GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Fachbereich 12 Informatik und Mathematik
Institut für Informatik

Diplomarbeit

Functional implementation of an interpreter for a concurrent lambda calculus with futures, reference cells and buffers

Olga Wenge
Studiengang: Diplom-Informatik
Matrikelnummer: 2475727

Frankfurt am Main
28. September 2009

Betreuer: Prof. Dr. Manfred Schmidt-Schauß

Gesetzt am 28. September 2009 um 18:02 Uhr mit LaTeX.

Acknowledgments

I would like to thank Prof. Dr. Manfred Schmidt-Schauß and Dr. David Sabel for providing me with the possibility to participate in an innovative and challenging research project and for the various ideas, suggestions and last but not least for the professional expert guidance.

Thanks to my family, for all the possibilities they have created for me and for giving me all the best support and personal reflection.

Olga Wenge

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Frankfurt am Main, den 28. September 2009,

(Olga Wenge)

Contents

List of Figures

1 Introduction

1.1 Motivation

Concurrent programming is no longer an esoteric approach in the modern world of computing. Nowadays we cannot imagine our life without Internet, real-time systems and interactive applications. To fullfil requirements of new technologies and provide desired interaction and efficiency, programs are to be concurrent.

The main idea of concurrent programming is the execution of sequential subprograms, called *processes*, in parallel. The processes have to "interact"with each other within a program to achieve efficiency and correctness of its execution. Not all sequential programming languages were fit to be used in concurrent programming, and for this reason new concurrent programming languages were designed.

But to "resque"the favored sequential programming languages from their "neglect", programmers have extended the standard libraries with concurrency using various concurrency mechanisms, so called *concurrency primitives*. Concurrent ML [1], Alice ML [2], MultiLisp [3] and Concurrent Haskell [4] are examples of extended functional programming languages, and Join Java [5] and C* [6] are examples of so extended object-oriented programming languages.

There is a wide range of concurrency primitives, but all of them persuit the same purpose: to provide concurrency through communication and synchronization of processes within a program. Therefore, the suggestion that various concurrency primitives could be equivalent was obvious but not evidenced. This open challenge in concurrent programming was the motivation for Manfred Schmidt-Schauß, David Sabel, Jan Schwinghammer, and Joachim Niehren to research synchronization concurrency primitives in the concurrent λ-calculus.

In the paper *On Proving the Equivalence of Concurrency Primitives*[SSNSS08] M. Schmidt-Schauß et al. focuced their research on the interactive behavior of concurrent buffers and concurrent handles in the extended λ-calculus with futures [7] [NSS06] and

[1]http://cml.cs.uchicago.edu/
[2]http://www.ps.uni-sb.de/alice/
[3]http://portal.acm.org/citation.cfm?id=4478
[4]http://www.haskell.org/haskellwiki/GHC/Concurrency
[5]http://joinjava.unisa.edu.au/
[6]http://www.1stworldpublishing.com/books/parallelprogramming.asp
[7]λ (f) is an undelying calculus of Alice ML

type conctructors $\lambda^\tau(\text{fc})$. This work introduces a mutual encoding of concurrent buffers and concurrent handles within the same calculus, and exemplarily prove their equivalence by the following approach.

The approach is based on extending the same λ-calculus with concurrent buffers (the calculus $\lambda^\tau(\text{fc}\mathbf{b})$) and concurrent handles (the calculus $\lambda^\tau(\text{fc}\mathbf{h})$) and observing adequacy of their translations $T_\mathbf{B} : \lambda^\tau(\text{fc}\mathbf{b}) \rightarrow \lambda^\tau(\text{fc}\mathbf{h})$ and $T_\mathbf{H} : \lambda^\tau(\text{fc}\mathbf{h}) \rightarrow \lambda^\tau(\text{fc}\mathbf{b})$ with respect to may- and must-convergence.

Proof techniques in [SSNSS08] require a proper observation of the reduction, what is syntactically very complex and error prone. Use of an interpreter could provide further research with technical support and be used for observation of other concurrency primitives and troubleshooting.

The purpose of this master's thesis is a technical validation of the equivalence of concurrency primitives, introduced by [SSNSS08].

To confirm the existing proof results we implement an interpreter for concurrent λ-calculus with reference cells, concurrent buffers and concurrent handles. The interpreter has to be able to provide the mutual encoding of concurrent buffers and concurrent handles, translate a program from one calculus into another one, and execute it.

1.2 Outline

In the following we give an outline of the chapters of this master's thesis.

Chapter 2 contains all the background information needed before we start with an interpreter implementation. This chapter begins with the introduction of concept of functional programming. Further we give an overview of λ-calculus, an abstract functional language, and of one modern functional programming language Haskell, we use later in Chapter 4 for the technical implementation. Finally, we introduce concurrent programming and describe some common concurrency mechanisms, which are widely-used nowadays.

In Chapter 3 we introduce the calculus $\lambda^\tau(\text{fc})$ and its extensions, calculi $\lambda^\tau(\text{fch})$ and $\lambda^\tau(\text{fcb})$, resulting in the calculus $\lambda^\tau(\text{fchb})$. Further in this chapter we define the mutual encoding of concurrent handles and concurrent buffers and adequate translations of calculi.

Chapter 4 is the technical implementation of the results of Chapter 3, where we design and implement our interpreter for concurrent λ-calculus with reference cells, concurrent buffers and concurrent handles.

In Chapter 5 we report on testing results of our software.

Finally, in Chapter 6 we give an overview of achieved results and make suggestions for the further work.

2 Background

This chapter provides some essential background required for the implementation of our interpreter. The chapter begins with describing of main ideas of the functional concept. Further we introduce Haskell, one of the most important functional languages, and give an overview about pure and extended λ-calculi. In the last section we introduce another programming concept - concurrent programming, and explain how sequential languages can be extended with concurrency primitives.

2.1 Functional Programming

The modern world of information technology offers programmers a wide range of programming languages. They can be roughly divided into four classes: imperative, object-oriented, logic and functional programming languages. Imperative languages like Fortran and Cobol describe computation of a program using statements that change a state of a program. Very popular nowadays languages Java and C++ are object-oriented languges, where a program consits of a collection of cooperating objects. Logic and functional languages are also declarative languages, they describe *what* the program should accomplish, rather than *how* to achive the result. Prolog is an example of logic languages, its progarams are based on the principles of formal logic. Functional languages such as Haskell, Miranda and Lisp based on the idea that a program is a computable function[Pau96].

Functional languages are very peculiar programming languages with a number of important benefits for programmers we introduce in the next sections.

2.1.1 Use of Functions in Functional Programming

In computer science a function can be described as a way of computing an output value from an input value, or as a "black box"that converts a program input into its output[Poh93]:

$$\text{Input x} \rightarrow \boxed{\text{Function f}} \rightarrow \text{Output y} = \text{f(x)}$$

As we already mentioned, functional programming is a programming approach based on the notion of function as the main programming unit. Therefore the theory of functional programming originates from λ-*calculus*, a formal system for description of computable functions. More details about λ-calculus we give in Section 2.3

A functional program is a collection of functions to solve some problem. The execution of a functional program involves the evaluation of an associated function which defines the program result. Its evaluation may also involve the evaluation of many other functions, if a function calls other functions or itself recursively via a function application. For example, the factorial function can be defined as recurrence equitation as follows[SS06]:

```
factorial n = if (n==0) then 1 else n* factorial (n-1)
```

And one simple functional program can look like this:

```
main = factorial 5
```

Functional programming allows functions take another functions as arguments and return new functions as results, such functions are called *higher-order functions*. This peculiarity of functional languages provide an elegant programming style and allows programmers to state the time sequence of events explicitly; that is what, for example, statements do in imperative languages [RL99]. For example, consider functions `map` and `factorial`. The function map (as defined in Haskell) returns a list constructed by appling its first argument (a function) to all items of its second argument (a list):

`map factorial [1, 2, 3]` returns `[1, 2, 6]`

The use of higher-order functions enables transformation of functions with multiple arguments into functions with a single argument to provide a partially evaluation of functions. This transformation is called *currying*[1]. For example, we consider functions `curry` (as defined in Haskell) and `smaller`. The function `curry` converts an uncurried function with two arguments into a curried function with only one argument. The function `smaller` takes the first argument (integer `x`) and returns a function `smaller x`; the function `smaller x` takes the second argument (integer `y`) and returns an integer (the smaller of `x` and `y`) [Tho99]:

```
curry f x y = f (x,y)
```

`smaller :: Integer → (Integer → Integer)` [2]

`smaller x y = if x ≤ then x else y`

2.1.2 Referential Transparency

Variables in functional programming are only used to identify functions and expressions; unlike in imperative languages, they do not identify one or more storage locations, and their values can not be changed using assignments. Forbidding assignments

[1]Is named after Haskell Brooks Curry, an American mathematician and logician.
[2]:: means 'has type'

relieves functional languages from side-effects, i.e. the result of a program depends only on the definition of its functions and values of its arguments. This feature of functional languages is called *referential transparency*. The functional programming languages which do not allow assignments are called *pure*:

"A language that supports the concept that "equals can be substituted for equals"without changing the values of expressions is said to be referentially transparent" [SS89].

In other words, a function can be freely replaced with its value without changing a program's semantics. The principle of referential transparency allow functional languages execute their programs in any evaluation order.

2.1.3 Evaluation Strategies

Functional programs are executed by evaluation their functions to values. Evaluation strategy determines the reduction order and are divided into two basic groups: *strict* and *non-strict*[SS06].

In strict evaluation an argument is always evaluated prior to a function call. This reduction order is called *applicative order*. In non-strict evaluation an argument of a function is evaluated only if its value is needed in the function, such reduction order is called *normal order*.

Additionally to normal-order (or call-by-name) evaluation and applicative-order (or call-by-value) evaluation, functional languages use *lazy* (or call-by-need) evaluation. In lazy evaluation all occurences of an argument can be replaced by its value once it is evaluated (principle of referential transparency). So, lazy evaluation possesses the full power of the normal-order evaluation and is more efficient than the applicative one.

The following example demonstrates that the result of a functional program does not depend on its evaluation strategy. We consider the evaluation of the function `double x` using the described evaluation strategies [Bir98]:

```
double x = add x x

add x y = x + y
```

normal-order evaluation:

```
 double (5 * 4)
→ add (5 * 4) (5 * 4)
→ (5 * 4) + (5 * 4)
→ 20 + (5 * 4)
→ 20 + 20
→ 40                                    (5 reduction steps)
```

applicative-order evaluation:

```
 double (5 * 4)
→ double 20
→ add 20 20
→ 20 + 20
→ 40                                  (4 reduction steps)
```

lazy evaluation:

```
 double (5 * 4)
→ add (5 * 4) (5 * 4)
→ (5 * 4) + (5 * 4)
→ 20 + 20
→ 40                                  (4 reduction steps)
```

We can see that the result value is the same, but the number of reduction steps is different. The number of reduction steps shows that the normal-order evaluation is less efficient.

Depending on their evaluation strategy, functional languages can be divided into lazy, strict and non-strict languages.

2.1.4 Type System

Most functional languages have a strong static type system with type checking during compile-time. Functional languages can also provide polymorphism and user-defined data types. The identity function is an example of polymorphism in Haskell [Bir98]:

```
id:: a → a
id x = x
```

The type of `id` is given as `a → a`, which can be read "for *all* types `a`, a function from `a` to `a`". Here `a` is called *a type variable.*

The other peculiarities of functional languages, such as modularity, pattern matching and monads we give in the next section, where we introduce Haskell, a lazy functional language with strong static type system. More information about functional programming can be found for example in [Bir98], [SS07] or [Mic88].

2.2 Haskell

Haskell Limerick [3]
Our language is state-free and lazy
The rest of the world says we're crazy
But yet we feel sure
That the pure will endure
As it makes derivation so easy.
Joy Goodman (Glasgow University)

In this section we introduce one paticular language, called Haskell as in [Bir98], [OSG08] and [HPF00]. We will use Haskell in Chapter 4 to implement our interpreter.

Haskell[4] is a purely non-strict functional programming language, named after the mathematician and logician Haskell Brooks Curry. Haskell was designed in the late 1980s as a research language for functional programming [Mic88]. Modularity, built-in concurrency and parallelism, lazy evaluation, a strong static type system and polymorphism are important benefits Haskell provides for programmers. We would like to introduce now some important peculiarities of Haskell in details.

2.2.1 Getting Started with Haskell

Haskell has many implementations, but the most common are an open source compiler with interactive environment **GHCi** [5] and an interpreter **Hugs** [6]. Both implementations include a wide range of libraries with built-in types, polymorphic type system, functions and modules.

2.2.2 Modularity

A program in Haskell can consists of multiple modules [OSG08]. These modules declare a set of functions and data type definitions in a closed environment (module). The module mechanism can export its functions completely or partially and can import needed functions from other modules. A module definition in Haskell is very simple:

```
module name (\textit{export list}) where
   (\textit{import list})
```

For example, we define `module Main`, export its functions `main`, `parser` and import fuctions from `module Lexer`, which are declared in the export list in `module Lexer`:

[3] http://haskell.org/haskellwiki/Humor/Limericks
[4] http://www.haskell.org/
[5] http://haskell.org/ghc/
[6] http://haskell.org/hugs/

Type	Name	Example
Integers (limited range)	Int	15
Integers (infinite range)	Integer	4654621363
Reals	Float	3.1415927
Booleans	Bool	True
Characters	Char	'a'
Strings	String	"Hello world!"
List	[a]	[1,2,3] [Integer]
Tupel	(a_1, a_2)	('a', 1) (Char, Integer)

Figure 2.1: Some data types in Haskell

```
module Main (main, parser) where
  import Lexer
  ...
  main = ...
  parser = ...
```

The main advantage of the module mechanism is the reusability of functions inside a program. It also provides a better control over export and import of functions.

2.2.3 Data types

Haskell has various basic and constructed types, which are built-in in Haskell standard library Prelude. Figure 2.1 [OSG08] shows some of these types. Types List and Tupel are polymorphic types, they are defined for every type `a`, for example, `Integer`, `Int`, `Char`, as shown in Figure 2.1. `a`, `a1`, `a2` are called *type variables.*

Haskell allows programmers to define their own types using a `data` declaration. For example, we can define a new data type `Expression` for λ-calculus as follows:

```
data Expression =   V Var
                |   Lambda Var (Expression)
                |   Application Expression Expression
```

Type `Expression` (ot type constructor) has three date constructors as values: `Var`, `Lambda Var (Expression)` and `Application Expression Expression`. `data Expression` is also a recursive data type, hence the data constructors `Lambda Var (Expression)` and `Application Expression Expression` are built using recursive calls of `Expression`.

User-defined types can also be polymorphic. For example, we define a binary tree type [HPF00]:

```
data Binarytree a = Leaf a
                  | Branch (Binarytree a) (Binarytree a)
```

This type is polymorphic, its values might be of any type `a`, for example `Integer`, `Int` or `Char`.

2.2.4 Function definitions

Simple functions in Haskell are declared as equitations:

```
square x = x * x
```

When functions have several cases, they can be declared using an `if then else` construct, guards, a `case`-construct or pattern matching [RL99].

if-then-else construct

This conditional construct is used in Haskell inside the function, for example, the factorial function fact n :

```
fact n  = if (n==0)
          then 1
          else n * fact (n-1)
```

Guards

Guards is a boolean expression, consisting of several equitations with values. If any of these equitations will be evaluated to True, then the function returns a value of this condition. For example, again the factorial function:

```
fact n  | n == if (n==0)
        | otherwise = n * fact (n-1)
```

Pattern matching

Haskell allows only "one-way"matching without backtracking, as, for example, in Prolog. Pattern matching is very practical in the declaration of recursive functions. It looks like a set of equitations of the same function but for different arguments. The factorial function can be defined using pattern matching as follows:

```
fact 0 = 1
fact n = n * fact (n-1)
```

The first equitation defines a case, which matches the argument with pattern 0, and this match will only succeed when the argument has the value 0. The second equitation covers all arguments by matching against the pattern n, so this match will always succeed.

For more convenient use of pattern matching Haskell offers two syntactic sugar constructs: as-patterns (`@`) and wild-cards (`_`) [Jon03]. As-patterns label whole arguments or their parts as a single variable that makes a programm code shorter and clearly represented. For example, we can define a function that duplicates the first element in a list as :

```
f (x:xs)  = x:x:xs
```

or using the as-pattern with a label `s` as follows:

```
f s@(x:xs) = x:s
```

The other construct, wild-card, is used to match against a value we do not care about, i.e. any argument value will match a pattern. For example, for list functions `head` and `tail` only one argument is essential and no matter how the other argument looks like:

```
head (x:_) = x
tail (_:xs = xs
```

Case construction

The case construction in Haskell is another possibility of matching, but matching over values of arguments. For example, an evaluation of function for different expressions may be written as:

```
evaluate e = case e of
             (V Var) -> Var;
             (Lambda Var (Expression) -> ...
```

Sometimes functions need a local scoping in their definitions [Jon03]. For this purpose Haskell provides `let` expressions and `where` clauses:

```
let x = 3 + 2
    y = 5 - 1
in f x y
```

```
f x y  |  y > z          =  ...
       |  y == z         =  ...
       |  y < z          =  ...
     where z = x * x
```

2.2.5 List comprehensions

List comprehensions is another syntactic sugar construct for creation of lists. List comprehensions have the form:

```
[e  | q_1, ... , q_n]
```

e is a generated expression, and `q_1, ..., q_n` are qualifiers. A qualifiers can be a generator of form `x <- exp`, or a boolean expression. As an example we consider a function pairs with two generators:

```
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

returns

```
pairs [1,2,3] [4,5]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```

and with one generator and one boolean expression:

```
[x |  x <- [1..10], even x]
```

returns

```
[2,4,6,8,10]
```

2.2.6 Monads

In the previous section we mentioned absence of side-effects as a great advantage of functional languages. But in some cases, for example, in input-output functions side-effects are needed. For such special cases Haskell provide a monad concept, based on a mathematical notion of monad in category theory.

In Haskell monads are used to build a sequence of actions. Monadic computation is described using a data type for which two main operations `(>>=)` and `return` are defined in the `Monad` type class [HPF00]:

```
class  Monad m  where
    (>>=)           :: m a -> (a -> m b) -> m b
    return          :: a -> m a

    (>>)            :: m a -> m b -> m b
    m >> k          =  m >>= \_ -> k

    fail            :: String -> m a
    fail S          :: error s
```

The operation `return` returns a value without any effect, the operation `(>>=)` is called bind-operator and combines computation when a value is passed from one computation to another. The other operation `(>>)` is used in expressions of `(>>=)` to combine two computations when the second does not need the value of the first one. The operation `fail` is used to raise an error.

For example, consider the following three functions: `f1` is a list comprehension, `f2` is a function with `do`-notation and `f3` is a monadic function. All these functions have the same functionality - to compute the Cartesian product of two sets `[1,2,3]` and `[1,2,3]` [HPF00]:

```
f1 = [(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

f2 = do x <- [1,2,3]
        y <- [1,2,3]
        True <- return (x /= y)
        return (x,y)

f3 = [1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

Cartesian product: `[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]`

We could see that Haskell is a very powerful functional programming language and efforts in its improving are still ongoing. Haskell was extended with concurency mechanisms to Concurrent Haskell [7], to Parallel Haskell [8] and Mobile Haskell [BTL05] wxHaskell[9] provides Haskell with a portable and native GUI library. More information about Haskell projects and Haskell news can be found on the Haskell homepage http://haskell.org/.

2.3 λ-Calculus

The λ-calculus is a very simple but powerful fomal system, invented by the logician Alonco Church in the 1930s to study functions, function application and recursion [Mic88]. The syntax of pure λ-calculus is short and simple, but it can be extended with different syntactic layers, for example, for constants, constructors, integers, booleans, recursions or processes in acordance with the application purpose. In Chapter 3 we will introduce the calculi λ(fch) and λ(fchb), extended λ-calculi with type constructors, processes and concurrency primitives.

2.3.1 Syntax of the Pure λ-calculus

The syntax of pure untyped λ-calculus is very simple and defined as follows [Sab08]:

$$e \in Exp :: \; x \mid \lambda x.e \mid (e_1 \; e_2)$$

$$x \in Variables$$

[7]http://www.haskell.org/haskellwiki/GHC/Concurrency
[8]http://haskell.org/haskellwiki/GHC/Data Parallel Haskell
[9]http://haskell.org/haskellwiki/WxHaskell

Expresstions (or terms) of the form $\lambda x.s$ are called *abstractions*. An abstraction is an anonymous function that binds the variable x within the λ-term s, i.e. the scope of the variable x is the term s. For example, the identity function can be represent in the λ-calculus as follows [Han04]:

$$\lambda x.x$$

Expressions of the form $(s\ t)$ are called *applications* and used to apply a function to its arguments. Applications are assumed to be left associative, so that $(e1\ e2\ e3)$ is the same as $((e1\ e2)\ e3)$. For example, suppose the identity function is applied to itself:

$$(\lambda x.x\ \lambda x.x)$$

Variables that occur in the scope of λ-term are called *bound*, all other variables are called *free*. The set of bound variables $BV(e)$ and the set of free variables $FV(e)$ of an expression e can be inductively defeined as follows [Sab08]:

$FV\ (x) = x$ $\qquad$ $BV\ (x) = 0$

$FV\ (\lambda x.s) = FV\ (e) \setminus \{s\}$ $\qquad$ $BV\ (\lambda x.e) = BV\ (s) \cup \{x\}$

$FV\ (s\ t) = FV\ (s) \cup FV\ (t)$ $\qquad$ $BV\ (s\ t) = BV\ (s) \cup BV\ (t)$

If $FV(s)$ is an empty set, we call s a *closed* term, or *combinator*, otherwise s is an *open* term.

For example, consider the expression [Mic88]

$$\lambda y.xxy$$

with $FV\ (\lambda y.x\ xy) = \{x\}$ and $BV\ (\lambda y.x\ xy) = \{y\}$, so the term $\lambda y.x\ xy$ is open.

2.3.2 Operational Semantics of the Pure λ-calculus

The operational semantics determines the way a program is to be executed, and consists of reduction rules and reduction contexts. To introduce the operational semantics of the pure λ-calculus we need to define first the notions of *substitution* and *context* [Sab08].

Substitution is the process of replacing all free occurrences of the variable x in s with the term t, written $s[t/x]$. Substitution can be inductively defined as follows [Sab08]:

$x[t/x] = t$

$y[t/x] = y$, if $x \neq y$

$(\lambda y.s)[t/x] = \lambda y.(s[t/x])$

$(s_1\ s_2)[t/x] = (s_1[t/x]\ s_2[t/x])$

and $x \notin BV\ (s)$.

Contexts are terms where a subterm is replaced by the constant $[\cdot]$, called the hole marker. The hole marker points where the reduction can take place within a term. The contexts C for λ-terms can be defined as follows [Sab08]:

$$C = [\cdot] \mid \lambda x.C \mid (C\ e) \mid (e\ C)$$

Reduction Rules

The operational semantics of Tthe pure λ-calculus includes three kinds of reduction rules (or conversions) [Sab08]:

(1) α-reduction or α-renaming

(2) β-reduction

(3) η-reduction

(1) *α-reduction* is used for renaming of bound variables. This renaming is needed to avoid unintended catching of variables during substitution. We define α-reduction as binary relation [Sab08]:

$$C[\lambda x.s] \xrightarrow{\alpha} C[\lambda y.s\ [y/x]] \text{ if } y \notin BV\ (\lambda x.s) \cup FV\ (\lambda x.s)$$

The reflexive-transitive closure of $\xrightarrow{\alpha}$ is called α-equivalence. For example, we substitute the term

$(\lambda x.x\ y)\ \{x\ /\ t\}$

To avoid misunderstandings we apply α-reduction:

$(\lambda z.z\ y)\ \{x\ /\ t\}$ and $z \notin BV\ (\lambda x.x\ y)$

$(\lambda x.x\ y)$ and $(\lambda z.z\ y)$ are α-equivalent

Now we substitute:

$\lambda z.z\ x$

(2) *β-reduction* is used for applying functions to their arguments and defined as [Sab08]:

$$C[(\lambda x.s)\ t] \xrightarrow{\beta} C[s\ [x/x]]$$

$(\lambda x.s)\ t$ is also called *redex*[10], an expression that matches the left hand side of a reduction rule.

For example, consider the following β-reduction:

$(\lambda x.\lambda x.x)\ z \rightarrow \lambda z.z$

[10] An acronym for reducible expression.

(3) η-*reduction* is also called an axiom of extensionality. This rule says that two functions are the same if and only if they give the same result for all arguments. The formal definition of η-reduction is [Sab08]

$C[s] \xrightarrow{\eta} C[\lambda x.s\ x],\ x \notin FV\ (s)$

Reduction Contexts

We introduce now reduction contexts and reduction rules of the pure λ-calculus for applicative order (or call-by-value) and normal order (or call-by-name) reductions as set out in [Sab08].

(1) Reduction contexts (R) and reduction rules for call-by-name strategyare defines as:

$$R_{cbn} ::= [\cdot] \mid (R_{cbn}\ t)$$

$$R_{cbn}[(\lambda x.s)\ t] \rightarrow R_{cbn}[s[t/x]]$$

The pure λ-calculus with call-by-name strategy is also called the lazy (call-by-need) λ-calculus. The result values of call-by-name strategy are abstractions for closed terms and a set of *weak head normal forms* for open terms.

A λ-expression is in *weak head normal form (WHNF)* if it is *a head normal form (HNF)* or any abstraction. A λ-expression is in *head normal form* if neither a β-reduction nor an η- reduction is possible. Some examples of HNFs are [Han04]:

- $\lambda x.x$

- $\lambda xy.x$

- $\lambda xy.x\ ((\lambda z.z)y)$

(2) Reduction contexts and reduction rules for call-by-value strategy are defines as:

$$R_{cbv} ::= [\cdot] \mid (R_{cbv}\ t) \mid ((\lambda x.s)\ R_{cbv})$$

$$R_{cbv}[(\lambda x.s)(\lambda y.t)] \rightarrow R_{cbv}[s[\lambda y.t/x]]$$

The values of call-by-value strategy are the same as in the call-by-name one.

As an example we evaluate the following λ-expression usig both strategies [Sab08]:

$$(\lambda x.x\ x)((\lambda y.y)(\lambda z.z)).$$

call-by-value:

$(\lambda x.x\ x)((\lambda y.y)(\lambda z.z))$

$\rightarrow (\lambda x.x\ x)(\lambda z.z)$

$\rightarrow (\lambda z.z)\ (\lambda z.z)$

$\rightarrow (\lambda z.z)$

call-by-name:

$(\lambda x.x\ x)((\lambda y.y)(\lambda z.z))$

$\rightarrow((\lambda y.y)(\lambda z.z))((\lambda y.y)(\lambda z.z))$

$\rightarrow (\lambda z.z)((\lambda y.y)(\lambda z.z))$

$\rightarrow (\lambda z.z)\ (\lambda z.z)$

$\rightarrow (\lambda z.z)$

More details about evaluation strategies in the λ-calculus can be found for example in [Bar84]

2.3.3 Extensions of the λ-calculus

To provide the pure λ-calculus with a type system it can be extended with data constructors (for example boolean oder list conctructors) and typed case-expressions.

The set of data constructors $c \in Constr$ can be defined as [Sab08]

$c \in Constr := \{c(x_1,...,x_{ar(c)})\}$, where $ar(i)$ is the fixed arity of a constructor c_i.

We consider list constructors $c_1(x_1, x_2)$ (Cons) with arity 2, and constructor c_2 (Nil) with arity 0. The application of constructors to the term $\lambda x.x$ results a list:

$(c_1\ (\lambda x.x)\ c_2) \rightarrow [\lambda x.x]$ or in the other form $((\lambda x.x)$:[])

An extension with case-expression provides λ-calculus with conditional constructs. A case-expression with patterns and alternatives has the following form [Sab08]:

$$\text{case } s\ ((c_1(x_{1,1},...,x_{1,ar(c_1)}) \rightarrow s_1)\ .\ .\ .\ (c_n(x_{1,n},...,x_{n,ar(c_n)}) \rightarrow s_n)),$$

where $(c_i(x_1,...,x_{ar(c_i)})\rightarrow s_i)$ is a case-alternative consisting of a pattern $c_i(x_{1,1},...,x_{i,ar(c_i)})$ and an expression s_i. Patterns are constructor application with pairwise distinct variables as arguments. These variables are bound in s.

The β-reduction for case-expression can be defined as follows [Sab08]:

$$C[\text{case } (c_i\ t_1...t_{ar(c_i)})...((c_i\ x_{i,1}...x_{n,ar(c_i)}) \rightarrow s_i)...]$$
$$\longrightarrow C[s_i[t_1/x_{i,1},...,t_{ar(c_i)}/x_{i,ar(c_i)}]]$$

The evaluation of this case-β-rule invokes pattern matching of the expression s by patterns. When a suitable pattern is found, the evaluation results with substitution of variables by the actual parameters within an assosiated alternative.

The λ-calculus can also be extended with recursion, concurrency and non-deterministic constructs. More details about other extensions can be found for example in [Sab08], [Han04] and [Bar84].

2.4 Concurrent Programming

Since the focus of this master's thesis is on the implementation of concurrency primitives, we would like to give now an overview about concurrent programming, its basic mechanisms and its importance in the modern world of computing.

2.4.1 What is concurrent computing?

The modern computing involves more and more systems with components which are concurrently active, i.e. they have an interactional behaviour. So much softwares today are structured as concurrent programs. The branch of computer science which studies this interactional behaviour of computing systems is called *concurrent computing.* Concurrent computing was already introduced in the mid 1960s by E.Dijkstra, T.J. Dekker, P.B. Hansen and C.A.R Hoar and develops rapidly nowadays [Han02].

The notion of *interaction* is fundamental in concurrent computing. There are two types of interactions: physical and virtual. Physical interactions communicate with each other via physical touching, so called *"handshaking".* A cash machine or a ticket vending machine are examples of physical interaction systems. Communication between virtual interactions is effected via virtual links, for example, the linking between pages in the internet. Both physical and virtual interactions could be applied to hardware or software systems. In our work we concern ourselves with virtual interactions in concurrent programs [Sch97].

A concurrent programm is composed of two or more independent sequential programms or threads, called *processes.* The processes can be executed *sequentially* on a single processor by the use of interleavings, or *in parallel* on a set of processors. The oldest examples of concurrent programs are operating systems, they consist of lots of independent processes for controlling hardwares and user activities [Sch97].

The following advantages brought concurrent programs so much popularity [BA06]:

- due to its independency, the processes can be executed in a partial order, that provides the improved flexibility in the whole program (sequental programs force a total order in their execution),
- therefore concurrent programs can be executed in less real time,
- their execution is non-deterministic,
- they support user interaction with applications, therefore they are widely used in reactive systems,
- concurrent programs enable interaction between processes via *concurrency primitives*,
- multiple threads controll every interaction in the program, that improves the manageability of this program.

But concurrent programs have some disadvantages as well [BA06]:

- they are less safe,
- non-determinism can lead to exponential number of interleavings,
- parallel execution can be a reason for deadlocks,
- and finally, concurrent programs can be very expensive.

2.4.2 Concurrency primitives

Any concurrent language is based on a sequential language, extended with different concurrency primitives. Concurrent ML [11] and Concurrent Haskell [12] are examples of such extention on concurrency. Concurrency primitives are typically divided into three kinds [Rep99]:

- concurrency primitives for introducing sequential threads or *processes*,
- concurrency primitives for supporting *communication* between processes,
- and cconcurrency primitieves to enable *synchronization* between processes.

There are also some concurrent primitives, which combine the properties of two or even three kinds. For example, channels in Concurrent ML support the language with communication and synchronization, or futures in Multilisp [13] and Alice ML [14], which combine all three properties into a single concurrency primitive.

Processes

As already described, processes are sequential programs, that can be executed in a partial order. Their number in a concurrent program can be restricted or they can be created in a program run-time if needed. Prosesses are used in many concurrent languages, for example, in Concurrent ML, Concurrent Haskell, Java [15]. Processes in concurrent computation are modeled in accordance with π*-calculus*, a process calculus.

The π-calculus was developed by Robin Milner in the late 1980s as a formal language for for describing and analizing concurrent computational processes.

We can describe the interaction within processes as follows: pairs of processes provide synchronized communication with each other by sending and receiving messages through complementary channels. The content of messages is also channels. The

[11]http://cml.cs.uchicago.edu/
[12]http://www.haskell.org/haskellwiki/GHC/Concurrency
[13]http://en.wikipedia.org/wiki/MultiLisp
[14]http://www.ps.uni-sb.de/alice/
[15]http://www.java.com/de/download/manual.jsp

recipient process reseives a channel and uses this channel for further communication. For more information see [Par01] or [Mil99].

Communication

Communication between processes is very important for their cooperation within a program. Communication allows one process to exchange data with another one, or to have influence over the execution order. Communication primitives are, for example, shared buffers in Java. Working of a shared buffer is very simple: one process writes values into the buffer, the other process reads them.

To avoid asynchronous communication, communication primitives are mostly used together with synchronization primitives or combine the both properties as, for example, channels in Concurrent ML [Rep99].

Channels in Concurrent ML are implemented through two functions, which create an event between a sender and a receiver:

```
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

Communication in the channel is synchronous, both processes must communicate with each other before they go on.

Synchronization

Proper communication between processes can not work without synchronization. Asynchronous communication can lead to deadlocks or erroneous program executions. Semaphores are widely used synchronization primitives. Semaphores restrict access to shared resources during execution, for example, shared memory.

Another example of synchronization primitives is a concurrent primitive `MVar` in Haskell [SPJ96]. This primitive has type:

```
type MVar a
```

The value of type `MVar t` is the name of a mutable location that can be empty or contain a value of type `t`. The MVar primitive is implemented using three functions:

`newMVar :: a -> IO (MVar a)` creates an new location with the supplied value.

`takeMVar :: MVar a -> IO a` return the contents of the MVar, if the location is currently empty, takeMVar waits until it is full, then takes takeMVar and leave the location empty.

`putMVar :: MVar a -> a -> IO ()` puts a value into an MVar, if the location is currently full, putMVar waits until it becomes empty.

In Chapter 3 we introduce some more concurrency primitives, such as handles, futures and buffers, the concurrency primitives of the calculus λ(fchb).

3 Equivalence of Handled Futures and Buffers

In this chapter we introduce the calculi λ^τ(fc) and λ^τ(fchb) as set out in [SSNSS08]. We begin with the description of the calculus λ^τ(**fc**), a typed concurrent λ-calculus with type **c**onstructors and **f**utures. Further we extend it with concurrent **h**andles and concurrent **b**uffers and result in the calculus λ^τ(fc**hb**) with subset calculi λ^τ(fc**h**) and λ^τ(fc**b**). Finally, we introduce adequate translations of concurrent handles and concurrent buffers, using which they can replace one another keeping the same expressiveness.

3.1 The Calculus λ^τ(fc)

We introduce now the calculus λ^τ(fc), a **t**yped call-by-value lambda-calculus with **f**utures, polymorphic data and type **c**onstructors, concurrent threads and reference cells from [SSNSS08]. The calculus λ^τ(fc) is a syntactic extension of the calculus λ(f), a simple typed λ-calculus with *futures*[1] as synchronization concurrency primitives designed in [NSS06].

A future is defined as a place-holder for a value of some evaluated expression in a concurrent program. This value is unknown until the expression is evaluated. Once the evaluation happens and the value becomes available, the associated future is identified with this value, i.e. the future is globally replaced with the value. The value can be a future on its own as well. Such concurrency mechanism together with another concurrency primitive *handle* (or *handled future*) provides implicit and data-driven thread synchronization within a concurrent program: futures together with handles couple concurrent threads and provide their eager evaluation as soon as the value becomes available or keep them waiting for it. [Ali, NSS06] We describe handled futures in Section 3.2.

The calculus λ^τ(fc) is simply-typed including recursive polymorphic type constructors (which must be used monomorphically). In our thesis we intended omit typing and the label $^\tau$, as we implement an untyped interpreter. A weak type restriction on case-expressions is guaranteed by the syntax (see Chapter 4). For every type there we defien a single case$_{type}$ construct, which ensures that the case-alternative are

[1]The calculus λ(f) is a core language of Alice ML (http://www.ps.uni-sb.de/alice/)

exhaustive. The complete description of typing and typing rules could be found in [SSNSS08]. The type checking in case-expressions will be observed in Chapter 4.

3.1.1 Syntax of λ(fc)

The calculus λ(fc) is an abstract model for concurrent programming and consists of multiple processes (or threads), which are to be executed in parallel, but the execution within processes is sequential. The syntax of λ(fc) is shown in Figure 3.1, and as in most concurrent programming languages it consists of two layers: a layer of expressions e and a layer of processes p.

$$
\begin{array}{rl}
e \in \mathit{Exp} ::= & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathbf{exch}\ (e_1, e_2) \mid k(e_1, ..., e_{ar(k)}) \\
 & \mid \mathbf{case}_k\ e\ \mathbf{of}\ \pi_1 \Rightarrow e_1 \mid ... \mid \pi_m \Rightarrow e_m \\
v \in \mathit{Val} ::= & x \mid c \mid \lambda x.e \mid k(e_1, ..., e_{ar(k)}) \\
c \in \mathit{Const} ::= & \mathbf{cell} \mid \mathbf{thread} \mid \mathbf{lazy} \mid \mathbf{unit} \\
p \in \mathit{Proc} ::= & p_1 | p_2 \mid (\nu x)p \mid x\ \mathbf{c}\ v \mid \mid x \overset{susp}{\Leftarrow} e \\
\pi \in \mathit{Pat} ::= & k(e_1, ..., e_{ar(k)})
\end{array}
$$

Figure 3.1: Syntax of λ(fc), where $x \in Var$ and $m \geq 0$

The layer of expressions $e \in \mathit{Exp}$ contains standard λ-expressions (variables, abstractions and applications described in Chapter 2), constants, constructors, **exch**- and **case**-expressions:

Values. Values are defined as in call-by-value λ-calculus as standard λ-expressions, constants and constructors: $v \in Val ::= x \mid c \mid \lambda x.e \mid k(e_1, ..., e_{ar(k)})$ and $Val \subset \mathit{Exp}$.

Constructors. Each constructor k has the form $k(e_1, ..., e_{ar(k)})$ with its fixed arity. For example, *Pair* (e_1, e_2) is a pair constructor with arity 2, *True* is a boolean constructor with arity 0.

exch-expression **exch** (e_1, e_2) provides an atomic exchange of values within a cell. For example, the expression **exch** (x, v) will exchange a current value in a reference cell x for a value v.

case-expressions are typed in the calculus λ(fc). Its patterns $\pi \in \mathit{Pat}$ have the form $k(x_1, ..., x_{ar(k)})$, where all variable x_i are different. The patterns must be non-overlapping and exhaustive within a **case**-expression. The following example shows a valid case-expression (1) with list-patterns and a non-valid case-expression (2) with overlapping boolean-patterns:

(1) $\mathbf{case}_{\mathit{List}}\ e\ \mathbf{of}\ \mathit{Nil} \Rightarrow \mathit{True} \mid$ (x:xs) $\Rightarrow \mathit{False}$

(2) $\mathbf{case}_{\mathit{Bool}}\ e\ \mathbf{of}\ \mathit{True} \Rightarrow e_1 \mid \mathit{True} \Rightarrow e_2$

Constants. We use four higher-order constants for creating new components within expressions. The constants **thread** and **lazy** introduce eager threads and lazy threads with futures, respectively. For example, an application (**thread** v) spawns a new thread to bind a value v to a future. Analogical to **thread** the constant **cell** is used for creating new reference cells: (**cell** v) creates a new cell with value v. The constant **unit** is used as a dummy value.

The layer $p \in Proc$ is a layer of processes. The process p can contain one or more components executed in parallel $(p_1 \mid p_2)$, but it can also create new components in run-time with a new name operator $(\nu x)p$. These actions succeed in accordance with the process syntax of π-calculus and the rules of structural congruence shown in Figure 3.2.

$p_1 \mid p_2 \equiv p_1 \mid p_2$	- commutativity of parallel processes
$(p_1 \mid p_2) \mid p_3 \equiv p_1 \mid (p_2 \mid p_3)$	- associativity of parallel processes
$(\nu x)(\nu y)p \equiv (\nu y)(\nu x)p$	- multiple communication scopes
$(\nu x)(p_1) \mid p_2 \equiv (\nu x)(p_1 \mid p_2)\ ,\ x \notin fv(p_2)$	- scope extrusion

Figure 3.2: Structural congruence of $\lambda(fc)$-processes

The basic process components are eager and lazy threads and reference cells:

Eager threads. Eager threads in the calculus λ(fc) are concurrent components of the form $x \Leftarrow e$, where x is *a concurrent future* and e is an expression. The task of an eager thread $x \Leftarrow e$ in a prosess p is to bind a concurrent future x to a value of the expression e, if the evaluation of e terminates. The value of e can be another future as well, for example, $x \Leftarrow y \mid y \Leftarrow e$. Eager threads $x \Leftarrow e$ are also considered as recursive equitations $x = e$ (directed as $e = x$) [NSS06].

Lazy threads. Lazy (or suspended) threads $x \overset{susp}{\Leftarrow} e$ are concurrent componets as well. They are called **lazy**, because they start their computation only if the value of the future x is required, x is called here a *lazy future*. In that case, the status *lazy* will be changed to *eager* and the threads will behavior as eager threads in the next evaluation step.

Reference cells. We define reference cells in the calculus λ(fc) as usual storage locations for values: x **c** v means, that the reference cell x is associated a with a value v.

3.1.2 Variable Bindings and Well-Formedness

In accordance with the syntax of the the calculus λ(fc) we derive only two binding constructs for variables: λ-binders and new name operators (νx). All introduced variables are to be bound. For example, we consider two processes p_1 and p_2. The process p_1 is closed, and the process p_2 is open, because the variables y and $\mathbf{w}$ are not bound:

(1) $p_1 = (\nu x)(\nu y)(x \Leftarrow \lambda z.z \mid y \ \mathbf{b} \ x)$

(2) $p_2 = (\nu x)(x \Leftarrow \lambda z.w \mid y \ \mathbf{b} \ x)$

For creation of new components and α-renaming we define sets of free variables *fv(e)* and *fv(p)* for expressions and prosesses respectively.

We define also a well-formedness rule for processes in the calculus λ(fc): a process is well-formed, if none of its components introduce any variable more then once. For example, we consider again two processes p_3 and p_4. The process p_3 is well-formed, all the variables in the components are unique; the process p_4 is not well-formed, the variable x is used in the thread and in the reference cell:

(3) $p_3 = (\nu x)(\nu y)(\nu z)(x \overset{susp}{\Leftarrow} z \mid y \ \mathbf{c} \ v \mid z \overset{susp}{\Leftarrow} v)$

(4) $p_4 = (\nu x)(x \Leftarrow y \mid x \ \mathbf{c} \ v \mid y \overset{susp}{\Leftarrow} v)$

3.1.3 Operational Semantics and Contexts of λ(fc)

The operational semantics defines how a program is executed, i.e. how reduction rules have to be applied to processes. Evaluation contexts are used to specifiy positions in prozesses where reduction rules are applied. In genereral, contexts are terms terms where exactly one subterm is replaced with the constant [.], called hole marker.

We describe now the operational semantics of the calculus λ(fc). It is a small-step semantics, as the evaluation of a process p consists of multiple steps (or reductions) in accordance with reduction rules shown in Figure 3.5. So we can describe it as a binary relation $p \rightarrow p'$, where p is an initial process and *p'* is a process p after one small-step reduction.

We define two classes of contexts: an expression context C and a process context D. The hole maker can replace only one subexpresion in the context C, and only one component in a prosess p in the contex D. The evaluation contexts of the calculus λ(fc) are shown in Figure 3.4. *ECs E* are the evaluation contexts for a traditional call-by-value evaluation of expressions, *Future ECs F* Future ECs are particular evaluation contexts which mark the future strict positions, i.e. positions which enforce the evaluation of concurrent and lazy future, and *Process ECs D* are evaluation contexts for processes.

As shown in Figure 3.4, the evaluation of expressions can begin only in an eager thread: $E ::= x \Leftarrow \tilde{E}$ or $F ::= x \Leftarrow \tilde{F}$; and involves evaluation contexts $\tilde{E}$ and $\tilde{F}$ to execute a small-step evaluation of a program. For example, we consider a process p_5:

(5) $p_5 = (\nu x)(\nu y)(x \ \mathbf{c} \ \mathit{True} \mid y \Leftarrow ((\lambda x.x) \ \mathit{False}))$

The hole marker for D context can be positioned only within one of two components:

ECs	$E ::= x \Leftarrow \tilde{E}$
	$\tilde{E} ::= [] \mid \tilde{E}e \mid v\tilde{E} \mid \mathbf{exch}\ (\tilde{E}, e) \mid \mathbf{exch}\ (v, \tilde{E})$
	$\mid \mathbf{case}\ \tilde{E}\ \mathbf{of}(\pi_i \Rightarrow e_i)^{i=1...n} \mid k(v_1, ...v_{i-1}, \tilde{E}, e_{i+1}, ..., e_n)$
Future ECs	$F ::= x \Leftarrow \tilde{F}$
	$\tilde{F} ::= \tilde{E}[[]v] \mid \tilde{E}[\mathbf{exch}\ ([], v)] \mid \tilde{E}[\mathbf{case}[]\ \mathbf{of}(\pi_i \Rightarrow e_i)^{i=1...n}]$
Process ECs	$D ::= [] \mid p\vert D \mid D\vert p \mid (\nu x)D$

Figure 3.3: Evaluation contexts of $\lambda(fc)$

$$[.] \mid y \Leftarrow ((\lambda x.x)\mathit{False})\ \mathbf{or}\ x\ \mathbf{c}\ \mathit{True} \mid [.]$$

The hole marker for C context can be positioned only within an eager thread:

$$(\nu x)(\nu y)(x\ \mathbf{c}\ \mathit{True} \mid [.])$$

$(y \Leftarrow (\lambda x.x)\ \mathit{False})$ matches $E ::= x \Leftarrow \tilde{E}$ evaluation context, as $((\lambda x.x)\ \mathit{False})$ is not a future. So, the following evaluation proceeds with the evaluation context E and evaluation of $((\lambda x.x)\ \mathit{False})$ with the β-reduction rule (see Figure 3.4 below).

We define reduction rules of the calculus $\lambda(fc)$ as shown in Figure 3.4.

$(\beta-\text{CBV}(ev))$	$[(\lambda x.e)v] \rightarrow E[e[v/x]]$
$(\text{THREAD.NEW}(ev))$	$E[\mathbf{thread}\ v] \rightarrow (\nu z)(E[z] \mid z \Leftarrow vz)$
$(\text{FUT.DEREF}(ev))$	$F[x] \mid x \Leftarrow v \rightarrow F[v] \mid x \Leftarrow v$
$(\text{CELL.NEW}(ev))$	$E[\mathbf{cell}\ v] \rightarrow (\nu z)(E[z] \mid z\ \mathbf{c}\ v)$
$(\text{CELL.EXCH}(ev))$	$E[\mathbf{exch}(z, v_1)] \mid z\ \mathbf{c}\ v_2 \rightarrow E[v_2] \mid z\ \mathbf{c}\ v_1$
$(\text{LAZY.NEW}(ev))$	$E[\mathbf{lazy}\ v] \rightarrow (\nu z)(E[z] \mid z \overset{susp}{\Leftarrow} v)$
$(\text{LAZY.TRIGGER}(ev))$	$F[x] \mid x \overset{susp}{\Leftarrow} e \rightarrow F[x] \mid x \Leftarrow e$
$(\text{CASE.BETA}(ev))$	$E[\mathbf{case}\ k_j(v_1, ..., v_{ar(k_j)})\ \mathbf{of}\ (k_i(x_1, ..., x_{ar(k_i)} \Leftarrow e_i)^{i=1...n}]$
	$\rightarrow E[e_j[v_1/x_1, ..., v_{ar(k_j)}/x_{ar(k_j)}]]$

Important notes: The reduction rules may be applied only to well-formed processes. All new binders use free variables $z \in fv$. α-renaming is to be performed after the following rules: $(\beta-\text{CBV}(ev))$, $(\text{CASE.BETA}(ev))$ and $(\text{FUT.DEREF}(ev))$, before applying the next reduction rule.

Figure 3.4: Reduction rules

The reduction rules in Figure 3.4 can be described as follows:

$(\beta-\mathbf{CBV}(ev))$ is a standard is thecall-by-value β-reduction as we described in Chapter 2, for example:

$$(\nu y)(y \Leftarrow (\lambda x.x)True) \rightarrow (\nu y)(y \Leftarrow True)$$

(**CASE.BETA**(ev)) is a standard case-β-reduction as well:

$(\nu x)(x \Leftarrow \textit{\textbf{case}}\ (Cons\ x\ y)\ \textit{\textbf{of}}\ ((Cons\ z\ zs) \rightarrow zs\ ,\ Nil \Rightarrow Nil) \rightarrow (\nu x)(x \Leftarrow y)$

(**THREAD.NEW**(ev)) creates a new eager thread with a concurrent future z :

$(\nu x)(x \Leftarrow \textbf{thread}\ \textit{True}) \rightarrow (\nu x)(\nu z)\ (x \Leftarrow \text{True} \mid \text{z} \Leftarrow \text{True z})\ ,\ z \in fv(p)$

(**LAZY.NEW**(ev)) creates a new lazy thread with a lazy future z :

$(\nu x)(x \Leftarrow \textbf{lazy}\ \textit{True}) \rightarrow (\nu x)(\nu z)\ (x \Leftarrow \text{True} \mid z \overset{susp}{\Leftarrow} \text{True z})\ ,\ z \in fv(p)$

(**CELL.NEW**(ev)) creates a new reference cell z with the value v:

$$(\nu x)(x \Leftarrow (\textbf{cell}\ False)) \rightarrow (\nu x)(\nu z)(x \Leftarrow z \mid z\ \textbf{c}\ False) \text{ and } z \in fv(p)$$

(**FUT.DEREF**(ev)) is used to derefence the value of a future, if it is needed, i.e the value of the future will be copied to replace its associetad future in a process, for example :

$$(\nu y)(\nu x)(y \Leftarrow x \mid x \Leftarrow v \rightarrow (\nu y)(\nu x)(y \Leftarrow v \mid x \Leftarrow v)$$

(**LAZY.TRIGGER**(ev)) is also used for the future evaluation context. It requires to start the computation of a lazy thread, i.e. this lazy thread becomes an eager thread in the next evaluation step:

$$(\nu y)(\nu x)(y \Leftarrow x \mid x \overset{susp}{\Leftarrow} v) \rightarrow (\nu y)(\nu x)(y \Leftarrow x \mid x \Leftarrow v)$$

(**CELL.EXCH**(ev)) replaces an actual value of a reference cell value with a new value v_1 and returns the first one v_2:

$$(\nu x)(\nu z)(x \Leftarrow \textbf{exch}(z, v_1) \mid z\ \textbf{c}\ v_2 \rightarrow (\nu x)(\nu z)(x \Leftarrow v_2 \mid z\ \textbf{c}\ v_1)$$

In Chapter 4 we give more detailed explanation of applying of reduction rules in different evaluation contexts.

3.2 The Calculus λ(fch)

We extend now the calculus λ(fc) with a new concurrency primitive, a concurrent *handle* or *a handled future* and result in the calculus λ(fch). Handled futures are basic synchronization primitives in the calculus λ(f) designed in [NSS06]. They are used to couple futures with values.

The changes in the syntax and operational semantics are shown in Figures 3.5 and 3.6.

The layer of expressions is now extended with a new higher-order constant **handle**. This constant is used to create new handles within expressions. The layer of processes becomes two basic components: handles and used handles:

$$\begin{array}{ll} c \in \mathit{Const} ::= & \textbf{handle} \; ... \\ p \in \mathit{Proc} ::= & y \; \textbf{h} \; x \mid y \; \textbf{h} \; \bullet \; ... \end{array}$$

Figure 3.5: Syntax extension of λ(fc) for λ(fch)

Handles. Our handle has the form y **h** x, where y is the name of a handle and x is the future. This handle y could be used to bind some value to the future x. Each handle may be used only once.

Used handles. If a handle was alredy used it becomes a used handle. Used handles have the following form: y **h** •, where y is the name of a used handle.

We explain now the new reduction rules shown in Figure 3.6 with concurrent handles and give some examples:

(**HANDLE.NEW**(ev)) is analogical to the other rules for creating new componets. It spawns a new concurrent handle z with a new future z', for example:

$$(\nu x)(x \Leftarrow \textbf{handle unit}) \rightarrow (\nu x)(\nu z)(\nu z')(x \Leftarrow \textbf{unit} \; z \; z' \mid z' \; \textbf{h} \; z \;)$$

(**HANDLE.BIND**(ev)) uses an existing handle x to bind its future with a value v. If this binding is successfil, a handle becomes a used handle and can not be used once more. For example:

$$(\nu x)(\nu y)(\nu z)(\mathrm{y} \Leftarrow x \; \mathit{True} \mid x \; \textbf{h} \; z) \rightarrow (\nu x)(\nu y)(\nu z) \; (\mathrm{y} \Leftarrow \textbf{unit} \mid \mathrm{z} \Leftarrow \mathit{True} \mid x \; \textbf{h} \; \bullet)$$

$$\begin{array}{ll} (\text{HANDLE.NEW}(ev)) & E[\textbf{handle} \; v] \rightarrow (\nu z)(\nu z')(E[vzz'] \mid z' \; \textbf{h} \; z) \\ (\text{HANDLE.BIND}(ev)) & E[xv] \mid x \; \textbf{h} \; y \rightarrow E[\textbf{unit}] \mid y \Leftarrow v \mid x \; \textbf{h} \; \bullet \end{array}$$

Figure 3.6: Extension of operational semantics of λ(fc) for λ(fch)

3.3 The Calculus λ(fcb)

We extend the calculus λ(fc) again with another concurrency primitive, with *a concurrent one-place buffer*, and result in the calculus λ(fcb).

As shown in Figure 3.7, the new syntax is extended with two new higher-order constants **buffer** and **get** and a new expression **put** (e_1, e_2) for realizing the main actions of a buffer. The constant **buffer** is used to create new buffers, the constant get obtains the content of non-empty buffer, and the expression **put** (e_1, e_2) stores a new value e_2 on an emty buffer e_1.

The new components in the layer of processes are defined as follows:

$e \in \mathit{Exp} ::=$	$\mathbf{put}\ (e_1, e_2)$ ...
$c \in \mathit{Const} ::=$	$\mathbf{buffer} \mid \mathbf{get}$ ...
$p \in \mathit{Proc} ::=$	$x\ \mathbf{b}$ - $\mid x\ \mathbf{b}\ v$ ...

Figure 3.7: Extension of syntax for λ(fcb)

Concurrent buffers. They have the form $x\ \mathbf{b}\ v$, where x is the name of a buffer and v is its content.

Empty buffers. Empty buffers $x\ \mathbf{b}$ - have no content, but the content can be filled using **put**-expression.

Figure 3.8. introduces the extensions of E and F contexts and reduction rules. We explain now the new rules as usually with examples:

(**BUFF.NEW**(ev)) creates a new empty buffer x:

$$(\nu y)(y \Leftarrow (\ \mathbf{buffer}\ v) \rightarrow (\nu y)(\nu x)(y \Leftarrow x \mid x\ \mathbf{b}\ \text{-}\ )$$

(**BUFF.PUT**(ev)) and stores a value **v** in an empty buffer with the name **x**:

$$(\nu y)(\nu x)(y \Leftarrow \mathbf{put}\ (x, \mathit{True}) \mid x\ \text{b -}) \rightarrow (\nu y)(\nu x)(y \Leftarrow \mathbf{unit} \mid x\ \mathbf{b}\ \mathit{True}\ )$$

(**BUFF.GET**(ev)) searches in the process components for a non-empty buffer with the name x and obtains its content empting the buffer:

$$(\nu y)(\nu x)(y \Leftarrow (\ \mathbf{get}\ x) \mid x\ \mathbf{b}\ \mathit{False}) \rightarrow (\nu y)(\nu x)(y \Leftarrow \mathit{False} \mid x\ \mathbf{b}\ \text{-}\ )$$

(BUFF.NEW(ev))	$E[\mathbf{buffer}\ v] \rightarrow (\nu x)(E[x] \mid x\ \mathbf{b}\ -)$
(BUFF.PUT(ev))	$E[\mathbf{put}(x, v)] \mid x\ \mathbf{b}\ - \rightarrow E[\mathbf{unit}] \mid x\ \mathbf{b}\ -$
(BUFF.GET(ev))	$E[\mathbf{get}\ x] \mid x\ \mathbf{b}\ v \rightarrow E[v] \mid x\ \mathbf{b}\ -$
ECs	$E ::= \mathbf{put}\ (\tilde{E}, e) \mid \mathbf{put}\ (v, \tilde{E})$
Future ECs	$F ::= \tilde{E}[\mathbf{put}\ ([], v)] \mid \tilde{E}[\mathbf{get}[]]$
	Important note: get-operation on an empty buffer, as well as a **put**-operation on a filled buffer, cannot be performed. I.e., every thread executing such an operation blocks until the state of the buffer changes.

Figure 3.8: Extension of operational semantics of λ(fc) for λ(fcb)

3.4 The Calculus λ(fchb)

We define the calculus λ(fchb) as the calculus λ(fc) with both concurrency primitives, concurrent buffers and concurrent handles, including their extensions in the syntax an operational semantics. The calculi λ(fch) and λ(fcb) we consider to be the subset calculi of λ(fchb).

3.4.1 Successfulness of processes in λ(fchb)

A well-formed process p is successful if all its components are successful: eager threads are successful if their identifiers are bound via a chain $x \Leftarrow x_1 \mid ... \mid x_n \Leftarrow v$ to a non-variable value, a reference cell, a lazy thread, a handle, a used handle, a buffer or an empty buffer; the other components are always succesful. For example, consider the following processes p_6 and p_7 (we omit here new name operators):

(6) $p_6 = x \Leftarrow y \mid y \Leftarrow z \mid z \; \mathbf{c} \; \mathit{True}$

(7) $p_7 = x \Leftarrow y \mid z \; \mathbf{c} \; \mathit{True}$

The process p_6 is successful, the process p_7 is not successful, its identifier x is bound to a variable y. Tho following processes are not successful as well [SSNSS08]:

(6) $p_8 = x \Leftarrow x$

(7) $p_9 = x \Leftarrow (\lambda u, v.v) \; (y \; \mathbf{unit}) \mid y \Leftarrow (\lambda u, v.v) \; (x \; \mathbf{unit})$

The process p_9 is a cylic chain or so called black hole, and the process p_8 is a deadlocked process.

3.4.2 Convergence and Divergence of processes in λ(fchb)

We define now predicates for may- and must-convergence and may- and must-divergence for the calculus λ(fchb).

A process p is may-convergent ($p \downarrow$) if there is at least one sequence of call-by-value small-step reductions $p \xrightarrow{*} p'$ and p' is successful, otherwise p is *must-divergent* ($p \Uparrow$):

$$p \downarrow := \exists p' \; (p \xrightarrow{*} p' \wedge p' \text{ is successful}) \; ; \neg p \downarrow \Leftrightarrow p \Uparrow$$

A process p is *must-convergent* ($p \Downarrow$) if all reduction successors p' (i.e. $p \xrightarrow{*} p'$) are *may-convergent*, otherwise p is *may-divergent* ($p \uparrow$):

$$p \Downarrow := \forall p' \; (p \xrightarrow{*} p' \Rightarrow p \downarrow) \; ; \neg p \Downarrow \Leftrightarrow p \uparrow$$

A process p is *total must-convergent* ($p \Downarrow_{to}$) if p is must convergent and there is no infinite reduction $p \rightarrow p_1 \rightarrow p_2 \rightarrow ...$

More information about may- and must-convergence can be found, for example, in [SSS08] and [Sab08].

3.5 Equivalence of Handled Futures and Buffers

In this section we introduce the mutual encoding of the calculi λ(fcb) and λ(fch) as set out in [SSNSS08]. We argue that the translations $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch) from buffers to handles and $T_H : \lambda$ (fch) $\rightarrow \lambda$(fcb) from handles to buffers are adequate. From this follows that both concurrency primitives concurrency handles and concurrency buffers are equivalent and they can replace one another in the same environment (here the calculus λ (fchb)). The explicit proofs of contextual equivalence and adequate translations can be found in [SSNSS08]. In Chapter 4 we will present a functional implementation of both translations and in Chapter 5 present testing results of some of our statements.

In our encoding we use syntactic sugar shown in Figure 3.9

let $x = e_1$ **in** e_2	$:= (\lambda x.e_2)\ e_1$
(e_1,e_2)	$:= (\lambda\_.e_2)\ e_1$
$\lambda\_.e$	$:= \lambda x.e_2$ and $x \in fv(e)$
(e_1,e_2,e_3)	$:= e_1,(e_2,e_3)$
wait e	$:=$ **case** e **of** $\{True \Rightarrow True\ ,\ False \Rightarrow True\}$
newhandled	$:=$ **handle** $\lambda f.\lambda h.\langle h, f\rangle$
$\lambda.\langle x_1, ...x_n\rangle.e$	$:= \lambda y.$ **case** y **of** $\{(x_1, ..., x_n) \Rightarrow e\}$ and $y \in fv(e)$
$\lambda.\langle\langle x_1, ..., x_n\rangle, v\rangle.e$	$:= \lambda y.$ **case** y_1 **of** $\{(z, v) \Rightarrow$ **case** z **of** $\{(x_1, ..., x_n) \Rightarrow e\}\}$ and $y \in fv(e)$

Figure 3.9: Syntatic sugar

3.5.1 Encoding Buffers Using Handled Futures

We begin with the translation $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch). The translation T_B is an implementation of expressions put, get and buffer and process components buffer and empty buffers using futures, handles and reference cells, i.e. using the calculus λ(fch), see Figure 3.9.

We encode a one-place buffer as a tupel of four reference cells:

$$\textit{concurrent buffer} \mathrel{\hat{=}} \langle x_p\ \mathbf{c}\ y_1,\ x_g\ \mathbf{c}\ y_2,\ x_s\ \mathbf{c}\ y_2,\ x_h\ \mathbf{c}\ y_2\rangle \text{ where } y_i \in Val$$

The reference cells x_p and x_g may contain True or a handled future, but they both may not contain handled futures or True simultaneously. x_p and x_g are guards and take care that the buffer operations put and get are correctly applied.

$$
\begin{array}{lcl}
\mathit{buffer} & \hat{=} & \lambda\_.\ \text{let } \langle h, f\rangle = \text{newhandled}, \langle h', f'\rangle = \text{newhandled}, \\
 & & \quad \mathit{putg} = \mathbf{cell}(\text{true}),\ \mathit{getg} = \mathbf{cell}(f), \\
 & & \quad \mathit{stored} = \mathbf{cell}(f'),\ \mathit{handler} = \mathbf{cell}(h) \\
 & & \text{in } \mathbf{thread}\ \lambda\_.\langle \mathit{putg}, \mathit{getg}, \mathit{stored}, \mathit{handler}\rangle \text{ end} \\
 & & \\
\mathit{put} & \hat{=} & \lambda\langle\langle x_p, x_g, x_s, x_h\rangle, v\rangle. \\
 & & \text{let } \langle h, f\rangle = \text{newhandled} \\
 & & \text{in wait } (\mathbf{exch}(x_p, f)); \\
 & & \quad\ \mathbf{exch}(x_s, v); \\
 & & \quad (\mathbf{exch}(x_h, h))(\text{true}) \\
 & & \text{end} \\
 & & \\
\mathit{get} & \hat{=} & \lambda\langle x_p, x_g, x_s, x_h\rangle. \\
 & & \text{let } \langle h, f\rangle = \text{newhandled} \\
 & & \quad \langle h', f'\rangle = \text{newhandled} \\
 & & \text{in wait } (\mathbf{exch}(x_g, f)); \\
 & & \quad \text{let } v = \mathbf{exch}(x_s, f'); \\
 & & \quad \text{in } (\mathbf{exch}(x_h, h))(\text{true});\ v \text{ end} \\
 & & \text{end} \\
 & & \\
x\ \mathbf{b}\ \text{-} & \longmapsto & (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h) \Leftarrow \langle x_p, x_g, x_s, x_h\rangle \\
 & & \mid (\nu h)(\nu f)(\nu h')(\nu f')(h\ \mathbf{h}\ f \mid h'\ \mathbf{h}\ f' \mid x_p\ \mathbf{c}\ \mathit{True} \mid x_g\ \mathbf{c}\ f \mid x_s\ \mathbf{c}\ f' \mid x_h\ \mathbf{c}\ h) \\
 & & \\
x\ \mathbf{b}\ v & \longmapsto & (\nu x_p)(\nu x_g)(\nu x_s)(\nu x_h) \Leftarrow \langle x_p, x_g, x_s, x_h\rangle \\
 & & \mid (\nu h)(\nu f)(x_p\ \mathbf{c}\ f \mid x_g\ \mathbf{c}\ \mathit{True} \mid x_s\ \mathbf{c}\ T_B\ (v) \mid x_h\ \mathbf{c}\ h) \\
 & & \\
T_B(p) & \hat{=} & \text{homomorphically wrt. the term structure}
\end{array}
$$

Figure 3.10: Translation T_B : λ (fcb) $\rightarrow \lambda$(fch)

The reference cell x_s is a storage for an actual buffer content, and the referenc cell x_h is handler for a handled future contained either in x_p or in x_g. So, we can describe the possible invariants as follows:

- if x_p contains a handled future that means the buffer is not empty, so the operation put can not be applied,

- if x_p contains True, then the value in x_s is a "gabarge", i.e. the buffer is empty,

- if x_g contains a handled future, that means the buffer is empty and the operation get can not be applied,

- if x_g contains True, that means the value in the referenc cell x_s is a current value of the buffer, i.e the buffer is not empty.

For example, we consider the translation of x **b** - into the calculus λ(fch) in Figure 3.10. The empty buffer x is implemented as a tupel of four reference cells $\langle x_p$,x_g,x_s,$x_h\rangle$ with the following contents :

- x_p contains *True*, as the buffer is empty, and the operation put can be applied,

- x_g contains a handled future *f*, so the operation get will be blocked,

- x_s contains a "garbage"value *f'*,

- x_h contains a handler h, a name of a handle that will bind the future f (h **h** f)

The encoding of the constant **buffer** is analocical to the encoding of x **b** -. It spawns a new empty buffer in the form of referece cells as well: $\langle$*putg*,*getg*,*stored*,*handler*$\rangle$.

In the implementation of the operation **put** (x,v) (or here $\langle\langle x_p$,x_g,x_s,$x_h\rangle$, $v\rangle$) it is necessary to make sure that the buffer is empty. This is tested in the expression **wait** (**exch**(x_p,f)), if it returns $True$, then the buffer is empty and the value v will be stored in x_s. After assignig the value x_g will be set to $True$ to demonstrate that the buffer is now not empty and to block the next operation *put*.

The encoding of the operation get is analogical to the operation put. Here will be first tested wether the buffer is not empty, i.e. if **wait** (**exch**(x_g,f)) returns $True.$ In this case the actuel value of the reference cell x_s will be bound to a future with the handle h and then overwrite with a "garbage"value f'.

3.5.2 Encoding Handles Using Buffers

In this section we introduce the translation T_H : λ (fch) $\rightarrow$ λ(fcb). The tranlation T_H is given in Figure 3.11 shows how handled futures can be encoded using one-place buffers.

$$
\begin{array}{lcl}
T_H(\textbf{handle}) & \hat{=} & \lambda x.\ \text{let } f' = \textbf{buffer unit} \\
 & & \qquad\quad f = \textbf{lazy}\ (\lambda\_.\ \text{let } v = \text{get } f' \text{ in put } (f',v)\ ;\ v) \\
 & & \qquad\quad h = \textbf{thread}\ (\lambda\_.\lambda z.\text{put } (f',z)) \\
 & & \quad\ \ \text{in } x\ f\ h \text{ end} \\
 & & \\
T_H\ (h\ \textbf{h}\ f) & \hat{=} & (\nu f')(f \overset{susp}{\Leftarrow} \text{let } v = \text{get } f' \text{ in put}(f',v);\ \text{v} \mid f'\ \textbf{b -} \mid h \Leftarrow \lambda z.\text{put}(f',z)) \\
 & & \\
T_H(h\ \textbf{h}\ \bullet) & \hat{=} & h \overset{susp}{\Leftarrow} h \\
 & & \\
T_H(p) & \hat{=} & \text{homomorphically wrt. the term structure}
\end{array}
$$

Figure 3.11: Translation T_H : λ (fch) $\rightarrow$ λ(fcb)

In the encoding of a handled future $h\ \mathbf{h}\ f$ we use an empty buffer, a lazy thread and the buffer operations. The use of an empty buffer $f'\ \mathbf{b}$ - together with the operation put(f', v) simulates a binding of the future f to the buffer f'. The expression **put** (f', z) is a cyclic action and provides that the buffer stays filled. A lazy thread is used to express that the handle is not used.

In the encoding of a used handle $h\ \mathbf{h}\ \bullet$ we use a lazy thread as well and in consider that evaluation of $h \overset{susp}{\Leftarrow} h$ is not successful, exactly as an attempt to use a used handle.

The implementation of the constant **handle** is similar to the implementation of a hadled future. This implemantation spawns a new empty buffer, which will be filled using the put operations as well.

4 An Interpreter for the Calculus λ(fchb)

In this chapter we describe a functional implementation of an interpreter for the calculus λ(fchb) and call our interpreter *Ifchb*. First, we discuss the functional specification and design of our interpreter program and give further the detailed description of its implementation usung the functional programming language Haskell. Our implementing approach is based on the techniques for implementation of functional programming languages described in [JL92], [SS207] and [Tra91]. Before we start with the implementation we give an overview about interpreters and interpreter design.

4.1 What is an interpreter?

In order that programs in a certain programming language may be executed on a computer, that language must be made availiable , or implemented, on the particular type of computer. This may be achieved in various ways. Implementations are mainly divided into interpreters and compilers.

What is an interpreter? An interpreter is a computer program that performs the instructions of another program written in source code. The execution of the instructions succeeds one by one and without transformation of the source code into a machine code, as compilers do, but directly or after its transformation into a less complex intermediate code, for example, an abstract syntax tree or a parse tree. In both cases the execution succeeds in run-time [WM95].

Interpreters have their advantages and disadvantages: on the one hand, they run more slowly and need more memory as compared to compilered languages; but on the other hand, interpreted languages provide excellent debugging support, they are much easier to build and their code is smaller. Platform independence and high degree of security make interpreted languages suitable for Internet and web-based applications [WM95].

The most common interpreted languages are Java, Python, Perl, PHP, Lisp, IDL, Basic and Cobol.

4.2 Interpreter design

As we already mentioned, there are interpreters that execute a program directly, we call such interpreters *pure* interpreters. But most interpreters execute a program after some *code optimization.*

The code optimization includes lexical, syntax and semantic analyses, in the way, they are used in compiler design. Figure 4.1 represents the phases of the interpreter design. The detailed description of all phases we give in the next sections.

4.3 Implementation of Ifchb

First of all we should define a functional specification of our software. We implement an interpreter for the calculus λ(fchb) introduced in Chapter 3. Our interpreter Ifchb has to be able to encode concurrency primitives of the calculus, provide translations described in Section 3.5 and evaluate a program. We are going to implement our software in already described functional programming language Haskell and explain its implementation step by step. The complete program code of Ifchb can be found under the following link: http://www.ki.informatik.uni-frankfurt.de/diplom/programme/owenge.tar.gz.

Our interpreter software consists of eleven modules: Syntax, Lexer, Parser, SemAnalysis, Transformation, Evaluation, ReductionRules, EncodingFCH, EncodingFCB, Decoding and Inrepreter. These modules corresponds with certain phases of the interpreter design shown in Figure 4.2.

4.3.1 Module Syntax

First of all, we need to define data structures for the Ifchb in accordance with the calculus λ(fchb) described in Chapter 3.

`data Proc` is the data structures for fchb-prosesses. The Ifchb programm can have one or more process. We decode these processes as follows:

`data Proc =`

`  Parall Proc Proc`	parallel processes $(p1 \| \ p2)$
`| New Var Proc`	new process operator $(\nu x\ )\ p$
`| Cell Var Exp`	reference cell x **c** v
`| Thread Var Exp`	thread $x \Leftarrow e$
`| Lazythread Var Exp`	lazy thread $x \overset{susp}{\Leftarrow} e$
`| Handle Var Var`	handle $y\ \mathbf{h}\ x$
`| Usedhandle Var`	used handle $y\ \mathbf{h}\ \bullet$
`| Buffer Var Exp`	buffer $x\ \mathbf{b}\ v$

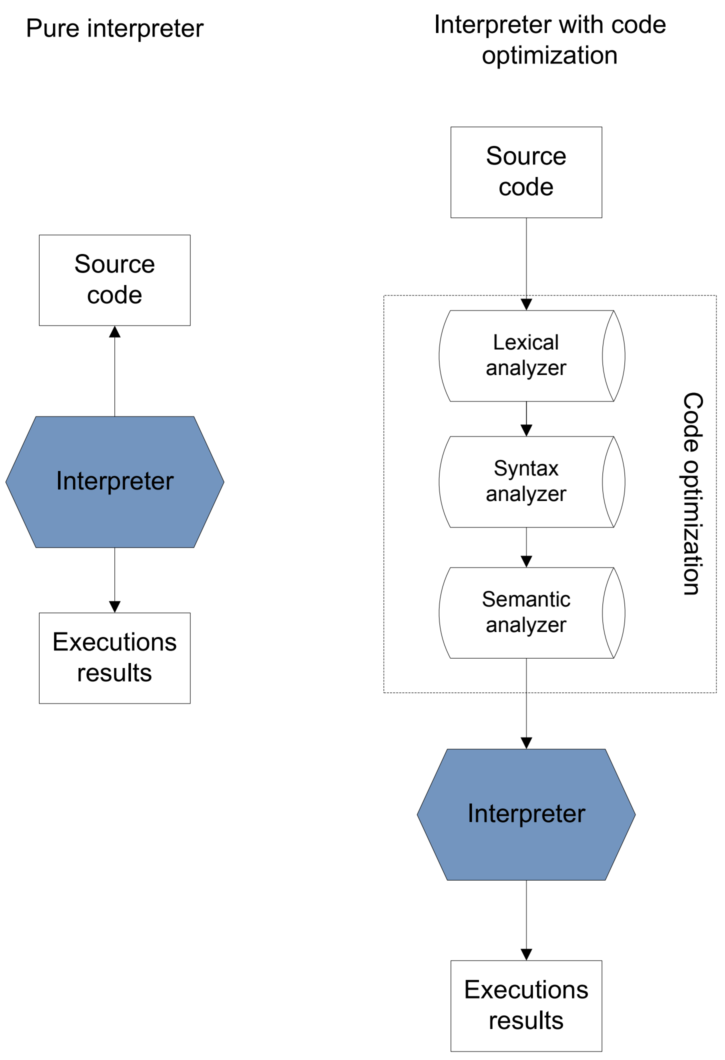


Figure 4.1: Interpreter design

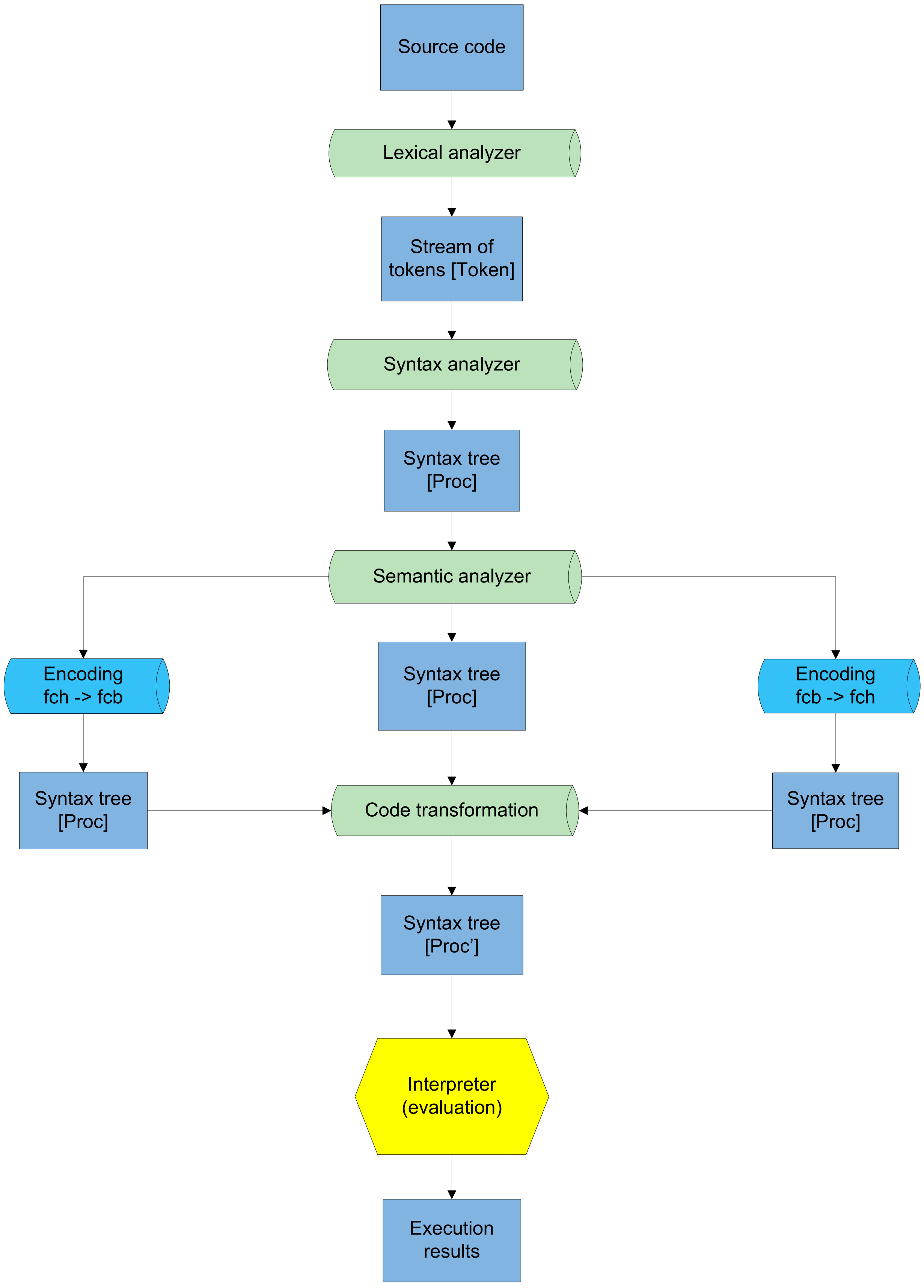


Figure 4.2: Design of Ifchb

`| Emptybuffer Var`	empty buffer x **b** $-$

In the same way we define `data Exp` and `data Alts`, the data structures for fchb-expressions and case alternatives:

`data Exp =`

`V Var`	variable
`| CCell`	constant **cell**
`| CThread`	constant **thread**
`| CHandle`	constant **handle**
`| CLazy`	constant **lazy**
`| CUnit`	constant **unit**
`| CBuffer`	constant **buffer**
`| CGet`	constant **get**
`| Lambda Var (Exp)`	$\lambda x.e$
`| App Exp Exp`	application $e_1\ e_2$
`| Exch Exp Exp`	**exch** (e_1, e_2)
`| Put Exp Exp`	**put** $(e_1, e_2)x$
`| Constr Int Int Int [Exp]`	constructor
`| Case Exp [Alts]`	**case** e **of** alternatives

`data Alts =`

`Alt Int Int Int [Var] Exp`	case alternative (constructor)

As you see, we decoded the constructors and case alternatives into the form:

`Constr Int Int Int [Exp]` or `Alt Int Int Int [Var] Exp`

The first two integers `Int` provide uniquely identification of constructors or alertnatives: the first integer implies the type, the second integer is a number to distinguish constructs or alternatives of the same type. The third integer implies their arity, i.e. it tells how many argumets they take. This form is easily manageable and very efficient for the syntax analysis we realize in the `module Parser`.

Finally, we define the type for identifiers:

```
type Var = String
```

The Ifchb data structure is now finished and we can start with the code analysis.

4.3.2 Module Lexer

The `module Lexer` carries out the lexical analysis of our source program. It reads the source program in the form of character string once only from left to right and converts it in linear-time into a sequence of lexical units (tokens). There are two basic techniques for recognizing tokens: using a finite state atomaton or constructing an ad hoc lexical analyzer by hand. Both techniques are based on the specification of the tokens through regular expressions [GBJL00].

Our lexical analyzer is written by hand and recognizes six classes of tokens: identifier, constant, keyword, string-literal, operator and punctuator. Such charakters as `<eb>`, `<n>`, `/t`, `<s=` are string-literal tokens, and `(`, `)`, `{`, `}` and `,` (comma) are punctuator tokens.

Figure 4.3 shows the token data structure. Each token has three arguments. The first argument is a token name, the second argument is a position (line, column) of a token in the input program. The last argument represents an integer or a name of identifier.

The marking of the position is very helpful for troubleshooting, therefore we also define functions `getPos` and `printToken`. The function `getPos` returns the position of a token and the function `printToken` converts a token into its source code string:

```
getPos :: Token -> Position
getPos (TokInt x y) = x
getPos (TokVar x y) = x
getPos (TokTrue x)=x
getPos (TokSusThread x) = x
getPos (TokNew x) = x
getPos (TokCell x) = x
getPos (TokHandle x) = x
getPos (TokUsedHandle x) = x
getPos (Error x) = x ...

printToken :: Token -> String
printToken (TokInt x y) = show y
printToken (TokVar x y) = y
printToken (TokTrue _) = "True"
printToken (TokSusThread _) = "<s="
printToken (TokNew _) = "<n>"
printToken (TokCell _) = "<c>"
printToken (TokHandle _) = "<h>"
printToken (TokUsedHandle _) = "<_>"
printToken (Error _ ) = ""  ...
```

For example, consider the following function calls:

```
*Lexer> getPos (TokTrue (1,6))
(1,6)
*Lexer> printToken (TokTrue (1,6))
"True"
```

```
 data Token = TokInt         Position   Integer  -- integer
            | TokVar         Position   String   -- identifier
            | TokTrue        Position            -- True
            | TokFalse       Position            -- False
            | TokNil         Position            -- []
            | TokCons        Position            -- :
            | TokCCell       Position            -- cell
            | TokCThread     Position            -- thread
            | TokCHandle     Position            -- handle
            | TokCLazy       Position            -- lazy
            | TokCUnit       Position            -- unit
            | TokCBuffer     Position            -- buffer
            | TokCGet        Position            -- get
            | TokExch        Position            -- exch
            | TokPut         Position            -- put
            | TokCase        Position            -- case
            | TokOf          Position            -- of
            | TokComma       Position            -- ,
            | TokCBOpen      Position            -- {
            | TokCBClose     Position            -- }
            | TokArrow       Position            -- ->
            | TokLam         Position            -- \\   (lambda)
            | TokBOpen       Position            -- (
            | TokBClose      Position            -- )
            | TokThread      Position            -- <=   (thread)
            | TokSusThread   Position            -- <s=  (lazy thread)
            | TokNew         Position            -- <n>  (new)
            | TokCell        Position            -- <c>  (cell)
            | TokHandle      Position            -- <h>  (handle)
            | TokUsedHandle  Position            -- <_>  (used handle)
            | TokBuffer      Position            -- <b>  (buffer)
            | TokEmptyBuffer Position            -- <eb> (empty buffer)
            | TokParall      Position            -- |
            | Error          Position
               deriving (Eq,Show)

type Position = (Int, Int)                      -- (line, column)
```

Figure 4.3: Token data structure

The function `lexer` is the top-level function in this module. Using Haskell pattern matching it convert the source program into the list of tokens and their positions.

```
lexer :: String -> [Token]
```

The subfunction `lexer1` starts with the position (`line` = 1, `column` = 1) and recognizes string-literal tokens and punctuator tokens in the input string.

```
lexer string = lexer1 1 1 string
```

The whitespace characters, such as blanks, tabs, newlines will be skipped during lexical analysis:

```
lexer1 l c []=[]
lexer1 l c ('\n':xs)= lexer1 (l+1) 0 xs
lexer1 l c ('\t':xs)= lexer1 l (c+1) xs
lexer1 l c ('\r':xs)= lexer1 (l+1) 0 xs
```

The other characters will be divided into tokens:

```
lexer1 l c ('[':(']':xs)) = TokNil (l,c):lexer1 l(c+2) xs
lexer1 l c (':':xs) = TokCons (l,c):lexer1 l(c+1) xs
lexer1 l c (',':xs) = TokComma (l,c):lexer1 l(c+1) xs
lexer1 l c ('{':xs) = TokCBOpen (l,c):lexer1 l(c+1) xs
lexer1 l c ('}':xs) = TokCBClose (l,c):lexer1 l(c+1) xs
lexer1 l c ('-':('>':xs)) = TokArrow (l,c):lexer1 l(c+2) xs
lexer1 l c ('<':('=':xs)) = TokThread (l,c):lexer1 l(c+2) xs
lexer1 l c ('<':('s':('=':xs))) = TokSusThread (l,c):lexer1 l(c+3) xs
lexer1 l c ('<':('n':('>':xs))) = TokNew (l,c):lexer1 l(c+3) xs
lexer1 l c ('<':('c':('>':xs))) = TokCell (l,c):lexer1 l(c+3) xs ...
```

The subfunctions `lexVar` and `lexNum` are acting for recognizing of keywords, alphanumeric identifiers and integer constants:

```
lexer1 l c (x:xs)
    | isSpace x = lexer1 l(c+1) xs
    | isAlpha x = lexVar l c (x:xs)
    | isDigit x = lexNum l c (x:xs)
    | otherwise = error(err_print (Error (l,c))++ show x)

lexNum l c xs = TokInt (l,c)(read num):lexer1 l(c+ length (num)) rest
    where (num,rest) = span isDigit xs

lexVar l c xs =
  case span isAlphaDigit xs of
     ("False",rest)   -> TokFalse (l,c):lexer1 l(c+5) rest
     ("True",rest)    -> TokTrue (l,c):lexer1 l(c+4) rest
     ("cell",rest)    -> TokCCell (l,c):lexer1 l(c+4) rest
     ("thread",rest)  -> TokCThread (l,c):lexer1 l(c+6) rest
     ("lazy",rest)    -> TokCLazy (l,c):lexer1 l(c+4) rest
     ("unit",rest)    -> TokCUnit (l,c):lexer1 l(c+4) rest
     ("exch",rest)    -> TokExch (l,c):lexer1 l(c+4) rest
     ("case",rest)    -> TokCase (l,c):lexer1 l(c+4) rest
```

```
      ...
   where isAlphaDigit x = isAlpha x || isDigit x
```

The subfunction `noComments` deletes all comments beginning with `--`.

```
noComments []=[]
noComments (x:xs)= if (x==’\n’) then xs else noComments xs
```

The only error can occur if any character does not match the specification of the tokens. In this case the function `errorPrint` returns a message with the unknown character and its position in the token stream.

```
errorPrint x = "\n *** Error during lexing at the position " ++
               show(getPos x) ++ "\n unknown character:"
```

The following examples show successfull and erroneous lexical analyses:

```
*Lexer> lexer "<n> x ( <n> y ((x <= y x | y <= \\ z -> z)))"

[TokNew (1,1),TokVar (1,5) "x",TokBOpen (1,7),TokNew (1,9),
TokVar (1,13) "y", TokBOpen (1,15),TokBOpen (1,16),TokVar (1,17)
"x",TokThread (1,19),TokVar (1,22)"y",TokVar (1,24) "x",
TokParall (1,26),TokVar (1,28) "y",TokThread (1,30),TokLam
(1,33),TokVar (1,35) "z",TokArrow (1,37),TokVar (1,40) "z",
TokBClose (1,41),TokBClose (1,42),TokBClose (1,43)]

*Lexer> lexer "<n> x ( <n> y ((x <=> y x | y <= \\ z -> z)))"

[TokNew (1,1),TokVar (1,5) "x",TokBOpen (1,7),TokNew (1,9),
TokVar (1,13) "y", TokBOpen (1,15),TokBOpen (1,16),TokVar (1,17)
 "x",TokThread (1,19)*** Exception:
 *** Error during lexing at the position (1,21)
 unknown character:’>’
```

The `module Lexer` exports the function `lexer` for its further using by parsing.

4.3.3 Module Parser

In the `module Parser` we define the specification of our parser for its automatically generation with `Happy`[1], a parse generator system for Haskell. The generated parser is a bottom-up (or shift-reduce) parser. It implements the syntax analysis of our interpreter. The parser receives a sequence of tokens, generated by the lexer, as input, runs through the sequence from left to right once only and recognizes in this sequence the syntax structure described in the `module Syntax`. Output of our parser is a syntax tree of type `Proc`

[1]haskell.org/happy

We describe now the `module Parser`, a `Happy` file for further parser generation. Our `Happy` file consits of four parts.

Part 1. The first part is a Haskell declaration of a module. This declaration is enclosed in curly braces as all occurrences of Haskell code in the `module Parser`.

```
 {module Parser
(
 parser,
 parse
 )
where
import Lexer
import Syntax
import Char
import Data.List
}
```

Part 2. In the second part we make some declarations.

`%name` declares a name of the generating parser function:

```
%name parse
```

`%tokentype` is a type of tokens, which the parser becomes from the lexer:

```
%tokentype { Token }
```

`%token` declares all the possible tokens. The symbols on the left are the terminals of our context-free grammar, and the symbols on the the right are the tokens, produced by the lexer in the Haskell code. The `$$` symbol is a placeholder and represents a value of a token. The parser receives a stream of tokens and matches each of them with the terminals on the left side:

```
%token
  int       {TokInt _ $$}
  var       {TokVar _ $$}
  'False'   {TokFalse _}
  'True'    {TokTrue _}
  '[]'      {TokNil _}
  ':'       {TokCons _}
  'case'    {TokCase _}
  'of'      {TokOf _}
  'cell'    {TokCCell _}
  'thread'  {TokCThread _}
  'handle'  {TokCHandle _}
  'lazy'    {TokCLazy _}
  'unit'    {TokCUnit _}
  'buffer'  {TokCBuffer _}
  'get'     {TokCGet _}
  'exch'    {TokExch _}
  'put'     {TokPut _ }
  '<='      {TokThread _}
  '<s='     {TokSusThread _}
  '<n>'     {TokNew _}
  '<c>'     {TokCell _}
  '<h>'     {TokHandle _}
  '<b>'     {TokBuffer _}
  '<eb>'    {TokEmptyBuffer _}
  '<_>'     {TokUsedHandle _}
  '|'       {TokParall _}
  ','       {TokComma _}
  '{'       {TokCBOpen _}
  '}'       {TokCBClose _}
  '('       {TokBOpen _}
  ')'       {TokBClose _}
  '->'      {TokArrow _}
  '\\'      {TokLam _}
  '='       {TokEq _}
  ''        {Error _}
```

Our context-free grammar is ambiguous, hence we specify precedences of the operators to avoid shift/reduce conflicts during parsing.

The `%left`, `%right` or `%nonassoc` declares tokens to be left, right or non-associative respectively:

```
%right '->'
%right ':'
%left '|'
%nonassoc ','
```

For example, the processes $p_1| \; p_2 \mid p_3$ are to be parsed as $(p_1| \; p_2) \mid p_3$, as $\mid$ is left-associative.

Part 3. In the next part we define all the production rules of our context-free grammar:

```
%%
Proc :: {Proc}
Proc  : Comp '|' Proc                 {Parall $1 $3}
      | Comp                          {$1}
      | '('Proc')'                    {$2}

Comp  : '<n>' var '('Proc')'          {New (mkV $2) $4}
      | var '<c>' Exp                 {Cell (mkV $1) (chkVal $2 $3)}
      | var '<='  Exp                       {Thread (mkV $1) $3}
      | var '<h>' var                       {Handle (mkV $1) (mkV $3)}
      | var '<_>'                           {Usedhandle (mkV $1)}
      | var '<s=' Exp                       {Lazythread (mkV $1) $3}
      | var '<b>' Exp                       {Buffer (mkV $1) (chkVal $2 $3)}
      | var '<eb>'                          {Emptybuffer (mkV $1)}


Exp : Exp AExp                              {App $1 $2}
    | AExp                                  {$1}

AExp: var                                   {V (mkV $1)}
    | int                                   {mkInt $1}
    | '\\' var '->' AExp                    {Lambda (mkV $2) $4}
    | 'exch' '('AExp','AExp')'              {Exch  $3 $5}
    | 'put' '('AExp','AExp')'               {Put  $3 $5}
    | 'case' AExp 'of' '{'Alts'}'           {Case $2 (chkAlts $5)}
    | '('AExp','AExp')'                     {pair $2 $4}
    | '('AExp','AExp','AExp')'              {tupel3 $2 $4 $6}
    | '('AExp','AExp','AExp','AExp')'       {tupel4 $2 $4 $6 $8}
    | '(' AExp ':' AExp ')'                 {cons $2 $4}
    | 'False'                               {false}
    | 'True'                                {true}
    | '[]'                                  {nil}
    | Constant                              {$1}
    | '('Exp')'                             {$2}

Constant : 'cell'                           {CCell}
         | 'thread'                         {CThread}
         | 'handle'                         {CHandle}
         | 'lazy'                           {CLazy}
         | 'unit'                           {CUnit}
         | 'buffer'                         {CBuffer}
         | 'get'                            {CGet}

Alts : Alt                                  {[$1]}
     | Alt','Alts                           {$1:$3}

Alt : Pat'->'Exp                            {$1 $3}

Pat  : 'True'                           {altTrue}
     | 'False'                          {altFalse}
     | '('Pat')'                        {$2}
     | var':'var                        {chkAltCons (mkV $1)(mkV $3)}
     | '[]'                             {altNil}
     | '('var','var')'             {chkAltPair (mkV $2)(mkV $4)}
     | '('var','var','var')'       {chkAlt3Tup (mkV $2)(mkV $4)(mkV $6)}
     |'('var','var','var','var')'  {chkAlt4Tup (mkV $2)(mkV $4)(mkV $6)(mkV $8)}
```

Each production consists of a non-terminal symbol, followed by a colon and one or more expansions, separated by |, on the left and a relevant Haskell code on the right.

This Haskell code can also include actions (e.g. functions `mkVar`, `checkVal`, `pair`, etc.) for the syntax check of expressions. We describe these actions in the last part of the `Happy` file.

For example, `AExp: '\\' var '->' AExp   {Lambda (mkV $2) $4}` means, that by the derivation of the production rule `AExp: '\\' var '->' AExp` the following Haskell code is to be created:

`Lambda`	`(mkV $2)`	`$4`
`Lambda`	result of the action `mkV` with the second component (terminal `var`)	the forth component `(AExp)`

Part 4. The forth and the last part of the `Happy` file is a Haskell code with definitions of functions and actions.

Firstly, we define here the function `happyError`. This function reports parse errors and shows erroneous stream of tokens.

```
{
happyError :: [Token] -> a
happyError xs = error $ "parse error!" ++ show xs
```

Further we specify the actions we need for syntax proof and for creating parser output of type `Proc`. Figure 4.4 shows all actions and their meanings.

Action	Meaning
`mkV`	identity function for variables
mkInt	converts integers into binary code, using one and zero constructors
chkVal	tests wether an expression is a value
chkAlts	proves wether case-alternatives are valid and exhaustive
chkAltPair	proves wether all variables in the pair patterns are varied
chkAltCons	proves wether all variables in the list patterns are varied
chkAlt3Tupel	proves wether all variables in the 3-tupel patterns are varied
chkAlt4Tupel	proves wether all variables in the 4-tupel patterns are varied

Figure 4.4: Actions of the module Parse

Figures 4.4 and 4.5 show the decoding of all constructors and case-alternatives accordingly to the `module Syntax`:

`Constr Int Int Int [Exp]` or `Alt Int Int Int [Var] Exp`

As we already described, the first integer `Int` implies the type of a constructor or alternative. They may be of the type `List` (1), `Bool` (2), `Pair` (3), `3-tupel` (4), `4-tupel` (5) or special type for binary code (6). The second integer we need to distinguish between the constructors and alternatives of the same type. The third integer implies the arity to tell how many arguments are allowed.

For example, the constructor `[]` (Nil) will be decoded as `Constr 1 1 0 []`, that means the constructor is of type `List`, has number 1 and arity 0 (= the list of arguments is empty `[]`).

Using these three intergers it is very simple to make the type check in the case-expressions.

```
nil              = Constr 1 1 0 []
cons a b         = Constr 1 2 2 [a,b]
true             = Constr 2 1 0 []
false            = Constr 2 2 0 []
pair a b         = Constr 3 1 2 [a,b]
tupel3 a b c     = Constr 4 1 3 [a,b,c]
tupel4 a b c d   = Constr 5 1 4 [a,b,c,d]
zero             = Constr 6 1 0 []
one              = Constr 6 2 0 []
npair a b        = Constr 6 3 2 [a,b]
```

Figure 4.5: Decoding of constructors

```
altNil              = Alt 1 1 0 []
altCons a b         = Alt 1 2 2 [a,b]
altTrue             = Alt 2 1 0 []
altFalse            = Alt 2 2 0 []
altPair a b         = Alt 3 1 2 [a,b]
alt3tupel a b c     = Alt 4 1 3 [a,b,c]
alt4tupel a b c d   = Alt 5 1 4 [a,b,c,d]
altZero             = Alt 6 1 0 []
altOne              = Alt 6 2 0 []
altNpair a b        = Alt 6 3 2 [a,b]
```

Figure 4.6: Decoding of case-alternatives

Now we define the top-level function for our module. The function `parser` becomes a string as input, calls our parser `parse` and the lexer function `lexer` and returns a syntax tree of type `Proc`.

```
parser :: String -> Proc
parse :: [Token] -> Proc
parser = (parse . lexer)
```

```
}
```

We save the Happy file as `Parser.y` and generate our parser file `Parser.hs`.

The following examples show successful and unsuccessful parsing.

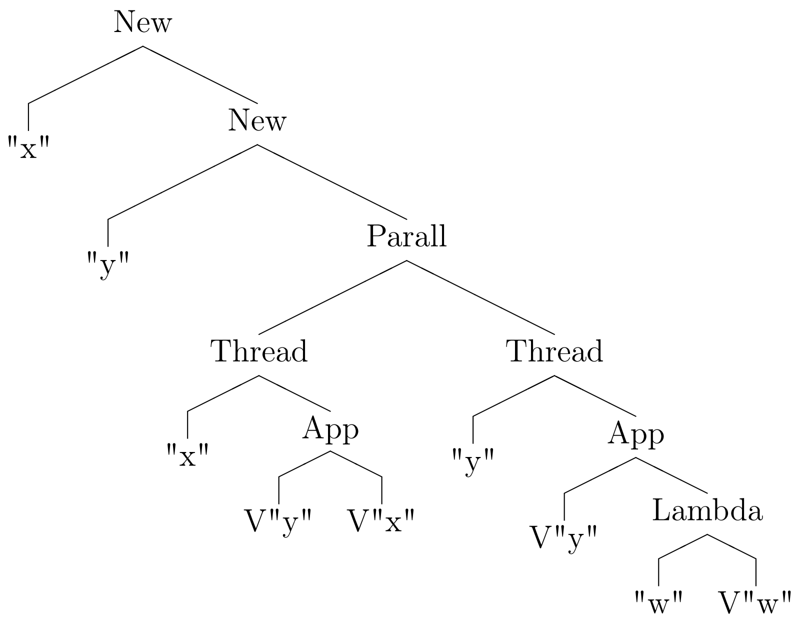
```
*Parser> parser "<n> x ( <n> y ((x <= y x | y <= \\ w -> w)))"

New "x" (New "y" (Parall (Thread "x" (App (V "y") (V "x")))
(Thread "y" (Lambda "w" (V "w")))))

*Parser> parser "<n> x ( <n> y ((x <= y True | y <= \\ z ->)))"

*** Exception: parse error![TokBClose (1,42),TokBClose (1,43),
TokBClose (1,44)]
```

And as syntax tree:



4.3.4 Module SemAnalysis

The parsing has brought our program into a syntactically valid form and now we have to check whether this form have a sence. So, we move forward to semantic analysis.

Semantic analysis is the last part of the code analysis and the last chance for our software to weed incorrect programs out. A program called to be semantically valid, if all variables, functions, classes, etc. are exactly defined; expressions and variables are used in accordance with the type system and so on.

In the module SemAnalysis we realize the following semantic checks:

- we prove wether the program is well-formed

- we check the new name operator (ν) - bindings
- we check the λ - bindings
- and make α - renaming for bounded variables.

As the IfFCHB is untyped, we do not need the type checking in this code analysis phase.

We start now with the function `isWellformed`, our well-formedness checker. Our program is well-formed, if there are no variables that are introduced in more than one process (see Chapter 3). For example, a program x **b** v | x **c** v is not well-formed, as the variable x occurs twice in the subprocesses.

The function `isWellformed` assembles all the variables from the processes into the list (`varList`) and checks using the function `nub` wether the variables occur in the list more than once. The output of the function `isWellformed` is `True`, if the program is well-formed, or an error message with the list of repeated variables:

```
isWellformed :: Proc -> Bool
isWellformed s = if (varList s == nub (varList s)) then True
                 else error ("is not wellformed " ++ show (varList s))

varList x = case x of
      (Parall a b)     -> (varList a) ++ (varList b);
      (Handle a b)     -> [a] ++ [b];
      (Cell a b)       -> [a];
      (Thread a b)     -> [a];
      (Usedhandle a)   -> [a];
      (Lazythread a b) -> [a];
      (Buffer a b)     -> [a];
      (Emptybuffer a)  -> [a]
      (New a b)        -> if is_wellformed b then []
            else error ("is not wellformed " ++ show (varList b))
```

The function `renameBoundedProc` checks the ν - bindings in processes. All variables in the processs have to be bounded with the new name operator ν. The function `renameNew` searches in the processes for variables and renames the bounded variables using the list of free variables `freshVariables`:

```
freshVariables = ["newVar" ++show x|x<-[1..500]]

renameBoundedProc :: Proc -> Proc
renameBoundedProc x  = fst (renameNew x fv1 [])

renameNew :: Proc -> [Var] -> [(Var,Var)] ->(Proc, [Var])
renameNew (Usedhandle a) y z = (Usedhandle (renameVar2 a z), y)
renameNew (Handle a b) y z = (Handle (renameVar2 a z) (renameVar2 b z), y)
renameNew (Buffer a b) y z = (Buffer (renameVar2 a z) (renameBoundedExp b z), y)
renameNew (Thread a b) y z = (Thread (renameVar2 a z) (renameBoundedExp b z), y)..
```

The function `renameBoundedExp` is called for the semantic analysis in expressions. Analogically to `renameBoundedProc`, it checks the λ - bindings in expressions and makes α - renaming using the function `renameLam` and the list of free variables `freshVariables`:

```
renameBoundedExp :: Exp -> [(Var, Var)] -> Exp
renameBoundedExp x mappings = fst(renameLam x fv mappings)

renameLam :: Exp -> [Var] -> [(Var,Var)] ->(Exp, [Var])
renameLam (V x) y z = (V (renameVar x z), y)
renameLam (CCell) y z = (CCell, y)
renameLam (CThread) y z = (CThread, y)...
```

The following examples explain our semantic analysis:

```
 *SemAnalysis> isWellformed parser "<n>x(<n>y(x<=y|y<=buffer))"
 True

 *SemAnalysis> isWellformed parser "<n>x(<n>y(y<=y|y<=buffer))"
 *** Exception: is not well-formed ["y","y"]

 *SemAnalysis> renameBoundedProc parser "<n>x(<n>y(x<=y|y<=\\w->w))"
 New "_newVar1" (New "_newVar2" (Parall (Thread "_newVar1" (V "_newVar2"))
 (Thread "_newVar2" (Lambda "_internal1" (V "_internal1")))))

 *SemAnalysis> renameBoundedProc parser "<n>x(<n>w(<n>y(x<=y|s<=\\s->s)))"
 New "_newVar1" (New "_newVar2" (New "_newVar3" (Parall (Thread "_newVar1"
 (V "_newVar3")) (Thread "*** Exception: Variable s is not bound
```

4.3.5 Module EncodingBtoH

In the module EncodingBtoH we implement the translation $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch) introduced in Section 3.5. and semantic analysis of the calculus λ(fch).

We encode the buffers and empty buffers components as follows:

```
Buffer a b = <n>xp (<n>xg (<n>xs (<n>xh (a <= (xp,xg,xs,xh)
           | <n>h( <n>f (h <h> f | xp <c> f | xg <c> True
           | xs <c> b | xh <c> h))))))


Emptybuffer a = <n>xp (<n>xg (<n>xs (<n>xh (a <= (xp,xg,xs,xh)
              | <n>h( <n>f( <n>hs( <n>fs ( h <h> f | hs <h> fs
              | xp <c> True | xg <c> f | xs <c> fs | xh <c> h)))))
```

We let the encoded components to be parsed and pass them to the function `renameNewBufferToHandle` as arguments for semantic analysis. The function `renameNewBufferToHandle` has the same functionality as the function `renameNew` in the `module SemAnalysis`:

```
renameNewBufferToHandle :: Proc -> [Var] -> [(Var,Var)] ->(Proc, [Var])
```

By encoding of the buffer operations we use syntactic sugar given in Figure 3.9. The encoded expressions will be recurvie called with the function `renameBoundedExp` for semantic analysis. The encoded buffer operations could be found in Appendix.

4.3.6 Module EncodingHtoB

The module EncodingHtoB is the implementation of the translation T_H : λ (fch) $\rightarrow \lambda$(fcb) introduced in Section 3.5. and semantic analysis of the calculus λ(fcb).

We encode the handle and used handle components as follows:

```
Usesedhandle a  =  a <s= a


Handle a b  = <n>y1 ( b <s= (\\y2->((\\y3 -> put(y1,y2))y2))(get y1 )
              |y1 <eb> |a <= \\y4 -> put(y1,y4))
```

We parse the encoded components and pass them to the function `renameNewHandlesToBuffer` as arguments for semantic analysis. The function `renameNewHandlesToBuffer` has the same functionality as `renameNewBufferToHandle` and `renameNew`:

```
renameNewHandleToBuffer :: Proc -> [Var] -> [(Var,Var)] ->(Proc, [Var])
```

We encode the constant handle (see Appendix) analogical to the buffer operation and perform semantic analysis with the function `renameBoundedExp` as well.

4.3.7 Module Transformation

The module Transformation contains two transformations of the data structure:

Transformation 1. Considering the structural congruence of processes we transform our syntax tree into a new, easily manageable data structure. The new data structure has a form `Proc' [list of new variables] [list of processes]`. The use of the lists provides a faster search in processes, that makes the furter implementation of evaluation rules easier. Figure 4.7 shows the new data structure for processes, the data structure for expressions remains unchanged.

```
data Proc’ = Proc’[Var] [Comp]

data Comp = Cell’ Var Exp
          | Thread’ Var Exp
          | Handle’ Var Var
          | Usedhandle’ Var
          | Lazythread’ Var Exp
          | Buffer’ Var Exp
          | Emptybuffer’ Var
```

Figure 4.7: New data structure Proc’

The function `transfToProc’` carries out the transformation of the λ(fchb) processes after their well-formedness and semantic check (see `Module SemAnalysis`) into the data structure `Proc’`.

```
transfToProc’ :: Proc -> Proc’
transfToProc’ x = if is_wellformed x == True
                  then transf [] (renameBoundedProc x)
                     else error "not wellformed"
```

The subfunctions `transf` and `addParall` assemble new variables and the processes directly and bring them into the form `Proc’[Var][Comp]`, whereas non-terminals `New` and `Parall` are to be eliminated.

```
transf v x = case x of
             (Cell a b) -> Proc’ v [Cell’ a b];
             (Thread a b) -> Proc’ v [Thread’ a b];
             (Handle a b) -> Proc’ v [Handle’ a b];
             (New a b) -> let  new_v = v ++ [a] in transf new_v b;
             (Parall a b) -> let Proc’ nuvars procs =
                                 addParall (transf [] a) (transf [] b)
                             in  Proc’ (v++nuvars) procs ...
addParall (Proc’ a b) (Proc’ c d) = Proc’ (a++c) (b++d)
```

In the next example you can see the difference between both data structures for the source code `<n> x (<n> y (x <= y | <n> d (<n> s (<n> m (s <= True)))))`:

- as `Proc` data structure:

```
New "_newVar1" (New "_newVar2" (Parall (Thread "_newVar1" (V "_newVar2"))
(New "_newVar3" (New "_newVar4" (New "_newVar5" (Thread "_newVar4"
(Constr 2 1 0 []))))))))
```

- and definitely less complex as new `Proc’` data structure:

```
Proc’ ["_newVar1","_newVar2","_newVar3","_newVar4","_newVar5"]
[Thread’ "_newVar1" (V "_newVar2"),Thread’ "_newVar4" (Constr 2 1 0 [])]
```

For transformations in the calculi λ(fch) and λ(fcb) we use the functions `transfToProcBtoH'` and `transfToProcHtoB'` respectevely. They have the same functionality with `transfToProc'`:

After the transformation into `Proc'` our program is ready for execution.

Transformation 2. We define also the function `procPrimeToProc`, which provide the program tranformation from `Proc'` back to `Proc`. This transformation we will use later for software testing (see Chapter 5):

```
procPrimeToProc :: Proc' -> Proc
procPrimeToProc (Proc' vars comps) =
                foldr New (foldr1 Parall (map compToProc comps)) vars

compToProc :: Comp -> Proc
compToProc c = case c of
                 (Cell' v e) -> Cell v e
                 (Thread' v e) -> Thread v e
                 (Handle' v1 v2) -> Handle v1 v2
                 (Usedhandle' v) -> (Usedhandle v)
                 (Lazythread' v e) ->  (Lazythread v e)
                 (Buffer'  v e) -> (Buffer v e)
                 (Emptybuffer' v) -> (Emptybuffer v)
```

4.3.8 Module ReductionRules

The `module ReductionRules` together with the `module Evaluation` is the operational semantics of the calculus λ(fchb). We implement in this module reduction rules introduced in Chapter 3 (Figure 3.4).

We start with the evaluation of threads in a process (see evaluation contexts in Figure 3.3). The function `eval1` performes one-step reduction of the first componet of a process, if this component is an eager thread, otherwise returns `Nothing`. Four arguments of the function are the first component of a process, a list of new operators, a list of the rest componets of a process and a list of fresh variables.

The evaluation of a thread is only possible if its value is a non-variable expression:

```
eval1 :: Comp -> [Var] -> [Comp] -> [Var] -> Maybe (Proc', [Var])
eval1 p ns ps frv =
        case p of
           (Thread' x (V y)) -> Nothing
           (Thread' x e) ->
           let s = eval2 e ns ps frv
                in case s of
                Just (e', ns', ps', frv') ->
                Just (Proc' ns' ((Thread' x e'):ps'), frv');
                Nothing -> Nothing
           otherwise -> Nothing
```

The function `eval2` is called by `eval1` to provide the execution within a thread. The function `eval2` provides a pattern matching of all possible expressions by reduction rules, and evaluate them if a suitable pattern is found. `eval2` requires four arguments as well:

eval2 :: Exp -> [Var] -> [Comp] -> [Var] -> Maybe (Exp, [Var], [Comp], [Var]),

The arguments are: an expresion, a list of new operators, a list of the rest components and a list of fresh variables. `eval2` returns an evaluated expression or `Noithing` and a list of unused fresh variables.

For example, the pattern matching fot the rule (beta-CBV(ev) is defined as

```
eval2 (App (Lambda x e1) e2) ns ps frv
       | isValue e2 = case (betaRed e2 x e1 frv) of
        (e',frv') -> Just (e',ns,ps,frv')
       | otherwise  = case eval2 e2 ns ps frv of
         Just (e',ns,ps,frv') -> Just (App (Lambda x e1) e',ns,ps,frv')
         Nothing -> Nothing
```

The evaluation of (beta-CBV(ev))-rule invokes the function `betaRed` to perform a standard β-redution, we described in Chapter 2:

```
betaRed :: Exp -> Var -> Exp -> [Var] -> (Exp, [Var])
betaRed t v (V x) y
    | x == v    = let (t',zs) = renameVal t y [] in (t', zs)
    | otherwise = ((V x), y)
```

As example we consider the evaluation of a thread with the function `eval1`:

$(x \Leftarrow (\lambda z.z)\ \mathit{True})$

```
*Evaluation>
eval1 (Thread' "x" (App (Lambda "z" (V "z")) (Constr 2 1 0 [])))[][]frv
Just (x <= True,["frVar1","frVar2","frVar3",...
```

The implementation of reduction rules for creationg of new components are similar. We consider the pattern matching for the rule (THREAD.NEW(ev)) that spawns a new eager thread:

```
eval2 (App CThread e2) ns ps (z:frv)
        | isValue e2 = Just ((V z), z:ns, ((Thread' z (App e2 (V z))):ps), frv)
        | otherwise  = case eval2 e2 ns ps (z:frv) of
                           Just (e',ns,ps,frv') -> Just (App CThread e',ns,ps,frv')
                           Nothing -> Nothing
```

And an example of new thread creation:

$(x \Leftarrow (\text{thread } ((\lambda z.z)\ \mathit{False}))))$

```
*Evaluation> eval1 (Thread' "x" (App CThread (App
                  (Lambda "w" (V "w")) (Constr 2 2 0 []))))[][]frv
Just (x <= (thread False),["frVar1","frVar2"..
```

For implementig such reduction rules as (FUT.DEREF(ev)), (LAZY.TRIGGER(ev)), (HANDLE.BIND(ev)), (CELL.EXCH(ev)), (BUFF.PUT(ev)) and (BUFF.GET(ev)) we define a search function, which searches in components of a process for component identifiers and returns their value:

```
searchInProcs :: Var -> [Comp] -> Exp -> Maybe (Comp, Exp, [Comp])
searchInProcs a x e = siP a x x e
searchInProcs a (x:xs) e2 = case x of
    Thread' b e3 -> if (a==b) then
          if (isValue e3) then Just ((Thread' b e3), e3, (x:xs))
                 else Nothing
          else let p = xs ++ [x] in (searchInProcs a p e2);
    Lazythread' b e3 -> if a==b then
             Just ((Lazythread' b e3), e3, ((Thread' a e3):xs))
                       else let p = xs ++ [x] in (searchInProcs a p e2);
    Handle' b y -> if a==b then ...
```

For example, we examine the pattern matching for an application (App (V x) e2). There are some reduction rules matches this expression. The result of evaluation here depends on the result of the function searchInProcs: if the identifier is used in an eager thread, the reduction rule (FUT.DEREF(ev)) will be evaluated, if the identifier is used in a lazy thread, then the rule (LAZY.TRIGGER(ev)) and so on:

```
eval2 (App (V x) e2) ns ps (z:frv)
 | isValue e2 = let s = searchInProcs x ps e2
          in case s of
   Just ((Thread' a b), e',ps') -> let (e'',zs') = renameVal e' (z:frv) []
          in Just ((App e'' e2), ns, ps', zs');
   Just ((Lazythread' a b), e',ps') -> Just (App (V x) e2,ns,ps',z:frv);
   Just ((Handle' a b), e',ps') -> Just (e',ns,ps',(z:frv));
     Nothing -> Nothing
 | otherwise = case eval2 e2 ns ps frv of
   Just (e',ns,ps,frv') -> Just (App (V x) e',ns,ps,frv');
   Nothing -> Nothing
```

So, for example if searchInProcs find a an eager thread, then (FUT.DEREF(ev)) rule will be evaluated, if lazy thread then (LAZY.TRIGGER(ev)).

4.3.9 Module Evaluation

In the module Evaluation we implement fuctions that provide i-step reduction of a process and check wether a process is sucessful and convergent.

First we define the function `eval1Step`, which perform a one-step reduction of a process, i.e. the reduction of the first component:

```
eval1Step :: Proc' -> Maybe (Proc', [Var])
eval1Step (Proc' ns (p:ps)) = eval1 p ns ps frv
```

To enable a one-step reduction for all components, we implement the function `all1StepSuccessors`. This function is based on the Round-robin scheduling and evaluates each eager thread:

```
all1StepSuccessors :: Proc' -> [Maybe (Proc', [Var])]
all1StepSuccessors (Proc' ns ps)
           = map eval1Step ( map (Proc' ns) (everyThreadFirst [] ps))
               where
 everyThreadFirst _ [] = []
 everyThreadFirst qs (p:ps) = ((p:ps) ++ qs) : everyThreadFirst (qs++[p]) ps
```

But only one-step reduction is not enough for the evaluation, and we define the function `allEndPoints`, which returns all end-points of irreducible reduction successors:

```
allEndPoints :: Proc' -> [Proc']
allEndPoints p = nub $ map fst $ allEP_it [(p,frv)]

allEP_it :: [(Proc', [Var])] -> [(Proc', [Var])]
allEP_it [] = []
allEP_it p =
 let xs  =  filter (\(p',v) -> isIrreducible p') p
     ys  =  concatMap (map fromJust . filter isJust . all1StepSuccessorsWV)
     (filter (\(p',v) -> not $ isIrreducible p') p) in xs ++ (allEP_it ys)
```

For example, `allEndPoints` of the process

$y \ \mathbf{b}\ True \mid p1 \Leftarrow \mathbf{get}\ y \mid p2 \Leftarrow (\lambda w.\ \mathbf{put}(y, w))\ (\mathbf{get}\ y))$

are two sequences :

```
newVar1 <= True
| newVar3 <eb> -
| newVar2 <= ((\newVar4 -> put (newVar3,newVar4)) (get newVar3)

and

newVar1 <= True
| newVar3 <eb> -
| newVar2 <= unit
```

where the second one is successful.

As we described in Chapter 3, a successful process is a well-formed process and in each thread of this process the identifier is bound to a non-variable value, a cell, a lazy future, a handle or a buffer. For example, the process $x \Leftarrow y \mid y \Leftarrow z \mid z$ **b** $-$ is successful, and the processes $x \Leftarrow x$ (a black hole) and $x \Leftarrow (\lambda u, v.v)(y$ **unit**$) \mid$ $|y \Leftarrow (\lambda u, v.v)(x$ **unit**$)$ are prohibited.

As we have already checked the well-formedness in the module `SemAnylysis`, so we need now only to check wether the identifiers are bound.

The function `isSuccessfull` perfomes the successfulness test. It runs through the process and checks wether all identifiers in the threads (`componentSuccessfull`) are bound (`boundToSP`) to a non-variable value, a cell, a lazy thread, a handle, or a buffer (`findSuccessor`) and returns True, when all the subprocesses are successful. We check here only threads, because the other subprocesses - reference cells, buffers, handles and and lazy threads - are always successful.

```
isSuccessfull :: Proc' -> Bool
isSuccessfull (Proc' ns ps) =
           all (==True)(map(\p-> componentSuccessfull p ps) ps)

componentSuccessfull (Thread' v (V x)) ps = boundToSP x [x] ps
componentSuccessfull (Thread' v e) ps = isValue e
componentSuccessfull _ ps = True

boundToSP x visitedvars ps =
     let successor = findSuccessor x ps
     in  case successor of
      (Thread' v (V y)) -> if y `elem` visitedvars then False
                    else boundToSP y (x:y:visitedvars) ps
      (Thread' v e)     -> isValue e
      other             -> True

findSuccessor x (p:ps) =
  let v = case p of
          (Thread' y e) -> [y]
          (Cell' y e)   -> [y]
          (Handle' y1 y2) -> [y1,y2]
          (Usedhandle' y) -> [y]
          (Buffer' y e) -> [y]
          (Emptybuffer' y) -> [y]
          (Lazythread' y e) -> [y]
  in if x `elem` v then p else findSuccessor x ps
```

Now we define two important functions to examine convergence of processes. The function `mayConvergent` checks wether a process is may-convergent (see Section 3.5):

```
mayConvergent :: Proc' -> Bool
mayConvergent p =
   any (== True) [any (isSuccessful) (alliStepSucc i p) | i <- [1..]]
```

The function `totalMustConvergent` checks wether a process is total must-convergent (see Section 3.5 as well).

```
totalMustConvergent :: Proc' -> Bool
totalMustConvergent p = all isSuccessful (allEndPoints p)
```

Examples of evaluation of these functions we give in Chapter 5.

4.3.10 Module Interpreter

The `module Interpreter` contains three functions, or three interpreters for the calculi λ(fcb), λ(fch), and λ(fchb):

```
interpretFCHB :: String -> [T.Proc']
interpretFCHB str = allEndPoints $ T.transfToProc' $  parser $ str

interpretFCH :: String -> [T.Proc']
interpretFCH str  = allEndPoints $ T.transfToProcBtoH' $ parser $ str

interpretFCB :: String -> [T.Proc']
interpretFCB str  = allEndPoints $ T.transfToProcHtoB' $ parser $ str
```

Each function provides an execution of a program in the required calculus. These function we use later in Chapter 5 to compare results of encoded and non-encoded programs.

4.3.11 Module Decoding

Finally, the `module Decoding` provide our software with functions `show1` and `decodeInt` to decode expressions of a program into the original source code. The function `show1` converts expressions into a string and the function `decodeInt` converts integers into the decimal code:

```
*Decoding> show1 (Constr 6 3 2 [Constr 2 1 0 [],Constr 1 2 2
                 [Constr 6 2 0 [],Constr 1 2 2 [Constr 6 1 0 [],
                 Constr 1 2 2 [Constr 6 1 0 [],Constr 1 2 2
                 [Constr 6 2 0 [],Constr 1 2 2 [Constr 6 2 0 [],
                 Constr 1 1 0 []]]]]]])

"25"
```

We are now finished with the implementation and report about software test results in the next Chapter.

5 Testing

Although the translations T_B : λ (fcb) $\rightarrow \lambda$(fch) and T_H : λ (fch) $\rightarrow \lambda$(fcb) have been shown adequate in [SSNSS08], in this chapter we validate these results by some tests. In particular for some examplary processes we evaluate the process as well as the encoded process. For non-encoded process we encode the result of the evaluation. Finally, we compare both results.

We compare results of the λ(fchb) interpreter (Ifchb), the λ(fch) interpreter (Ifch) and the λ(fcb) interpreter (Ifcb). [1]

Test 1

We examine the translation T_B : λ (fcb) $\rightarrow \lambda$(fch). Consider the process p_1:

$p_1 = (\nu x)\ (\nu y)\ (y \Leftarrow \lambda w.\mathbf{put}(x, False)\ (\mathbf{get}\ x) \mid x\ \mathbf{b}\ True)$

Ifchb produces a non-empty buffer with the value `False`:

(1)

```
*Iterpreter> interpretFCHB p1
newVar2 <= unit
| newVar1 <b> False
```

Ifch produces a non-empty buffer with the value `False` as well. The buffer is represented as a 4-tupel `(newVar36,newVar37,newVar38,newVar39)`, where `newVar36`, `newVar37`, `newVar38` and `newVar39` are the names of reference cells.

If we fill the 4-tupel with contents of reference cells using handle bindings, we get `(frVar5,True,False,frVar6)`, where `frVar5` is a future, `True` means that the buffer is not empty, `False` is an actual content of the buffer, and `frVar` is the name of a handle that binds the future `frVar5`:

(2)

```
*Iterpreter> interpretFCH p1
newVar2 <= unit
| frVar1 <= True
| frVar2 <h> -
| frVar6 <h> frVar5
```

[1]For better performance we omit all new name operators in results.

```
| newVar1 <= (newVar36,newVar37,newVar38,newVar39)
| newVar41 <= True
| newVar36 <c> frVar5
| newVar37 <c> frVar1
| newVar38 <c> False
| newVar39 <c> frVar6
| frVar4 <h> frVar3
```

After removing some unused and successful components (i.e. garbage collecting some components) and removing some indirections (i.e. copying value True of frVar1 into the thread named newVar37) and adding a fresh renaming we get the (contextual) equivalent process:

(2')

```
  newVar1 <= (newVar3,newVar4,newVar5,newVar6)
| newVar2 <= unit
| newVar3 <c> newVar8
| newVar4 <c> True
| newVar5 <c> False
| newVar6 <c> newVar7
| newVar7 <h> newVar8
```

The result (3) is exactly (up to the ordering of the components) the result (2').

To compare our results more precisely we apply our encoding to the result (1) again to encode the buffer `newVar1 <b> False`:

(3)

```
*Iterpreter> map (T.transfToProcBtoH'. T.procPrimeToProc) (interpretFCHB p1)
newVar2 <= unit
| newVar1 <= (newVar3,newVar4,newVar5,newVar6)
| newVar7 <h> newVar8
| newVar3 <c> newVar8
| newVar4 <c> True
| newVar5 <c> False
| newVar6 <c> newVar7
```

Test result: We can see that the result (3) is exactly (up to the ordering of the components) the result (2'), i.e. the translation $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch) is equivalent.

Test 2

We consider another process p_2 and the translation T_B:

$p_2 = (\nu x)(\nu z)(x \Leftarrow z \mid z \Leftarrow$ **buffer unit**)

The evaluation of p_2 with **Ifchb** returns a new empty buffer:

(4)

```
*Iterpreter> interpretFCHB p2
newVar2 <= frVar1
| frVar1 <eb> -
| newVar1 <= newVar2
```

The evaluation with **Ifch** returns a new buffer as 4-tupel as well (for more details about invariants see the result (2) above):

(5)

```
*Iterpreter> interpretFCH p2
frVar9 <= (frVar5,frVar6,frVar7,frVar8)
| frVar8 <c> frVar2
| frVar7 <c> frVar3
| frVar6 <c> frVar1
| frVar5 <c> True
| frVar4 <h> frVar3
| frVar2 <h> frVar1
| newVar1 <= newVar2
| newVar2 <= frVar9
```

We apply **Ifch** interpreter to the result (4) to encode the empty buffer `frVar1 <eb> -` :

(6)

```
*Iterpreter> map (T.transfToProcBtoH'. T.procPrimeToProc) (interpretFCHB p2)
newVar3 <= newVar1
| newVar1 <= (newVar4,newVar5,newVar6,newVar7)
| newVar8 <h> newVar9
| newVar10 <h> newVar11
| newVar4 <c> True
| newVar5 <c> newVar9
| newVar6 <c> newVar11
| newVar7 <c> newVar8
| newVar2 <= newVar3
```

Test result: the results (5) and (6) are alpha-equivalent, i.e. the translation $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch) is equivalent.

Test 3

We examine now wether the translation T_B is convergence equivalent. Consider again the same processes p_1 and p_2 and the results of function calls `mayConvergent` and `totalMustConvergent`:

```
*Evaluation> mayConvergent $ transfToProc' $ parser $ p_1
```

```
True
*Evaluation> mayConvergent $ transfToProcBtoH' $ parser $ p_1
True
Evaluation> totalMustConvergent $ transfToProc' $ parser $ p_1
True
*Evaluation> totalMustConvergent $ transfToProcBtoH' $ parser $ p_1
True
```

and

```
*Evaluation> mayConvergent $ transfToProc' $ parser $ p_2
True
*Evaluation> mayConvergent $ transfToProcBtoH' $ parser $ p_2
True
Evaluation> totalMustConvergent $ transfToProc' $ parser $ p_2
True
*Evaluation> totalMustConvergent $ transfToProcBtoH' $ parser $ p_2
True
```

Test result: both interpretrs return the same results, and we can suppose that the translation T_B : λ (fcb) $\rightarrow \lambda$(fch) is convergence equivalent.

Test 4

Consider the process p_3 and the translation and T_H:

$p_3 = \nu x\ (\nu z\ (\nu v\ (\nu y(\ y\ \mathbf{b}\ True \mid x \Leftarrow \mathbf{get}\ y \mid z \Leftarrow (\lambda w.\ \mathbf{put}\ (y, w))\ (\mathbf{get}\ y)$

The interpreter **Ifchb** returns two possible end-points of this process:

(7)

```
newVar1 <= True
| newVar3 <eb> -
| newVar2 <= ((\newVar4 -> put (newVar3,newVar4)) (get newVar3)))))
```

and

(8)

```
newVar1 <= True
| newVar3 <eb> -
| newVar2 <= unit
```

The result (7) is a result , when the component $x \Leftarrow$ **get** y is evaluated first, and (7) is not successful. The result (8) is successful, if the evaluation begins with the thread $z \Leftarrow (\lambda w.\ \mathbf{put}\ (y, w))\ (\mathbf{get}\ y)$, that provides evaluation of both threads.

The results of convergece tests are again the same by both interpreters: the process p_3 is may-convergent, but not total must-convergent:

```
*Iterpreter>  mayConvergent $ T.transfToProc’ $ parser $ p_3
True
*Iterpreter>  totalMustConvergent $ T.transfToProc’ $ parser $ p_3
False

Iterpreter>  mayConvergent $ T.transfToProcBtoH’ $ parser $ p_3
True
*Iterpreter>  totalMustConvergent $ T.transfToProcBtoH’ $ parser $ p_3
False
```

Test result: We can again suppose that the translation $T_B : \lambda$ (fcb) $\rightarrow \lambda$(fch) is convergence equivalent.

Test 5

We examine the translation $T_H : \lambda$ (fch) $\rightarrow \lambda$(fcb). Consider the process p_4:

$p_4 = (\nu x)\ (x \Leftarrow \textbf{handle}\ (\lambda x.\lambda y.((\lambda w.(x\ True))(y\ (\lambda q.q)))))$

The evaluation of p_4 with **Ifchb** binds the futures `newVar1` and `frVar1` to theier values and returns a used handle after this binding.

(9)

```
*Iterpreter> interpretFCHB p_4
newVar1 <= unit
| frVar1 <= True
| frVar2
<h> -
```

The evaluation of p_4 with **Ifcb** returns the non-empty buffer `frVar5<b>(\frVar9->frVar9)`, which stays filled using the combination of **put** and **get** operations. This buffer simulates behavior of a handled future, it "binds"its value to its name. The lazy thread here controls that this binding occurs once only:

(10)

```
*Iterpreter> interpretFCB p_4
nnewVar1 <= True
| frVar6 <= (\frVar10 -> frVar10)
| frVar5 <b> (\frVar11 -> frVar11)
| frVar7 <= (\newVar15 -> put (frVar5,newVar15))
```

Test result: the results (9) and (10) we can conclude that they are equivalent.

Test 6

Now we examine wether the translation T_B is convergence equivalent. We consider again the processe p_4 and the functions `mayConvergent` and `totalMustConvergent`:

```
*Evaluation> mayConvergent $ transfToProc' $ parser $ p_4
True
*Evaluation> mayConvergent $ transfToProcBtoH' $ parser $ p_4
True
Evaluation> totalMustConvergent $ transfToProc' $ parser $ p_4
True
*Evaluation> totalMustConvergent $ transfToProcBtoH' $ parser $ p_4
True
```

Test result: the translation $T_H : \lambda$ (fch) $\rightarrow \lambda$(fcb) is convergence equivalent.

Test 7

Consider the process p_5:

$p_5 = (\nu x)\ (\nu y)\ (\nu z)\ (\nu a)\ (x\ \mathbf{h}\ y \mid z \Leftarrow x\ True\ | a \Leftarrow x\ False)$

The interpreter **Ifchb** returns again two sequinces, as thera two pssibilities for handle bindings `z <= x True` and `a <=  x False`:

```
newVar3 <= unit
| newVar2 <= True
| newVar1 <h> -
| newVar4 <= (newVar1 False)
```

and

```
newVar4 <= unit
| newVar2 <= False
| newVar1 <h> - |
```

The interpreter **Ifcb** returns an error message, as a process tries to use a handle once more.

Test result: The convergence test for the process p_5 fails, it is not may-convergent, as one handle is used twice in the process.

6 Conclusion and Further Work

In our master's thesis we introduced two important programming approaches in the modern programming: functional an concurrent programming. We gave an overview about Haskell, a functional language and described how sequential languages could become concurrent using concurrency primitives. We also gave an overview of the pure and some extended λ-calculi in Chapter 2.

In Chapter 3 we introduced another extended calculus λ(fchb), a concurrent lambda-calculus with handled futures, one-place buffers and reference cells and showed how concurrent primitives be mutually encoded within the same language, we also provide an approach of proving equivalence of concurrency primitives using translations of calculi.

In Chapter 4 we implemented an untyped interpreter for the calculus λ(fchb) as a technical validation for proving the equivalence of concurrent primitives and adequacy of the translations. Our implemented interpreter is able to execute mutual encoding of concurrent buffers and concurrent handles and examine adequacy of both translations $T_B : \lambda\ (\text{fcb}) \rightarrow \lambda(\text{fch})$ and $T_H : \lambda\ (\text{fch}) \rightarrow \lambda(\text{fcb})$.

In Chapter 5 we performed some tests to compare the results of translations and encodings. Using our implemented interpreter we tested wether encoded and non-encoded programs return the same result. These results were succesful and could be seen as partially confirmations of our theoretical arguments about equivalence of concurrent primitives and adequacy of translations.

We presume that we performed only some tests and can not be yet completely assured in the correctness of equivalence, but we have made the first steps. As further work we would suggest further testing of processes with our software, and an extending of our software with type checker and garbage collector.

Appendix

Encoding of the buffer operations

```
bufferToHandle =

Lambda "new" (App (Lambda "new2" (Case (V "new2")[Alt 3 1 2 ["h","f"]
(App (Lambda "new1" (Case (V "new1")[Alt 3 1 2 ["hs","fs"](App
(Lambda "putg" (App (Lambda "getg" (App (Lambda "stored" (App (Lambda "handler"
(App CThread (Lambda "new" (Constr 5 1 4 [V "putg",V "getg",V "stored",
V "handler"]))))(App CCell (V "h"))))(App CCell (V "fs")))) (App CCell (V "f"))))
(App CCell (Constr 2 1 0 []))))])) (App CHandle (Lambda "fs" (Lambda "hs"
(Constr 3 1 2 [V "hs",V"fs"]))))))]))(App CHandle (Lambda "f" (Lambda "h"
(Constr 3 1 2 [V "h",V "f"])))))

putToHandle x v =

App (Lambda "new4" (Case (V "new4") [Alt 3 1 2 ["x","v"] (Case (V "x")
[Alt 5 1 4 ["xp","xg","xs","xh"] (App (Lambda "new3" (Case (V "new3")
[Alt 3 1 2 ["h","f"] (App (Lambda "new2" (App (Exch (V "xh") (V "h"))
(Constr 2 1 0 []))) (App (Lambda "new1" (Exch (V "xs") (V "v"))) (Case
(Exch (V "xp")(V "f")) [Alt 2 1 0 [] (Constr 2 1 0 []),Alt 2 2 0 []
(Constr 2 1 0 [])]))))]))(App CHandle (Lambda "f" (Lambda "h"
(Constr 3 1 2 [V "h",V "f"]))))))])])) (Constr 3 1 2 [x, v])

getToHandle =

Lambda "new5" (Case (V "new5") [Alt 5 1 4 ["xp","xg","xs","xh"] (App
(Lambda "new4" (Case (V "new4") [Alt 3 1 2 ["h","f"] (App (Lambda "new3"
(Case (V "new3")[Alt 3 1 2 ["hs","fs"] (App (Lambda "new2" (App (Lambda "v"
(App (Lambda "new1" (V "v"))(App (Exch (V "xh") (V "h")) (Constr 2 1 0 []))))
(Exch (V "xs") (V "fs"))))(Case (Exch (V "xg") (V "f")) [Alt 2 1 0 []
(Constr 2 1 0 []),Alt 2 2 0 [](Constr 2 1 0 [])]))]))) (App CHandle
(Lambda "fs" (Lambda "hs" (Constr 3 1 2 [V "hs",V "fs"]))))))]))
(App CHandle (Lambda "f" (Lambda "h" (Constr 3 1 2 [V "h",V "f"])))))])
```

Encoding of the constant buffer

```
handleToBuffer y1 y2 y3 y4 y5 y6 y7 y8 =

Lambda y1 (App (Lambda y2 (App (Lambda y3 (App (Lambda y4 (App
(App (V y1) (V y3)) (V y4))) (App CThread (Lambda y5 (Lambda y6
(Put (V y2) (V y6))))))) (App CLazy (Lambda y5 (App (Lambda y7
(App (Lambda y8 (V y7)) (Put (V y2) (V y7)))) (App CGet (V y2))))))
(App CBuffer CUnit))
```

Bibliography

[Ali] *Alice Project, Programming Systems Lab, Saarland University, http://www.ps.uni-sb.de/alice/*

[BA06] Ben-Ari, M.: *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006

[Bar84] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984

[Bir98] Bird, R.: *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998

[BTL05] Bois, A. R. D.; Trinder, P. W.; Loidl, H.-W.: mHaskell: Mobile Computation in a Purely Functional Language. **In:** *J. UCS* 11 (2005), Nr. 7, S. 1234–1254

[GBJL00] Grune, D.; Bal, H. E.; Jacobs, C. J. H.; Langendoen, K. G.: *Modern Compiler Design*. John Wiley & Sons, LTD, 2000

[Han02] Hansen, P. B.: The invention of concurrent programming. (2002), S. 3–61. ISBN 0–387–95401–5

[Han04] Hankin, C.: *An Introduction to Lambda Calculi for Computer Scientists*. Bd. Volume 2. King's College Publications, 2004

[HPF00] Hudak, P.; Peterson, J.; Fasel, J.: *A Gentle Introduction to Haskell*, 2000

[JL92] Jones, S. L. P.; Lester, D.: *Implementing Functional Languages*. Prentice Hall, 1992

[Jon03] Jones, S. P.: *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003

[Mic88] Michaelson, G.: *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley Publishing Company, 1988

[Mil99] Milner, R.: *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999

[NSS06] Niehren, J.; Schwinghammer, J.; Smolka, G.: A concurrent lambda calculus with futures. **In:** *Theor. Comput. Sci.* 364 (2006), Nr. 3, S. 338–356. – ISSN 0304–3975

[OSG08] O'Sullivan, B.; Stewart, D. B.; Goerzen, J.: *Real World Haskell*. O'Reilly, 2008

[Par01] Kap. 3 **In:** Parrow, J.: *An introduction to the pi-calculus in Handbook of Process Algebra*. Elsevier, 2001, S. 479–543

[Pau96] Paulson, L. C.: *ML for the Working Programmer*. Cambridge University Press, 1996

[Poh93] Pohlers, W.: *Mathematische Grundlagen der Informatik*. Oldenbourg, 1993

[Rep99] Reppy, J. H.: *Concurrent Programming in ML*. Cambridge University Press, 1999

[RL99] Rabhi, F.; Lapalme, G.: *Algorithms: a functional programming approach*. Addison-Wesley, 1999

[Sab08] Sabel, D. Semantics of a call-by-need lambda calculus with McCarthy's amb for program equivalence. 2008

[Sch97] Schneider, F. B.: *On Concurrent Programming*. Springer, 1997

[SPJ96] SL Peyton Jones, S. F.: Concurrent Haskell. **In:** *23rd ACM Symposium on Principles of Programming Languages* (1996), S. 295–308

[SS89] Søndergaard, H.; Sestoft, P.: Referential Transparency, Definiteness and Unfoldability. **In:** *Acta Inf.* 27 (1989), Nr. 6, S. 505–517

[SS06] Schmidt-Schauss, M. Skript zur Vorlesung "Einführung in die Funktionale Programmierung"(WS 2006/2007). 2006

[SS07] Schmidt-Schauss, M. Skript zur Vorlesung "Funktionale Programmierung: Programmtransformationen"(WS 2006/2007). 2007

[SS207] Funktionale Programmierung. Anleitung zum Praktikim. http://www.ki.informatik.uni-frankfurt.de/lehre/SS2007/FPPR. SS 2007

[SSNSS08] Schwinghammer, J.; Sabel, D.; Niehren, J.; Schmidt-Schauss, M.: On Proving the Equivalence of Concurrency Primitives / Research group for Artificial Intelligence and Software Technology. Goethe Universität Frankfurt, Germany, 2008. – Frank-34

[SSS08] Schmidt-Schauss, M.; Sabel, D.: Closures of May and Must Convergence for Contextual Equivalence / Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main. 2008 (35). – Frank report

[Tho99] THOMPSON, S.: *Haskell. The Craft of Functional Programming*. Addison-Wesley, 1999

[Tra91] TRAUB, K. R.: *Implementation of Non-Strict Functional Programming LAnguages*. Pitman, 1991

[WM95] WILHELM, R.; MAURER, D.: *Compiler Design*. Addisson-Wesley Publishing Company, 1995