an illegal merges is defined by the typing rules in section 2.1. Whenever the type checker finds an illegal variable it will report it by the following function call:

```
errAt fi ("TypeCheckError. Repeated variable usage for "
           ^ x ^
          "violates linearity restriction.")
```

If there is a most important error message in the compiler then it must be this one. It captures the essence of linear typing which provides the basis for our in-place manipulations.

4.5 Code Translator

The code translator is implemented in a rather straightforward way. The input is a static environment computed by the type checker and the algorithm will be to first generate all the code for datatypes in the datatype environment, then all the code for lists occurring in the list environment, and finally put this together with the code for functions. Splitting the code translation into such phases makes the output more clean. The overall structure of the code-file can be seen in figure 4.1.

The code translation for datatypes is done by generating a piece of abstract c-code for each definition in the datatype environment and then return a list of such pieces. The method can be thought of as a kind of stamp which given the name and signature of a datatype will output an appropriate piece of c-code. Each datatype is translated using the code templates from section 2.2.

Code translation of lists is done in a way much similar to the translation of datatypes. For each kind in the list environment there will be generated a piece of abstract syntax corresponding to the code template for list. When all the pieces have been generated they will form a list of abstract syntax which is returned. The method is again like a stamp which given a type will output abstract c-code.

The most interesting part of the code translator is concerning functions. For each function in the signature environment there will be generated abstract c-code corresponding to the code templates in section 2.2. The algorithm used is a bit tricky so it might need a bit of explanation. First there will be generated a function header from the signature found in the signature environment. This is just the function name together with the parameter and return types. Then the c-code for the function body is generated by recursively applying code templates to abstract Borel syntax. The code templates for expressions are defined to assign its result to a c-variable and this leads to introducing an extra name parameter in the recursion. The parameter is the name of a variable and it makes it possible to recursively generate assignments. In order to obtain uniqueness of variables in the generated code it was decided to postfix with a unique number (see section 3.5). This leads to introducing a number parameter in the recursion which will count the recursion depth. By knowing the recursion depth it is very easy to generate unique numbers. Another way of obtaining the unique numbers would be a global reference holding a counter but this is not a good functional programming style so the recursion method was preferred.

In order to make testing a bit easier it was decided to automatically generate a function which allows the creation of $\diamond$ values. The code for this can be seen in figure 4.1. By defining a maximal type as the union of all types occurring in the program it is ensured that each $\diamond$ value is big enough to hold the largest heap-allocated type. The `new()` function should of course be used with care (and

/ ∗ File: filename.c /∗
/ ∗ Automatically generated by Borel compiler ∗ /

#include < stdlib.h >

typedef int unit_t;
typedef int int_t;
typedef void *dia_t;

/ ∗ Datatypes section ∗ /

$\llbracket \mathrm{Datatype}_1 \rrbracket^c$
...
$\llbracket \mathrm{Datatype}_n \rrbracket^c$

/ ∗ Lists section ∗ /

$\llbracket \mathrm{List}_1 \rrbracket^c$
...
$\llbracket \mathrm{List}_m \rrbracket^c$

/ ∗ Diamond section ∗ /

typedef union { All types occuring } max_type;
dia_t new() {
    return malloc(sizeof(max_type))
}

/ ∗ Functions section ∗ /

$\llbracket \mathrm{Function}_1 \rrbracket^c$
...
$\llbracket \mathrm{Function}_m \rrbracket^c$

Figure 4.1: Overall structure of generated c-code.

preferably only for testing) since it overrules the malloc-freeness of the c-code.

4.6 Code Optimiser

As described in section 3.6 the code optimiser will be called once for every function in a program. The returned value is either a tuple of sub-expressions for generating iterative c-code or an empty tuple indicating that the function is not tail recursive.

The algorithm for computing the tuple of sub-expressions is very simple. By recursively trying to match with optimisation cases the tuple will be build by the corresponding transformation. Both optimisation cases and transformations was defined in section 2.3. If the code optimiser is able to match the whole expression then the function is tail recursive and otherwise it is not.

4.7 Example

As an example of the c-code generated by the compiler is the tree mirror function:

```
/* File: Mirror.c */
/* Automatically generated by Borel compiler. */

#include <stdlib.h>
typedef void *dia_t;

/*  ---------- DATATYPES SECTION ---------- */

typedef enum {TREE_LEAF, TREE_NODE} tree_kind_t;

typedef struct tree_node {
   tree_kind_t kind;
   union {
      int tree_leaf;
      struct {
         int node_0;
         struct tree_t *node_1;
         struct tree_t *node_2;
      } tree_node;
   } val;
} tree_t;

typedef struct tree_denode {
   tree_kind_t kind;
   union {
      int tree_leaf;
```

```
      struct {
         dia_t d1;
         dia_t d2;
         int node_0;
         tree_t node_1;
         tree_t node_2;
      } tree_node;
   } val;
} tree_destr_t;

tree_t cons_leaf(int tree_leaf) {
   tree_t res;
   res.kind=TREE_LEAF;
   res.val.tree_leaf=tree_leaf;
   return res;
}

tree_t cons_node(dia_t d1, dia_t d2, int node_0,
                 tree_t node_1, tree_t node_2) {
   tree_t res;
   res.kind=TREE_NODE;
   *(tree_t *)d1=node_1;
   *(tree_t *)d2=node_2;
   res.val.tree_node.node_0=node_0;
   res.val.tree_node.node_1=(struct tree_t *)d1;
   res.val.tree_node.node_2=(struct tree_t *)d2;
   return res;
}

tree_destr_t tree_destr(tree_t tree) {
   tree_destr_t res;
   switch (tree.kind) {
      case TREE_LEAF:
         res.kind=TREE_LEAF;
         res.val.tree_leaf=tree.val.tree_leaf;
         break;
      case TREE_NODE:
         res.kind=TREE_NODE;
         res.val.tree_node.node_0=tree.val.tree_node.node_0;
         res.val.tree_node.d1=
                (dia_t )tree.val.tree_node.node_1;
         res.val.tree_node.d2=
                (dia_t )tree.val.tree_node.node_2;
         res.val.tree_node.node_1=
                *(tree_t *)tree.val.tree_node.node_1;
         res.val.tree_node.node_2=
                *(tree_t *)tree.val.tree_node.node_2;
         break;
   }
   return res;
}
```

```
/*  ------------ LISTS SECTION ------------ */

/*  (No lists used in program)  */

/*  ----------- MALLOC SECTION -----------  */

/*     Use the function new with care!.     */

typedef union {
   tree_t tree;
} max_type;

dia_t new() {
   return malloc(sizeof(max_type));
}

/*  ---------- FUNCTIONS SECTION ---------- */

tree_t mirror(tree_t var_t) {
   tree_t res;
   {
      tree_t ctrl0;
      tree_destr_t destr0;
      ctrl0=var_t;
      destr0=tree_destr(ctrl0);
      switch (destr0.kind) {
         case TREE_LEAF:
            {
               int var_i;
               var_i=destr0.val.tree_leaf;
               {
                  int act1;
                  act1=var_i;
                  res=cons_leaf(act1);
               }
               break;
            }
         case TREE_NODE:
            {
               int var_i;
               tree_t var_t1;
               dia_t var_d1;
               tree_t var_t2;
               dia_t var_d2;
               var_i=destr0.val.tree_node.node_0;
               var_t1=destr0.val.tree_node.node_1;
               var_d1=destr0.val.tree_node.d1;
               var_t2=destr0.val.tree_node.node_2;
               var_d2=destr0.val.tree_node.d2;
               {
```

```
                  dia_t d2;
                  dia_t d3;
                  int act1;
                  tree_t act2;
                  tree_t act3;
                  act1=var_i;
                  d2=var_d1;
                  {
                     tree_t act3;
                     act3=var_t2;
                     act2=mirror(act3);
                  }
                  d3=var_d2;
                  {
                     tree_t act4;
                     act4=var_t1;
                     act3=mirror(act4);
                  }
                  res=cons_node(d2, d3, act1, act2, act3);
               }
               break;
            }
      }
   }
   return res;
}
```

5. Test and Analysis

5.1 Testing functionality

The different phases of the Borel compiler has been tested to ensure correct functionality. This section will briefly describe how the phases have been tested separately and as a whole.

The lexical analyser and syntax analyser was tested by implementing a simple pretty printer to print the contents of the AST. The pretty printer will traverse the AST and the output is a great help when tracing errors in the structure of the tree. A number of small an specialised programs was tried in order to test that every node of the AST is generated correctly.

Checking the functionality of the type checker is a bit subtle because the information about types is hidden in the static environment. The only test performed was designing small programs that systematically would provoke every type error. This is not really enough to ensure correct functionality. A better test would print all types defined in the program to make sure they are represented correctly. Also it is necessary to print all expressions together with their type and then trace by hand to make sure it is correct.

The back-end of the compiler is quite easy to test as an output that can be used to verify the functionality is produced. The strategy used has been to write programs with nested constructs and thereby try to provoke errors. Another test was to try all the small programs from earlier phases and then check by hand that the code is generated correctly.

A number of big and complicated test programs has been used to verify the compiler as a whole. It is of course not possible to test everything in such a way but still it is a good way of provoking overflow errors or other things which could happen when the input and output are large text files.

5.2 Comparing Borel with Ocaml and Clean

This section will be a comparison of the Borel language with the Ocaml and Clean languages. The Borel language is described in this thesis so it should not need further introduction. The Ocaml language (see [8, 10]) is a state-of-the-art ML-style functional language with higher-order types and polymorphism. The native code generated by the Ocaml compiler is well optimised and the heap-space is managed by an efficient unobstructive incremental garbage collector. The Clean language (see [9, 3]) is a state-of-the-art lazy functional language with a higher-order type system, polymorphism and graph rewriting semantics. The native code generated by the Clean compiler is very fast and heap-space is managed by an optimised mark-and-sweep garbage collector.

The two points of comparison will be the amount of heap-space used by programs and the execution time of programs. The three languages are chosen so that they complement each other in an interesting way. A Borel program will have a provable upper limit on the amount of heap-space used and the execution times are quick since the generated code is mostly iterative c-code. The Ocaml programs has an overhead of using a garbage collector for managing heap-space and this might influence the execution time. A Clean program will be executed by lazy-evaluation so that actual computations are postponed until needed. This should give fast and memory efficient programs but of course there is an overhead in representing the intermediate results.

The PC's used for executing programs are 650 MHz Pentium II machines with 128 Mb RAM running Red Hat Linux 6.2 (kernel 2.2.16).

The programs used for the comparison are listed in the following. It should be clear that they represent the same breadth-first algorithm written in different functional syntax. The word `SIZE` is used to indicate where the depth of the tree is decided.

The Borel program is:

```
prog

 datatype tree = Leaf | Node of tree * tree * int

 def tree build(int depth) =
   if (depth=0)
   then Leaf
   else Node(build(depth-1)@new(<>),
             build(depth-1)@new(<>),depth)

 def list(int) breadth(list(tree) l1) =
   match l1 with
     Nil(tree) -> Nil(int)
   | l2::tail@d ->
       match l2 with
         Leaf -> Nil(int)
       | Node(t1@d1,t2@d2,i) ->
           i::breadth(Snoc(t2,Snoc(t1,tail@d2)@d1))@d
```

```
 def int main() =
   let d = SIZE
   in let  t = build(d)
      and n = Nil(tree)
      in let b = breadth( t::n@new(<>) )
         in 1

end
```

The Ocaml program is:

```
type tree = Leaf | Node of tree * tree * int

let rec build depth =
  if (depth=0) then
    Leaf
  else
    Node(build(depth-1),build(depth-1),depth)

let rec snoc (l,t) =
  match l with
    [] -> [t]
  | l1::tail -> l1::snoc(tail,t)

let rec breadth l1 =
  match l1 with
    [] -> []
  | l2::tail ->
       (match l2 with
        Leaf -> []
      | Node(t1,t2,i) ->
           i::breadth(snoc(snoc(tail,t1),t2)))

let main =
  let d = SIZE
  let t = build(d) in
  let b = breadth[t]
  in 1
```

The Clean program is:

```
module breadth

import StdClass
import StdInt

::Tree = Leaf | Node Int Tree Tree
```

```
Build::Int->*Tree
Build 0 = Leaf
Build n = Node n (Build (n-1)) (Build (n-1))

Snoc :: *[*Tree] *Tree -> [*Tree]
Snoc [] x    = [x]
Snoc [a:l] x = [a:Snoc l  x]

Breadth::*[*Tree] -> [Int]
Breadth [Node a l r : tl] = [a:Breadth (Snoc (Snoc tl l) r)]
Breadth [Leaf : tl]       = []
Breadth []                = []

Eval::[Int]->Int
Eval []    = 1
Eval [x:l] = Eval l

Start::Int
Start = Eval(Breadth([Build SIZE]))
```

5.2.1 Comparing execution time

The following will describe how the execution time for breadth-first ordering was compared by an experiment. Also there will be notes on how to interpret the result. The tool used for comparing execution times is the Tcl *time* utility. The utility allow for a simple experiment by simply executing the command:

```
for x in  1  2  3  4  5  6  7  8  9 10 \
         11 12 13 14 15 16 17 18 19 20
do
  echo -n "Running breadth experiment, depth: " $x
  time ./breadth $x
done
```

which will execute the breadth program with increasing depth argument and report the time used for each depth. Running the experiment on the three programs above give the set of data in figure 5.1. The tree depth has been plotted against execution time and the result can be seen in figure 5.2. A bigger print of the plot can be found in appendix A.

As it can be seen the Borel program is much faster than the Ocaml and Clean programs. There is more than one explanation for this. Firstly, the build-in snoc operator is a big advantage for the Borel program. It will not have to traverse

Depth:	Borel prog:	Ocaml prog:	Clean prog:
1	0.000	0.000	0.000
2	0.000	0.000	0.000
3	0.000	0.000	0.000
4	0.000	0.000	0.000
5	0.000	0.010	0.000
6	0.010	0.010	0.000
7	0.010	0.010	0.010
8	0.010	0.010	0.010
9	0.010	0.030	0.020
10	0.010	0.170	0.060
11	0.010	1.290	0.370
12	0.020	9.530	1.820
13	0.040	74.230	10.430
14	0.070	442.500	60.090
15	0.150	2375.950	303.440
16	0.300	12582.470	2188.850
17	0.620	71677.940	7088.870
18	1.260	408328.751	28758.321
19	2.530	-	-
20	4.810	-	-

Figure 5.1: Execution time (in seconds) for breadth-first ordering.

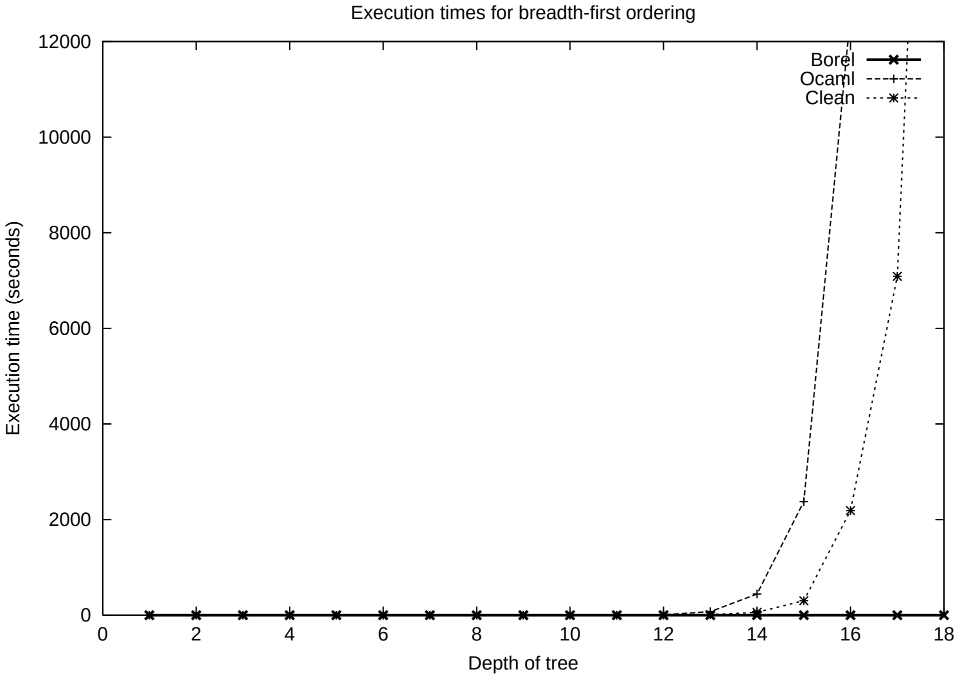


Figure 5.2: Tree depth plotted against execution time.

the list-nodes (since the build-in operator will just update a couple of pointers) like the other programs and expectedly the effect gets more visible as the depth grows. It was tried to make an experiment where the Borel program did not use the build-in operator, but it failed since the recursive version of snoc results in a stack-overflow (for depth 15 and higher). A second reason for the fast execution of the Borel program is that the breadth-algorithm is tail-recursive. This means that the generated c-code is iterative allowing the c-compiler to output very effective native code. The Ocaml and Clean compilers should also optimise for tail recursion, but they seem unable to recognise tail recursion within a list constructor.

The results of the experiment should not be interpreted as a special case where the Borel compiler accidently is able to generate very effective code. The reason behind the good (and maybe surprising) results can be found in many algorithms that rearrange heap-space. Other examples are reversal and flattening of lists. The condition that characterise such *good* algorithms is that they are tail recursive, thereby allowing the Borel compiler to generate iterative code. Both the Ocaml and the Clean compiler are generally known to generate efficient code. The result shown here will not justify saying that this is wrong. What has been exposed is that when it comes to modifications of huge datastructures then it is possible to produce a competitive alternative to the compilers already available. Of course Borel is a more specialised language than Ocaml an Clean and that might be an explanation of its strength in this comparison

5.2.2 Comparing use of heap-space

The following is a description of how amount of heap-space used when breadth-first ordering was compared by an experiment. There are some technical problems involved with measuring exactly how much heap-space a program uses. The difficulties start when a program uses garbage collection since this normally means allocating a big chunk of the heap and then manage this internally.

For Borel programs it is easy to find the amount of heap-space used since there will not normally be any allocations. First of all it is not needed to allocate memory, since the $\diamond$ type allow re-using heap-space, and next it should be avoided because it overrules the basic feature (and advantage) of the type system. The `new()` function (which is a wrapping of malloc) was introduced in the language only to allow building up test-values. When this is said it should be noted that the the Borel version of the breadth-first algorithm actually uses `new()` to build up a tree. This is indeed a test-value and therefore thought to be fine. Measuring the amount of heap-space allocated by the Borel algorithm is done by using the Linux memory profiler (*memprof*). It is designed to count the number of bytes allocated by malloc and therefore it will output the exact amount of heap-space used by the program.

An Ocaml program will make heavy use of a garbage collector and this complicates the task of measuring the exact amount of heap-space used. It is of course possible to give an upper bound by using *memprof* but this could lead to very misleading results if the memory is allocated in big chunks. It has been impossible to find any information on the allocation mechanism used by Ocaml programs and it is surely not possible to predict anything about its behaviour. For this comparison it was decided that the most reasonable measure to use is the amount of heap-space actually allocated by the program while executing the breadth-first algorithm. This is not a precise measure of how much heap-space the program uses but it will say something about the order (i.e. number of chunks) of heap-space required. The program used measuring is again *memprof*.

Measuring the amount of heap-space used by Clean programs is easy. There is a build in option for the programs which will force them to report the use of heap-space. At first this seems to be a reasonable measure to use for a comparison, but there is a small problem. When reading the Clean documentation it becomes clear that the reported usage is "heap-space after garbage collecting". This means that the numbers are lower bounds and not the upper bounds which would be more appropriate. A method for finding upper bounds could be the same as for Ocaml programs, but this leads to another problem. A Clean program will

Depth:	Borel prog:	Ocaml prog:	Clean prog:
	(exact)	(upper bound)	(lower bound)
1	72	427182	0
2	168	427182	0
3	360	427182	0
4	744	427182	0
5	1512	427182	0
6	3048	427182	0
7	6120	428206	0
8	12264	428206	0
9	24552	428206	12032
10	49128	428206	24344
11	98280	428206	49208
12	196584	694526	98264
13	292192	1227166	196544
14	786408	2034318	393272
15	1572840	3390527	786464
16	3145704	6619200	1572920
17	6291432	12560192	3145784
18	12582888	23833458	6291488

Figure 5.3: Heap-space (in bytes) used for breadth-first ordering.

allocate a fixed amount of heap-space (2Mb) and this will not change during the execution. Using *memprof* will be of little use since it will output the same values no matter what depth of three is used. The fixed amount of heap-space can be changed manually but this is not really an option when running an experiment like this. Given the information above it was decided that for it was acceptable to use the heap-space usage reported by the Clean program.

The experiment for memory comparison was to run the three breadth-first programs with tree depths 1 to 18. For each depth it was measured (by the measures defined above) how much heap-space the program used. The results can be seen in figure 5.3. A plot of tree depth against use of heap-space was produced can be seen in figure 5.4 and in appendix A.

The data shows that the Borel compiler is allocating exactly 24 bytes for each node in the tree and the graph has the expected exponential form. When comparing the results it can be seen that the Borel data is placed below the Ocaml data and above the Clean data. Remembering that the Ocaml data represents an upper bound and the Clean data represents a lower bound on the amount of memory used by the programs it is not really a surprising result. It is hard to read much from the data since they are not precise. Still it should be emphasised that

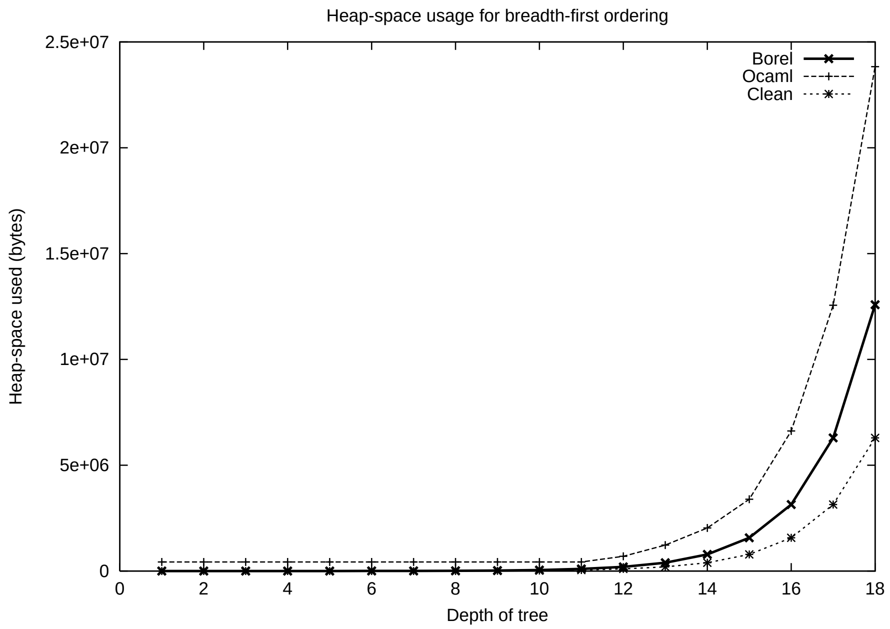


Figure 5.4: Tree depth plotted against usage of heap-space.

the advantage of the Borel program over the others is that it does not allocate extra memory while performing the breadth-first ordering. The heap-space allocated by the program comes from building the tree and not from performing the modifications. The same can not be said about the Ocaml and Clean programs.

6. Conclusions

The aim of this thesis was to describe a functional programming language, and show how to design and implement a compiler that translates the language into malloc-free c-code. The functional language is characterised by a linear first-order type system and a special resource type for maintaining heap-space. Previous work in the area has been done by Martin Hofmann (see papers [6, 7]) and Nicholas Brown (see thesis [4]). This project has been aimed at improving and extending their work to provide a stronger language with new features.

All of the mentioned goals have been fulfilled by defining a new language called Borel. The new features introduced are support for user defined datatypes, a bigger grammar for expressions, and more build-in operations on lists. Designing a compiler for Borel lead to introducing new concrete syntax, which is familiar to SML and Ocaml programmers, and with this it has become very intuitive to write Borel functions and programs. The work on optimising tail recursive functions is improving the efficiency of the generated c-code significantly.

The c-code produced by the implemented compiler was compared to other state-of-the-art functional languages. The comparison with Ocaml and Clean showed that the generated c-code is exceptionally fast. Also it was described how the Borel language has an advantage over Ocaml and Clean when it comes allocation of heap-space. The upper bound on heap-space usage provided by Borel is unique. Both Ocaml and Clean are based on garbage collecting and therefore heap-space usage is unpredictable. The comparison of memory usage did not lead to any conclusion about effectivity of memory management, but Borel was still best in the sense that it will do all the modification of heap-space in an in-place manner. This was not the case for Ocaml and Clean.

The source code for the Borel compiler can be downloaded from:

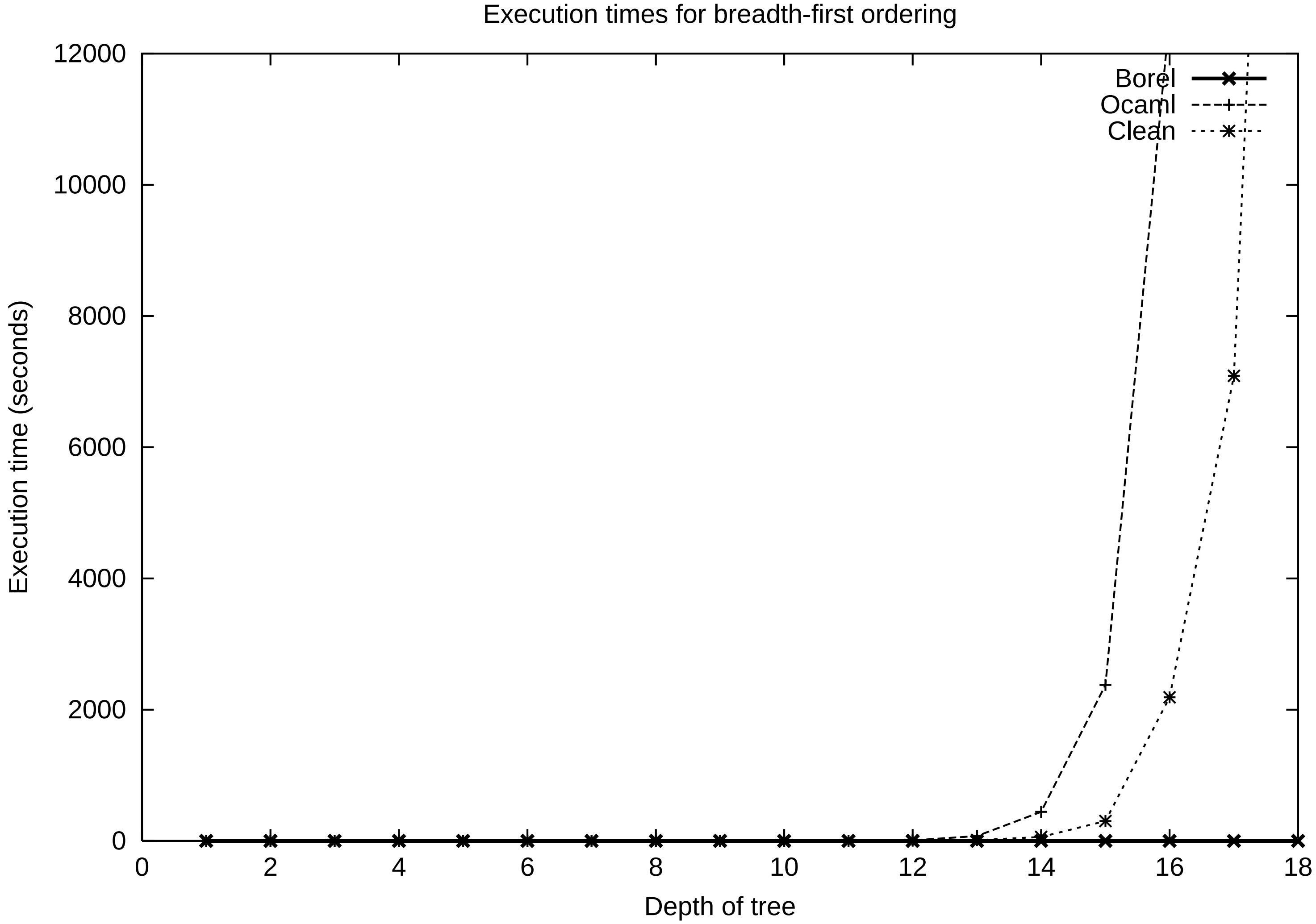
`http://www.daimi.au.dk/~ck/borel.tar.gz`

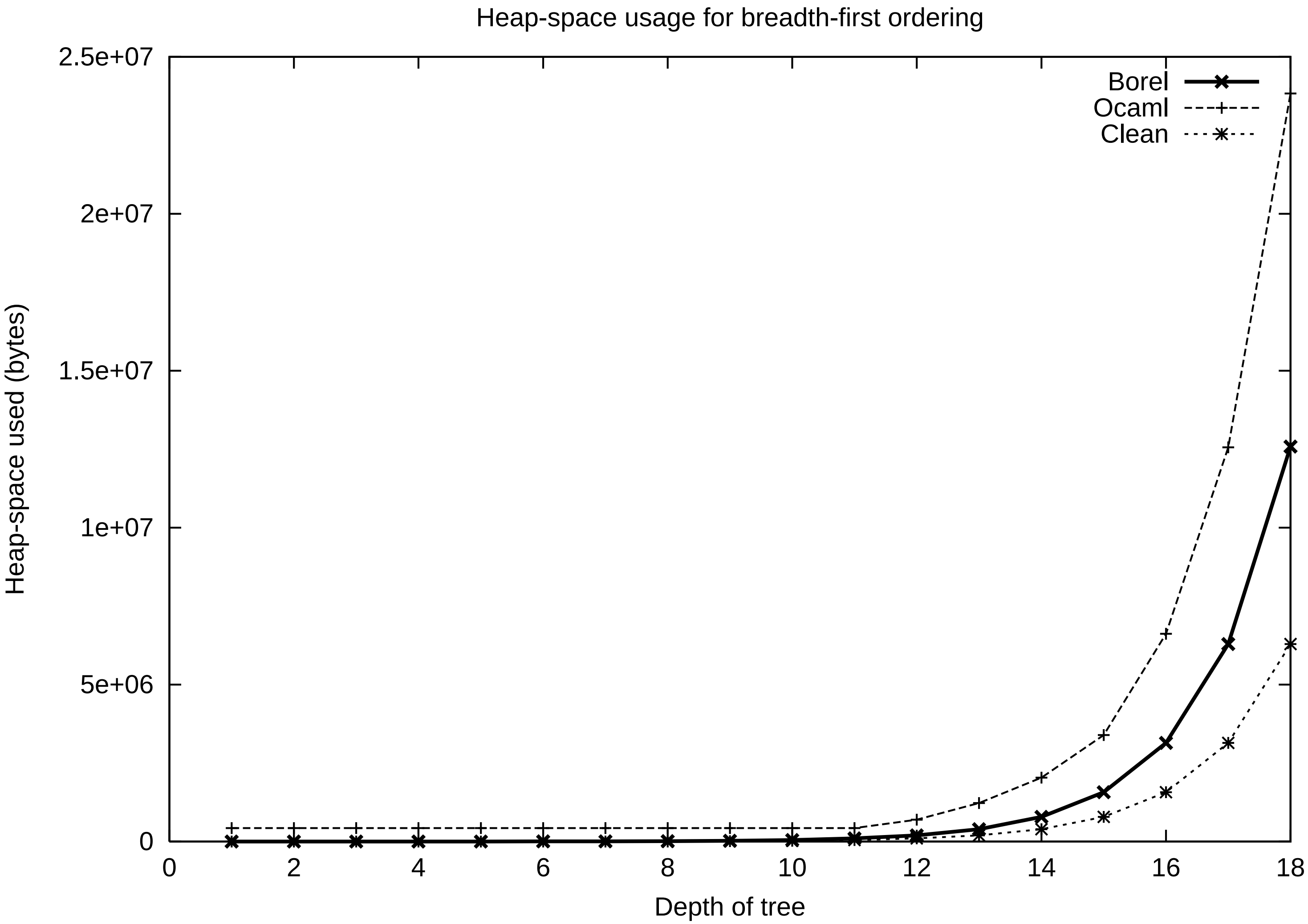
At the moment there are no instructions for using the compiler but it will be added as soon as possible.

A note should be made on the possible applications of the Borel language. Being able to statically prove what resources are needed by a program is important for many real systems. Think of embedded systems with a limited amount of heap-space available or proof-carrying code for computations over the Internet. All predictions say that these areas are the future of computation.

The produced compiler does not have 'industrial strength' in its current form. More work will have to be done on improving syntax and functionality. Yet this project has served to a betterment of the basic definitions introduced in [7]. The results presented in this thesis should be seen as a big step in the direction of an application for solving real-world problems.

Appendix A. Plots




Heap-space usage for breadth-first ordering
Heap-space used (bytes)
Depth of tree
Borel
Ocaml
Clean
0
5e+06
1e+07
1.5e+07
2e+07
2.5e+07
0
2
4
6
8
10
12
14
16
18

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, England, 1998.

[2] Webmaster at Bell Labs Innovations. Standard ml of new jersey documentation pages. *ftp://ftp.research.bell-labs.com/dist/smlnj/doc/SMLNJ/smlnj.tar.Z*, 1997.

[3] Webmaster at Software Technology Research Group. Overview of clean language features. *http://www.cs.kun.nl/∼clean/About_Clean/*.html*, 2000.

[4] Nicholas Brown. Translating a linear functional programming language into c. Master's thesis, Division of Informatics, Computer Science, University of Edinburgh, 2000.

[5] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design.* John Wiley & Sons, Ltd, West Sussex, England, 2000.

[6] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *LICS*, 1999.

[7] Martin Hofmann. A type system for bounded space and in-place update. In G. Smolka, editor, *Programming Languages and Systems*, pages 165–179. Springer LNCS, 2000.

[8] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.00 - documentation and users manual. *http://caml.inria.fr/ocaml/htmlman/index.html*, 2001.

[9] Rinus Plasmeijer and Markus van Eekelen. Concurrent clean version 1.1 - language report. *ftp://ftp.cs.kun.nl/pub/Clean/Clean11/doc/refman11.ps.gz*, 1996.

[10] Didier Rémy, Xavier Leroy, and Pierre Weis. Objective caml - a general purpose high-level programming language. *ERCIM News no. 36*, 1999.