CCFL
A *C*oncurrent *C*onstraint *F*unctional *L*anguage

Petra Hofstedt
ph@cs.tu-berlin.de

Technischer Bericht 2008-08
ISSN 1436-9915

Contents

List of Figures

This report describes the design and implementation of the Concurrent Constraint Functional Language CCFL.

CCFL combines concepts from the functional and the constraint paradigms and allows the description of systems of concurrent processes, whose communication and synchronization is based on the concurrent constraint programming (CCP) model [SR90]. A compiler for the language has been implemented in HASKELL which translates CCFL programs into LMNtal code. The language LMNtal (pronounced "elemental") [UK02, UK05, UKHM06, LMN07] has been designed and developed by Kazunori Ueda and his group at Waseda University in Tokyo. It is a new concurrent language model based on rewriting hierarchical graphs. One of its major aims is to unify various paradigms of computation resp. computational models. Thus, LMNtal lends itself as base model and target language for the compilation of CCFL programs.

This report is structured as follows: Chapter 1 introduces CCFL by examples and discusses the formal representation. Chapter 2 deals with context conditions and type checking. We give a brief introduction to the compiler target language LMNtal in Chapter 3, where we also discuss restrictions on LMNtal links which caused different approaches for the code generation. Chapter 4 presents a naive approach to code generation for a functional sublanguage of CCFL, while Chapter 5 shows an extended compilation scheme for full CCFL. Chapter 6 is dedicated to the extension of CCFL by new constraint domains. Finally we conclude and discuss related and future work in Chapter 7.

Chapter 1

The language CCFL

The first chapter introduces the Concurrent Constraint Functional Language CCFL. We start with an informal presentation of structure and usage of CCFL by means of examples in Section 1.1. Section 1.2 deals with the formal description of the syntax of CCFL and Section 1.3 sketches on the language semantics.

1.1 Programming with CCFL

CCFL is a declarative language combining the functional and constraint paradigms which allows for concurrent computation.

A CCFL program is a sequence of data type definitions, functional and constraint abstractions. *Functional abstractions* resp. *functions* are used to express deterministic computations. *Constraint abstractions* resp. *user-defined constraints* are used to describe cooperating processes and non-deterministic behaviour. Constraint abstractions represent a number of possible computations, that is, relations resp. *constraints* over their arguments.

1.1.1 Functional programming

The functional sub-language partially inherits notions and concepts from the functional languages HASKELL and OPAL (see e.g. [Bir98, Hud00, PH06]). A functional CCFL program consists of data type definitions and functional abstractions resp. functions. A function consists of a function definition and a function type declaration. Function definitions allow typical constructs such as case-expressions, let-expressions, function application and some predefined infix operation applications, constants, variables and constructor terms. A function call[1] evokes a computation by reduction.

Program 1.1.1 and the function *append* in Program 1.1.2 are typical examples of functional abstractions in CCFL. Program 1.1.2 moreover shows the user-defined data type *List a*.

1.1.2 Free variables

Free variables are one of the main characteristics of constraints. However, in CCFL expressions in general are allowed to contain free variables. This also applies to function applications.

Thus, function applications are evaluated using the residuation principle [Smo93]. This means, functions calls are suspended until variables are bound to values or terms such that a deterministic reduction is possible. For example a function call $(4 + x)$ with free variable x will suspend.

[1]without free variables (see Section 1.1.2)

Program 1.1.1 A naive implementation of the Fibonacci function

```
fun fibonacci :: Int -> Int
def fibonacci n =
  case n of 0 -> 0 ;
            1 -> 1 ;
            m -> let f1 = fibonacci (m-1);
                     f2 = fibonacci (m-2)
                 in f1 + f2
```

Program 1.1.2 List append

```
data List a = Nil | Cons a (List a)

fun append :: List a -> List a -> List a
def append l1 l2 =
   case l1 of Nil -> l2 ;
              Cons x xs -> Cons x (append xs l2)

fun main :: List Int
def main =
  with y :: Int
  in append (Cons 1 Nil) (Cons y Nil)
```

Consider in contrast Program 1.1.2. Function *main* introduces the free variable *y* of type ***Int*** using the with-construct. Here a concrete binding of *y* is not necessary to proceed with the computation of *main*. The result of *main* is thus *Cons* 1 (*Cons y Nil*).

1.1.3 Constraint abstractions

A CCFL program usually describes a combination of functional and constraint abstractions.

A constraint abstraction consists of a head and a body which may contain the same elements as a functional abstraction. Additionally, the body can be defined by several alternatives the choice of which is decided by guards. A constraint abstraction is allowed to introduce free variables and each body alternative is a conjunction of constraint atoms. A constraint abstraction has result type ***C***.

Examples of constraint abstractions are *game*, *dice*, and *isIn* in Program 1.1.3 and *produce*, *consume*, and *main* in Program 1.1.4 which we will discuss in detail in the following.

Ask- and *tell*-constraints

The atoms of the guard of an alternative are called *ask*-constraints (see also [SR90]). If a guard of a rule with matching left-hand side is entailed, the concerning rule alternative may be chosen for further derivation. In case the guard fails or cannot be decided (yet), this rule alternative is suspended. If all matching rule alternatives suspend, the computation waits (possibly infinitely) for a sufficient instantiation of the concerning variables.

For *ask*-constraints, we distinguish between bound-constraints (***bound*** *x*) checking whether a variable *x* is bound to a non-variable term and match-constraints (*x* =:= *c x_1 ... x_n*) which test for a concrete matching of the root symbol of a term bound to the variable *x* with a certain constructor *c*.

Constraints in the body of a rule are called *tell*-constraints [SR90] and they generate bindings. *Tell*-constraints include applications (*f fexpr_1 ... fexpr_n*) of user-defined con-

Program 1.1.3 A simple game of dice

```
fun game :: Int -> Int -> Int -> C
def game x y n =
   case n of 0 -> x =:= 0 & y =:= 0 ;
             m -> with x1 :: Int, y1 :: Int, x2 :: Int, y2 :: Int
                  in dice x1 & dice y1 &
                     x =:= add x1 x2 & y =:= add y1 y2 &
                     game x2 y2 (m-1)

fun add :: Int -> Int -> Int
def add x y = x + y

fun dice :: Int -> C
def dice x =
   isIn (Cons 1 (Cons 2 (Cons 3 (Cons 4 (Cons 5 (Cons 6 Nil)))))) x

fun isIn :: List a -> a -> C
def isIn l x =
    l =:= Cons y ys -> x =:= y |
    l =:= Cons y ys -> case ys of Nil -> x =:= y ;
                                  Cons z zs -> isIn ys x
```

straints and equality constraints (x =:= *fexpr*) on a variable x and a functional expression *fexpr*. While a satisfiable equality constraint produces a binding and terminates with result value *Success*, a unsatisfiable equality is reduced to the value *Fail* representing an unsuccessful computation.

Example 1.1.1 Presuppose the user-defined data type *List a* from Program 1.1.2. The body of the constraint abstraction *failList* consists of a conjunction of two conflicting equality constraints.

```
fun failList :: C
def failList = with x :: List Int in x =:= Nil & x =:= Cons 1 Nil
```

The derivation of the constraint *failList* yields *Fail*. The arrow $\rightsquigarrow$ stands for a computation step, where we may annote computed bindings for free variables.

$$\mathit{failList} \rightsquigarrow (x =:= \mathit{Nil}) \;\&\; (x =:= \mathit{Cons}\ 1\ \mathit{Nil}) \rightsquigarrow_{\{x/\mathit{Nil}\}} (x =:= \mathit{Cons}\ 1\ \mathit{Nil}) \rightsquigarrow \mathit{Fail}$$

Non-deterministic computations

Constraint abstractions resp. user-defined constraints can be used to describe non-deterministic computations.

Consider Program 1.1.3 as an example. It shows a game between two players throwing the dice n times. The user-defined constraint *game* uses both kinds of *tell*-constraints: The *constraint applications dice x1* and *dice y1* non-deterministically produce values which are consumed by the applications of the function *add* in the *equality constraints* x =:= *add x1 x2* and y =:= *add y1 y2* resp. These function applications suspend until their arguments are sufficiently instantiated, that is, completely instantiated for addition in this example. The constraint abstraction *isIn* chooses a value from a list. Since the match-constraints of the guards of both alternatives are the same, i.e. (l =:= *Cons y ys*), the alternatives are chosen non-deterministically.

The following trace shows a non-deterministic computation with $n = 2$. The arrows $\rightsquigarrow_{\ldots}$ and $\rightsquigarrow^{\star}_{\ldots}$ stand for one or some computation steps resp.

Program 1.1.4 A producer and a consumer communicating over a buffer

```
fun produce :: List a -> C
def produce buf =
    with buf1 :: List a, item :: a
    in -- produce item here
       ...
       -- then put it into the buffer and continue
       buf =:= Cons item buf1 & produce buf1

fun consume :: List a -> C
def consume buf =
    buf =:= Cons first buf1 ->
       -- consume first here
       ...
       -- and continue
       consume buf1

fun main :: C
def main =
    with buf :: List a
    in produce buf & consume buf
```

$$
\begin{array}{l}
\mathit{game}\ a\ b\ 2 \\
\quad \leadsto_{\ldots}\ \mathit{dice}\ \mathit{a1}\ \&\ \mathit{dice}\ \mathit{b1}\ \&\ a =:= \mathit{add}\ \mathit{a1}\ \mathit{a2}\ \&\ b =:= \mathit{add}\ \mathit{b1}\ \mathit{b2}\ \&\ \mathit{game}\ \mathit{a2}\ \mathit{a2}\ 1 \\
\quad \leadsto^{\star}_{\{a1/2,b1/1\}}\ a =:= \mathit{add}\ 2\ \mathit{a2}\ \&\ b =:= \mathit{add}\ 1\ \mathit{b2}\ \&\ \mathit{game}\ \mathit{a2}\ \mathit{b2}\ 1 \\
\quad \leadsto_{\ldots}\ a =:= \mathit{add}\ 2\ \mathit{a2}\ \&\ b =:= \mathit{add}\ 1\ \mathit{b2}\ \& \\
\qquad \mathit{dice}\ \mathit{a3}\ \&\ \mathit{dice}\ \mathit{b3}\ \&\ \mathit{a2} =:= \mathit{add}\ \mathit{a3}\ \mathit{a4}\ \&\ \mathit{b2} =:= \mathit{add}\ \mathit{b3}\ \mathit{b4}\ \&\ \mathit{game}\ \mathit{a4}\ \mathit{b4}\ 0 \\
\quad \leadsto^{\star}_{\{\ldots,a3/4,b3/6\}}\ a =:= \mathit{add}\ 2\ \mathit{a2}\ \&\ b =:= \mathit{add}\ 1\ \mathit{b2}\ \& \\
\qquad \mathit{a2} =:= \mathit{add}\ 4\ \mathit{a4}\ \&\ \mathit{b2} =:= \mathit{add}\ 6\ \mathit{b4}\ \&\ \mathit{game}\ \mathit{a4}\ \mathit{b4}\ 0 \\
\quad \leadsto_{\{\ldots,a4/0,b4/0\}}\ a =:= \mathit{add}\ 2\ \mathit{a2}\ \&\ b =:= \mathit{add}\ 1\ \mathit{b2}\ \&\ \mathit{a2} =:= \mathit{add}\ 4\ 0\ \&\ \mathit{b2} =:= \mathit{add}\ 6\ 0 \\
\quad \leadsto^{\star}_{\{\ldots,a2/4,b2/6\}}\ a =:= \mathit{add}\ 2\ 4\ \&\ b =:= \mathit{add}\ 1\ 6 \\
\quad \leadsto^{\star}_{\{\ldots,a/6,b/7\}}\ \mathit{Success}
\end{array}
$$

Concurrent processes

Constraint abstractions also allow to describe the computation of concurrent processes. For example, in Program 1.1.3 both computations of dice values, i.e. *dice x1* and *dice y1* which are independent, could have been computed concurrently. There is no required order of the players in this game. However, the computation of the sum by $x =:= \mathit{add}\ \mathit{x1}\ \mathit{x2}$ depends on the computation of the dice values which must have been processed before.

In general, constraint applications enable the description of computations by means of cooperating and communicating processes. Program 1.1.4 shows an example of a consumer and a producer cooperating over a common buffer.

While the *consume* process must wait until the buffer has been filled with at least one element *first*, the *produce* process is not restricted to synchronise with the consumer.

However, one may e.g. restrict the buffer to force the producer to take the behaviour of the consumer into account. Consider Program 1.1.5. The *main* abstraction initialises the buffer with two element. The *produce* process is allowed to proceed as long as there are free elements in the buffer which it binds to values. In contrast, the *consume* process consumes the values but puts new free slots into the buffer. In this way, producer and consumer communicate and synchronise over the bounded buffer.

Program 1.1.5 A bounded buffer holding at most two elements

```
fun produce :: List a -> C
def produce buf =
  buf =:= Cons first buf1 ->
    -- compute a value v for first here
    ...
    -- then bind it to first and continue
    first =:= v & produce buf1

fun consume :: List a -> C
def consume buf =
  buf =:= Cons v buf1 & bound v ->
    with a :: a, b :: a, buf2 :: List a
    in -- consume value v here
       ...
       -- then re-initialise the buffer and continue
       buf1 =:= Cons a (Cons b buf2) & consume buf1

fun main :: C
def main =
  with a :: a, b :: a, buf :: List a, buf1 :: List a
  in  -- call producer and consumer ...
      -- with buffer buf of two unbound elements
      produce buf & consume buf & buf =:= Cons a (Cons b buf1)
```

1.2 Syntax

This section presents the formal syntax of CCFL.

1.2.1 CCFL programs

A CCFL program consists of data type definitions and declarations and definitions of functions and user-defined constraints. In the following we use the notion function for both, i.e. we consider user-defined constraints as functions of result type ***C***.

A data type definition starts with the keyword ***data***. We (currently) do not allow functions to appear in user-defined data types.

A function declaration is marked with the keyword ***fun***, a function definition with the keyword ***def***.

$$
\begin{array}{lcl}
\mathit{Prog} & \rightarrow & \mathit{Def}^{\,+} \\
\mathit{Def} & \rightarrow & \mathit{TDef} \mid \mathit{FDecl} \mid \mathit{FDef} \\
\\
\mathit{FDef} & \rightarrow & \textbf{\textit{def}}\ \mathit{FName}\ \mathit{Var}^{\,\star}\ \boldsymbol{=}\ \mathit{Expr} \\
\mathit{FDecl} & \rightarrow & \textbf{\textit{fun}}\ \mathit{FName}\ \boldsymbol{::}\ \mathit{Type} \\
\mathit{TDef} & \rightarrow & \textbf{\textit{data}}\ \mathit{TypeName}\ \mathit{TypeVar}^{\,\star}\ \boldsymbol{=}\ \mathit{CType}\ (\boldsymbol{|}\ \mathit{CType})^{\,\star} \\
\\
\mathit{CType} & \rightarrow & \mathit{Constructor}\ \mathit{SType}^{\,\star} \\
\mathit{SType} & \rightarrow & \mathit{TypeVar} \mid \boldsymbol{(}\ \mathit{CType}\boldsymbol{)} \mid \textbf{\textit{Int}} \mid \textbf{\textit{Float}} \mid \textbf{\textit{Bool}} \\
\mathit{Type} & \rightarrow & \mathit{SType} \mid \mathit{Type} -> \mathit{Type} \mid \textbf{\textit{C}}
\end{array}
$$

Names consist of alphanumerical symbols; data type names and constructors start with an upper letter while identifiers of variables and functions resp. user-defined constraints start with a lower letter.

Program 1.2.1 shows a definition of a data type (*Tree a*) containing the empty tree *Nil* and trees consisting of nodes *Node* with a value of (the polymorphic) type *a* and a left and a right sub-tree. Furthermore, it declares and defines an arithmetic function *foo* with two arguments.

Program 1.2.1 A data type definition for trees and a simple function definition on floats

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

fun foo :: Float -> Float -> Float
def foo x y = y +. (x /. y)
```

1.2.2 Expressions

We distinguish functional, constraint and guarded expressions.

Expr	→	*FExpr* | *CExpr* | *GExpr*

Functional expressions

A functional expression is either a function application, a (optionally parenthesised) basic infix operator application, a case-expression or a let-expression.

FExpr	→	*Appl* | *Infix* | *CaseExpr* | *LetExpr*

A function application consists of a (optional) functional expression and a simple expression, like a variable, a constant, a parenthesised functional expression or a constructor (expression). Signed numbers must be enclosed within parentheses.

Appl	→	*SExpr* | *FExpr SExpr*
SExpr	→	*Var* | *Constant* | **(** *FExpr* **)** | *Constructor*
Constant	→	*Float* | *Int* | *Bool*

A number of basic binary arithmetic and Boolean operations and relations can be used in infix notation. Operations on floating point numbers are followed by a dot ".", comparison operations on Booleans by a colon ":".

Infix	→	*FExpr BinOp FExpr*
BinOp	→	*ArithOp* | *RelOp* | *BoolOp*
ArithOp	→	**+** | **-** | ***** | **/** | **+.** | **-.** | ***.** | **/.**
RelOp	→	== | <= | < | > | >= | ~= | ==: | ~=:
		| == . | <= . | < . | > . | >= . | ~= .
BoolOp	→	&& | ||

Case-expressions use the keywords ***case*** and ***of*** and the different branches are separated by semicolon ";". Different cases are distinguished by their top constructor symbols, where the following variables must match the arity of the constructor. We allow to branch on natural numbers and Booleans by considering them as (0-ary) constructors. An otherwise-branch can be defined for case alternatives on natural numbers as shown for the naive implementation of the function *fibonacci* in Program 1.1.1 in line 5.

An if-construct is realised as function using the case-construct of basic CCFL as shown in Program 1.2.2. Function *depth* calculates the depth of a tree of type *Tree a* as defined in

Program 1.2.2 if-alternatives are realised using the case-construct

```
fun if :: Bool -> a -> a -> a
def if c t e = case c of True -> t ;
                         False -> e

fun depth :: Tree a -> Int
def depth tree =
  case tree of
     Nil -> 0 ;
     Node left value right ->
        let ldepth = depth left ;
            rdepth = depth right
        in if (ldepth >= rdepth) (ldepth + 1) (rdepth + 1)
```

Program 1.2.3 A function foo using local definitions

```
fun foo :: Int -> Int -> Int

def foo a b = a + (let a = b;
                       b = a + 2
                   in (let a = a + b
                       in b + a))

-- or semantically equivalent:
def foo a b = a + (let a0 = b;
                       b1 = a0 + 2
                   in (let a2 = a0 + b1
                       in b1 + a2))
```

Program 1.2.1.

CaseExpr	→	**case** *FExpr* **of** *Branch* (**;** *Branch*) *	
Branch	→	*Constructor Var* * − > *CExpr*	
		| *Var* − > *CExpr*	(⋆ *otherwise-branch* ⋆)

Local definitions are introduced by let-expressions. They are allowed to refer to previously defined let-elements; thus, their order is crucial.[2]

Program 1.2.3 show two semantically equivalent versions of a function *foo* illustrating the effect of shadowing variables.

LetExpr	→	**let** *Binding* (**;** *Binding*) * **in** *CExpr*
Binding	→	*Var* **=** *FExpr*

Constraint expressions

The second form of expressions are constraint expressions. These may introduce new logical variables (annotated with their type) using the with-construct and consist in general of a conjunction of atoms. An atom is either a *tell*-equality or a functional expression (including

[2]Alternatively (and which is as well implemented), one could put all local let-bindings on the same level such that they do not refer to each other. In this case, it is unproblematic to change their order.

applications of user-defined constraints).

CExpr	→	*With* | *Conj*	
With	→	***with*** (*Var* :: *CType*) $^+$ ***in*** *Conj*	
Conj	→	*Atom* (& *Atom*) *	
Atom	→	*Var* **=:=** *FExpr*	($\star$ *tell-equality* $\star$)
		| *FExpr*	($\star$ *functional expression* $\star$)

Guarded expressions

Finally, there are guarded expressions: They consist of a disjunction of constraint expressions, each preceded by a guard which is a conjunction of constraint primitives. While constraint expressions may be looked at as *tell*-constraints, guard atoms represent *ask*-constraints [SR90]. They include a test ***bound*** *x* of a variable *x* to be bound to a non-variable term and a match-constraint *x* =:= *c x_1 ... x_n* of a variable *x* to be bound to a term with constructor root symbol *c* and fresh variables *x_1 ... x_n* as its arguments.

GExpr	→	*GAlt* (|*GAlt*) *
GAlt	→	*Guard* → *CExpr*
Guard	→	*CPrim* (& *CPrim*) *
CPrim	→	***bound*** *Var* | *Var* **=:=** *CTerm*
CTerm	→	*Constructor* *Var* *

The producer and the consumer processes of Program 1.1.4 are defined using constraint and guarded expressions.

Constants, constructors, variable and function names and numbers as used above are defined by the grammar below.

Bool	→	***True*** | ***False***
Int	→	([*Sign*] *Digit* $^+$) | *Digit* $^+$
Float	→	([*Sign*] *Digit* $^+$ **.** *Digit* $^+$ [*Ext*])
		| *Digit* $^+$ **.** *Digit* $^+$ [*Ext*]
Ext	→	(***e*** | ***E***) [*Sign*] *Digit* $^+$
Var	→	*LowerId*
TypeVar	→	*LowerId*
FName	→	*LowerId*
Constructor	→	*UpperId*
TypeName	→	*UpperId*
LowerId	→	*LAlpha* (*Alpha* | *Digit*) *
UpperId	→	*UAlpha* (*Alpha* | *Digit*) *
Alpha	→	*UAlpha* | *LAlpha*
LAlpha	→	a | ... | z
UAlpha	→	A | ... | Z
Digit	→	0 | ... | 9
Sign	→	+ | −

1.2.3 Abstract syntax

We show the structure of the abstract syntax of CCFL programs of our implementation. The abstract syntax strongly reflects the descriptions in Sections 1.2.1 and 1.2.2.

Programs are lists of function definitions, function declarations, and data type definitions. These are represented by the data type *Def* and distinguishing constructors as given in

Program 1.2.4 Programs

```
type Prog = [Def]

data Def = FDef  { fName  :: String,
                   fPars  :: [Var],
                   fBody  :: Expr }
         | FTDef { ftName :: String,
                   ftType :: Type }
         | TDef  { tName  :: String,
                   tPars  :: [String],
                   tBody  :: [TElem] }

data TElem = TElem { tConstr :: String,
                     eTypes  :: [Type] }

data Type = TConstr String [Type]
          | TVar String
          | TInt
          | TFloat
          | TBool
          | TFun Type Type
          | TConstraint
```

Program 1.2.5 Expressions

```
data Expr = FExpr FExpr
          | CExpr CExpr
          | GExpr GExpr
```

Program 1.2.4.

Expressions are again distinguished into functional, constraint, and guarded expressions (see data type *Expr*, Program 1.2.5).

Functional expressions (Program 1.2.6) are separated into case-expressions, let-expressions, basic infix operations, applications, constants, identifiers (function or variable names) and constructor expressions.

Constraint expressions (Program 1.2.7) may introduce new logical variables and consist furthermore just of a conjunction of atoms. Atoms are *tell*-equality-constraints or functional expressions.

Finally, there remain guarded expressions as lists of alternatives, each consisting of a conjunction of guard atoms (bound or match constraints) and a constraint expression as shown in Program 1.2.8.

Example

An example of a data type definition for lists and a function computing the length of lists in CCFL together with their abstract representations is given in Program 1.2.9.

1.2.4 Implementation

The files *Syntax.hs* and *Parser.hs* implement the abstract syntax and the CCFL parser, resp. The parser relies on the monadic parser combinator library Parsec [Lei01].

Program 1.2.6 Functional expressions

```
data FExpr = Case    { caseFExpr :: FExpr,
                       branches  :: [Branch] }
           | Op      { opName :: String,
                       opFE1  :: FExpr,
                       opFE2  :: FExpr }
           | Apply   { appFE1 :: FExpr,
                       appFE2 :: FExpr }
           | Let     { letBdgs :: [Binding],
                       letCE   :: CExpr }
           | Const   { constant :: Constant }
           | FunVar  { variable :: Var }
           | Constr  { feConstructor :: String }

data Var = Var { varName :: String }

data Constant = Float Double
              | Int Integer
              | Bool Bool

data Branch = CBranch { constrName   :: String,
                        constrArgs   :: [Var],
                        cConsequence :: CExpr }
            | BBranch { bAlt :: Bool,
                        bConsequence :: CExpr }
            | NBranch { nAlt :: Integer,
                        nConsequence :: CExpr }
            | OBranch { oAlt :: Var,
                        oConsequence :: CExpr }

data Binding = Bind { bindVar :: Var,
                      bindFe  :: FExpr }
```

Program 1.2.7 Constraint expressions

```
data CExpr = With { logicalVars :: [Var],
                    withCe :: CExpr }
           | Conj [Atom]

data Atom = TellEq { tellVar :: Var,
                     tellFe  :: FExpr }
          | CFExpr { cFe :: FExpr }
```

Program 1.2.8 Guarded expressions

```
type GExpr = [GAlt]

data GAlt = GAlt { guards :: [Guard],
                   gCe :: CExpr }

data Guard = Bound { boundVar :: Var }
           | Match { matchVar :: Var,
                     matchT :: CTerm }

data CTerm = GConst { gconstant :: Constant }
           | GVar Var
           | GConstr { cTConstr :: String,
                       cTArgs :: [Var] }
```

Program 1.2.9 Definitions and declaration and their abstract representations

```
data List a = Nil | Cons a (List a)

fun length :: List a -> Int
def length list =
  case list of Nil -> 0 ;
               Cons elem rest -> 1 + length rest

[TDef "List"
    ["a"]
    [TElem "Nil" [],
     TElem "Cons" [TVar "a",
                   TConstr "List" [TVar "a"]]],

 FTDef "length"
    (TFun (TConstr "List" [TVar "a"]) TInt),

 FDef "length" [Var "list"]
    (FExpr
        (Case (FunVar (Var "list"))
              [CBranch "Nil" []
                       (Conj [CFExpr (Const (Int 0))]),
               CBranch "Cons" [Var "elem", Var "rest"]
                       (Conj [CFExpr
                                (Op "+"
                                  (Const (Int 1))
                                  (Apply (FunVar (Var "length"))
                                         (FunVar (Var "rest"))))
                             ])]))]
```

1.3 Semantics

CCFL was intended to realise a *lazy evaluation* strategy. However, we examine the evaluation of functional CCFL using a *call-by-value* and a *call-by-name strategy* in Section 4.7 as well.

Since we allow free variables in general and, thus, also for function applications, these are evaluated using the *residuation* principle [Smo93]. That means, functions calls are suspended until variables are bound to values or terms such that a deterministic reduction is possible.

Tell-equality-constraints are interpreted as *strict* [HAB$^+$06]. That is, the constraint x =:= *fexpr* is satisfied, if both expressions can be reduced to the same ground data term.

We discuss these issues in the Sections 4.7, 5.2.2, and 5.3.1 in combination with the compilation scheme.

Chapter 2

Context conditions and type checking

In this chapter we discuss the context conditions checked by the compiler, variable scopes, and the type checking algorithm.

2.1 Context conditions

Context conditions checked by the CCFL compiler include amongst others ...

- checks on multiple declarations and definitions of functions (including predefined functions), multiply defined data types, and multiply declared data constructors;
- checks to ensure that variable and function names have been introduced before they are used;
- checks on and renaming of re-declared (shadowing) variable names in
  - nested let-expressions and
  - with-expressions for logical variables;
- checks on multiple declarations of variable names within
  - the parameters of a function definition,
  - the sequence of logical variables introduced by a with-construct,
  - the variables *x_i*, $i \in \{1, \ldots, n\}$ in a match-constraint *x =:= c x_1 ... x_n* of a guard, and
  - constructor pattern variables of a case-expression;
- checks to ensure that placeholder variables in
  - otherwise-branches,
  - constructor patterns in case-expressions, and
  - match-constraints

  have not been declared before, i.e. they must be fresh;
- a test to ensure that at most one otherwise-branch appears per case-construct (matching on natural numbers).

2.2 Scopes

In the following we show visibility rules for variable and function names by means of environments. An *environment* $\mathcal{E}$ of a certain point in the program contains all names which are valid resp. known there.

Function definitions, function declarations, and type definitions. The names of all function definitions and declarations and of all type definitions are visible for each other. This includes also all predefined/built-in functions and data-types. This allows functions to be defined on top of other functions and function declarations and data type definitions to use other types.

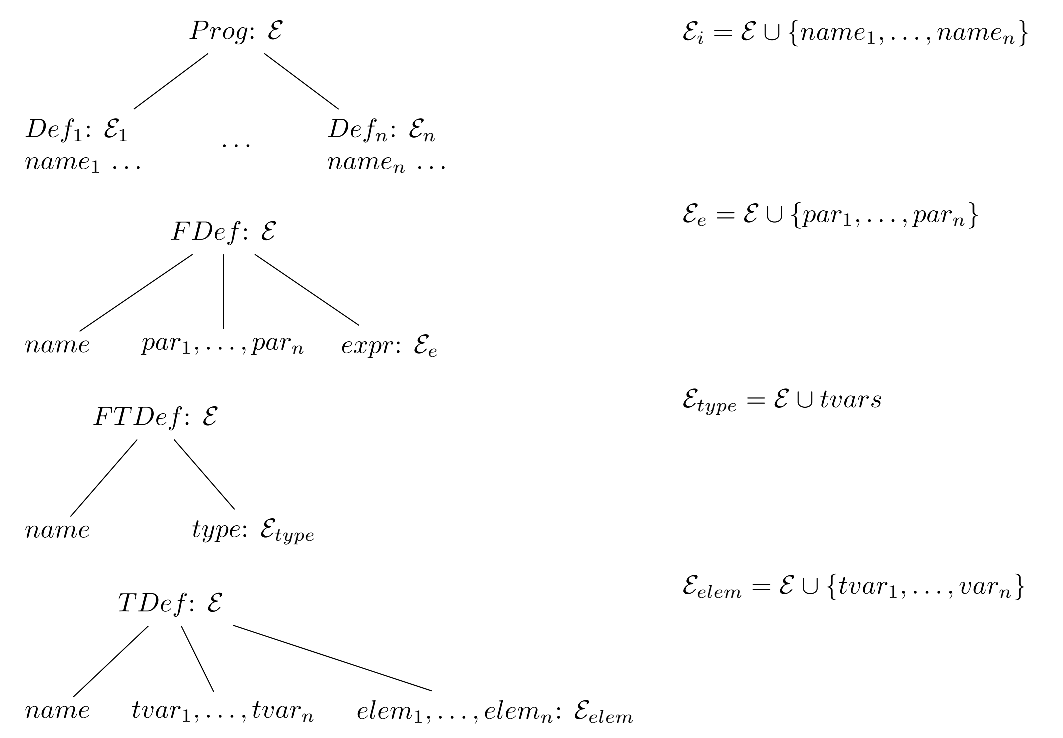

$Prog\colon \mathcal{E}$
$Def_1\colon \mathcal{E}_1$ $name_1 \ldots$
$\ldots$
$Def_n\colon \mathcal{E}_n$ $name_n \ldots$
$\mathcal{E}_i = \mathcal{E} \cup \{name_1, \ldots, name_n\}$
$FDef\colon \mathcal{E}$
$name$
$par_1, \ldots, par_n$
$expr\colon \mathcal{E}_e$
$\mathcal{E}_e = \mathcal{E} \cup \{par_1, \ldots, par_n\}$
$FTDef\colon \mathcal{E}$
$name$
$type\colon \mathcal{E}_{type}$
$\mathcal{E}_{type} = \mathcal{E} \cup tvars$
$TDef\colon \mathcal{E}$
$name$
$tvar_1, \ldots, tvar_n$
$elem_1, \ldots, elem_n\colon \mathcal{E}_{elem}$
$\mathcal{E}_{elem} = \mathcal{E} \cup \{tvar_1, \ldots, var_n\}$

Figure 2.1: Scoping rules for function definitions, function declarations, and type definitions

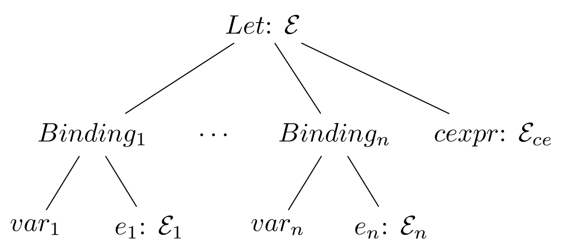
Function declarations $FTDef$ can use predefined and user-defined data types and arbitrary type variables *tvar*. Type definitions introduce a type name and type variables which can be used to define the types of the elements of the data type. The concerning rules are presented in Figure 2.1.

Local definitions. Using the keyword ***let*** local definitions introduce new variable names. Since they are allowed to be used in following local definitions, their order is essential. All newly introduced variable names are finally known in the expression following the keyword ***in*** (see Figure 2.2).

Note that the new variable names may shadow previously declared variable names. Thus the program analysis performs a renaming of shadowing variables.

Case-expressions. According to Program 1.2.6 we distinguish four kinds of branch alternatives in case-expressions (see Figure 2.3):

- *CBranch* for alternative constructor patterns of user-defined data types,



$$\begin{aligned}\mathcal{E}_1 &= \mathcal{E},\\ \mathcal{E}_i &= \mathcal{E}_{i-1} \cup \{var_{i-1}\}, i \in \{2,\ldots,n\},\\ \mathcal{E}_{ce} &= \mathcal{E}_n \cup \{var_n\}\end{aligned}$$

Figure 2.2: Scoping rules for local definitions

- *BBranch* to decide for a program branch on Booleans,
- *NBranch* to decide on natural numbers, and
- *OBranch* as default (otherwise-branch) for natural numbers.

For case-expressions on natural numbers it is possible to use an otherwise-branch *OBranch* to represent a default alternative for all remaining cases. Per case-expression one *OBranch* may occur; this is optional, thus, represented in gray color in Figure 2.3.

Concerning names we need to consider here in particular variables in constructor pattens and placeholder variables of otherwise-branches because both introduce fresh variables.

Constraint expressions. Using the with-construct the programmer may introduce new logical variables in constraint expressions. These may shadow previously introduced variables. Figure 2.4 shows the scoping rule for the with-construct.

Guarded Expressions. Constraints in guards reside all on the same level, thus, they may exchange their order arbitrarily.

In guarded expressions match-constraints are the interesting point wrt. the aspect of scoping because they may introduce new variables. While a variable *x* on the left-hand side of a match-constraint *x =:= c x_1 ... x_n* must have been introduced previously, the variables *x_1 ... x_n* on the right-hand side are fresh. The concerning rules are shown in Figure 2.5.

Further constructs. All other program constructs take over the environments of their parent nodes, as it is shown in Figure 2.6 for infix operation applications as an example.

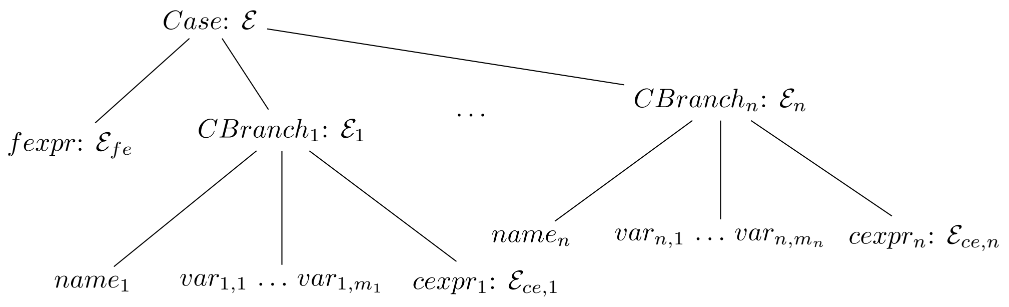
2.3 Types

This section considers the type system of CCFL.

2.3.1 Data types

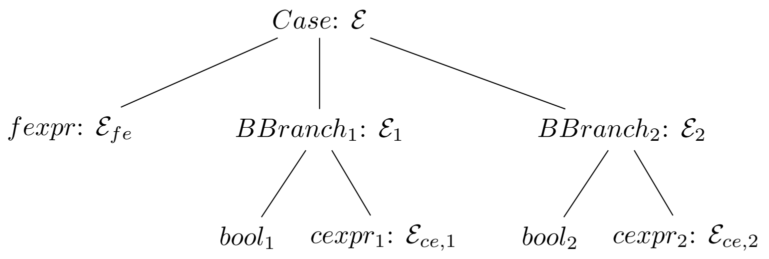
CCFL has four built-in data types as shown in Figure 2.7. Additionally, the user may define algebraic data types using the keyword ***data*** as shown e.g. in the Programs 1.1.2 and 1.2.1. For data type definitions the compiler checks the following properties:

- All types in function declarations and in declarations of logical variables in with-constructs are type variables, built-in types, or have been defined in the users program.
- In the definition of data type elements
  - all type variables must have been declared before,



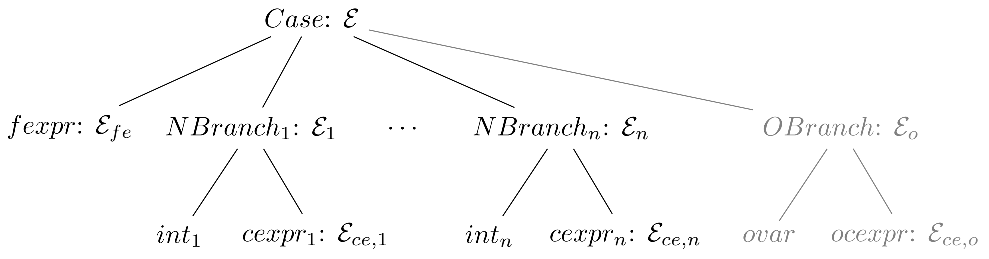
$Case\colon \mathcal{E}$
$fexpr\colon \mathcal{E}_{fe}$
$CBranch_1\colon \mathcal{E}_1$
$\ldots$
$CBranch_n\colon \mathcal{E}_n$
$name_1$
$var_{1,1} \ldots var_{1,m_1}$
$cexpr_1\colon \mathcal{E}_{ce,1}$
$name_n$
$var_{n,1} \ldots var_{n,m_n}$
$cexpr_n\colon \mathcal{E}_{ce,n}$

$\mathcal{E}_i \quad = \mathcal{E},\ i \in \{fe, 1, \ldots, n\}$,
$\mathcal{E}_{ce,i} = \mathcal{E}_i \cup \{var_{i,j}\},\ i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m_i\}$



$Case\colon \mathcal{E}$
$fexpr\colon \mathcal{E}_{fe}$
$BBranch_1\colon \mathcal{E}_1$
$BBranch_2\colon \mathcal{E}_2$
$bool_1$
$cexpr_1\colon \mathcal{E}_{ce,1}$
$bool_2$
$cexpr_2\colon \mathcal{E}_{ce,2}$

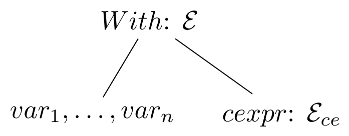
$\mathcal{E}_i = \mathcal{E}_{ce,j} = \mathcal{E},\ i \in \{fe, 1, 2\},\ j \in \{1, 2\}$



$Case\colon \mathcal{E}$
$fexpr\colon \mathcal{E}_{fe}$
$NBranch_1\colon \mathcal{E}_1$
$\cdots$
$NBranch_n\colon \mathcal{E}_n$
$OBranch\colon \mathcal{E}_o$
int_1
$cexpr_1\colon \mathcal{E}_{ce,1}$
int_n
$cexpr_n\colon \mathcal{E}_{ce,n}$
$ovar$
$ocexpr\colon \mathcal{E}_{ce,o}$

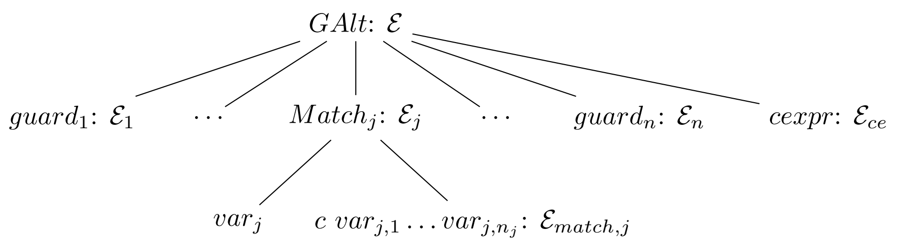
$\mathcal{E}_i \quad = \mathcal{E}_{ce,j} = \mathcal{E},\ i \in \{fe, 1, \ldots, n, o\},\ j \in \{1, \ldots, n\}$,
$\mathcal{E}_{ce,o} = \mathcal{E} \cup \{ovar\}$

Figure 2.3: Scoping rules for case-expressions



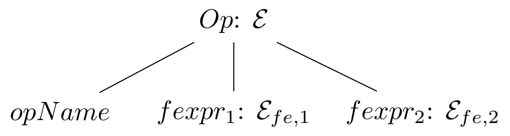
$\mathcal{E}_{ce} = \mathcal{E} \cup \{var_1, \ldots, var_n\}$

Figure 2.4: Scoping rule for the with-construct


$GAlt$: $\mathcal{E}$
$guard_1$: $\mathcal{E}_1$
$\ldots$
$Match_j$: $\mathcal{E}_j$
$\ldots$
$guard_n$: $\mathcal{E}_n$
$cexpr$: $\mathcal{E}_{ce}$
var_j
$c\ var_{j,1} \ldots var_{j,n_j}$: $\mathcal{E}_{match,j}$

$\mathcal{E}_{ce} = \bigcup_{i \in \{1,\ldots,n\}} \mathcal{E}_i$,
$var_j \in \mathcal{E}$ and $\mathcal{E}_j = \mathcal{E}_{match,j} = \mathcal{E} \cup \{var_{j,1}, \ldots, var_{j,n_j}\}$ for all match-constraints

Figure 2.5: Scoping rules for alternatives of guarded expressions


Op: $\mathcal{E}$
$opName$
$fexpr_1$: $\mathcal{E}_{fe,1}$
$fexpr_2$: $\mathcal{E}_{fe,2}$

$\mathcal{E}_{fe,1} = \mathcal{E}_{fe,2} = \mathcal{E}$,
$opName \in \mathcal{E}$

Figure 2.6: Scoping rule for infix operator applications

Int	Integer numbers $-2147483648, \ldots, -1, 0, 1, \ldots, 2147483647$
Float	Floating point numbers
Bool	Booleans *True* and *False*
C	*Success* and *Fail*

Figure 2.7: Built-in data types of CCFL

– all data types must have been defined in the user program (or are predefined),

– all data types are used with correct arity, and

– data type C and functional types do not appear.

2.3.2 Functions and constraints

A CCFL program must contain a declaration for every function (and constraint) definition. Furthermore, for all logical variables declared in with-constructs a type declaration must be given.

Figure 2.8 shows the types of all predefined functions and built-in constraints in CCFL-like notation, where we allow more than one function name per declaration (separated by spaces) unlike in the language itself.

```
fun  +  −  *  /  ::  Int -> Int -> Int
fun  +.  −.  *.  /.  ::  Float -> Float -> Float
fun  ~=  <=  >=  <  >  ==  ::  Int -> Int -> Bool
fun  ~=.  <=.  >=.  <.  >.  ==.  ::  Float -> Float -> Bool
fun  ~=:  ==:  &&  ||  ::  Bool -> Bool -> Bool
fun  not  ::  Bool -> Bool

fun  =:=  ::  a -> a -> C
fun  &  ::  C -> C -> C
fun  bound  ::  a -> C
```

Figure 2.8: Types of predefined functions and built-in constraints

The type check algorithm traverses the abstract syntax tree (*AST* in the following) of each defined function twice. In the first traversal it mainly collects type equations from the tree structure. Then the algorithm computes a most general unifier from the equations and applies it in the second tree traversal to build an updated AST.

Additional checks are performed during the type checking of functions including

- a test to ensure that the number of parameters of a function is less than or equal to the number specified by its type

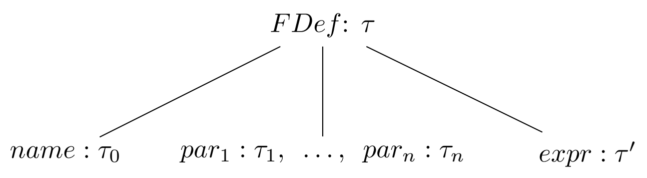
- tests for partial application of constructors in case branches and in match-constraints which are disallowed in both cases.

In the following we show typing rules resp. type equalities for CCFL constructs by annotated syntax trees.

Functions. The type of a function definition is, in general, a function type. The types of its arguments are the parameter types and the result type is the type of the body expression (see Figure 2.9).

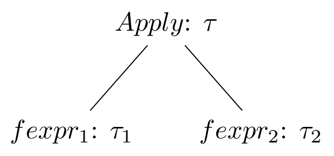
Type checking of *Expr* nodes just lifts type checking of their potential sub-nodes *FExpr*, *CExpr*, and *GExpr*. We consider rules for these alternatives in the following.

Functional expressions. Here, we left out infix operation applications, constants, constructors, and variables because their typing rules are straightforward. Instead we consider applications, local definitions, and case expressions. The rules are illustrated in the Figures 2.10 – 2.12.



$$\tau = \tau_0 = \tau_1 \to \ldots \to \tau_n \to \tau'$$

Figure 2.9: Typing rule for function definitions

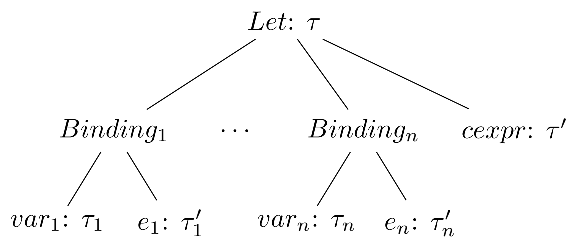


$$\tau_1 = \tau_2 \to \tau$$

Figure 2.10: Typing rule for function applications

The type of an *application* is the result type of its first functional expression. The argument type of this must match the type of the second functional expression.

The type of a *local definition,* i.e. a *let-expression*, is the type of its constraint expression. The types of the local variables must be the same like their defining expressions.



$$\begin{aligned} \tau &= \tau', \\ \tau_i &= \tau'_i \text{ for } i \in \{1,\ldots,n\} \end{aligned}$$

Figure 2.11: Typing rules for local definitions

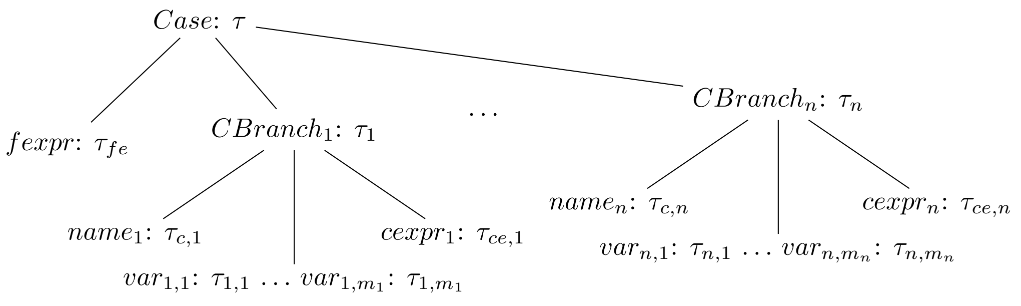
The type τ of a *case-expression* is equal to the types τ_i of its branches. Each of these has the type $\tau_{ce,i}$ of the constraint expression defining it. The functional expression which determines the choice of a particular alternative is of the same type τ_{fe} like the expressions $name_i\ var_{i,1}\ \ldots\ var_{i,m_i}$ matching it.

Since *OBranch* is optional, it is represented in gray color in Figure 2.12 again.

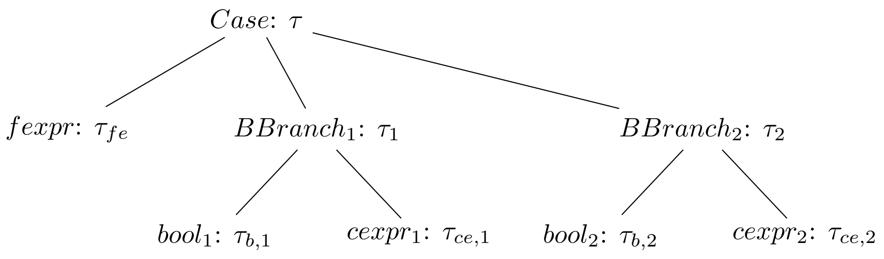
Constraint expressions. A constraint expression may (thus, represented in gray in Figure 2.13) be proceeded by the introduction of new logical variables using the keyword ***with***. These variables can be used within the atoms of the constraint expression.

The type τ' of a with-expression is the same as that of its conjunction of atoms. We need to consider two cases here: Either, we have an actual conjunction of two or more atoms. In this case, the atoms are constraints of type ***C***. Or, we have exactly one atom. Then this atom may be a functional expression of a type different from ***C***. The given equations take both cases into consideration.

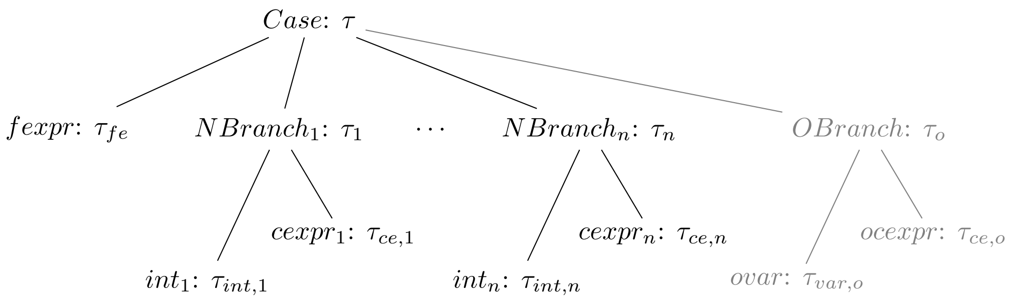
An atom may be an equality constraint (of type ***C***) or a functional expression. The typing rules are straightforward (see also Figure 2.8).


$Case\colon \tau$
$fexpr\colon \tau_{fe}$
$CBranch_1\colon \tau_1$
$\dots$
$CBranch_n\colon \tau_n$
$name_1\colon \tau_{c,1}$
$cexpr_1\colon \tau_{ce,1}$
$var_{1,1}\colon \tau_{1,1} \dots var_{1,m_1}\colon \tau_{1,m_1}$
$name_n\colon \tau_{c,n}$
$cexpr_n\colon \tau_{ce,n}$
$var_{n,1}\colon \tau_{n,1} \dots var_{n,m_n}\colon \tau_{n,m_n}$

$\tau \quad = \tau_i = \tau_{ce,i}$, $i \in \{1, \dots, n\}$,
$\tau_{c,i} = \tau_{i,1} \to \dots \to \tau_{i,m_i} \to \tau_{fe}$, $i \in \{1, \dots, n\}$

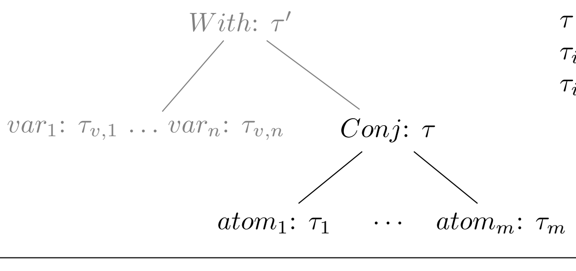

$Case\colon \tau$
$fexpr\colon \tau_{fe}$
$BBranch_1\colon \tau_1$
$BBranch_2\colon \tau_2$
$bool_1\colon \tau_{b,1}$
$cexpr_1\colon \tau_{ce,1}$
$bool_2\colon \tau_{b,2}$
$cexpr_2\colon \tau_{ce,2}$

$\tau \quad = \tau_1 \quad = \tau_2 \quad = \tau_{ce,1} = \tau_{ce,2}$
$\tau_{fe} = \tau_{b,1} = \tau_{b,2} = \boldsymbol{Bool}$


$Case\colon \tau$
$fexpr\colon \tau_{fe}$
$NBranch_1\colon \tau_1$
$\cdots$
$NBranch_n\colon \tau_n$
$OBranch\colon \tau_o$
$cexpr_1\colon \tau_{ce,1}$
$int_1\colon \tau_{int,1}$
$cexpr_n\colon \tau_{ce,n}$
$int_n\colon \tau_{int,n}$
$ocexpr\colon \tau_{ce,o}$
$ovar\colon \tau_{var,o}$

$\tau \quad = \tau_i \qquad = \tau_{ce,i}$, $i \in \{1, \dots, n, o\}$,
$\tau_{fe} = \tau_{int,j} = \tau_{var,o} = \boldsymbol{Int}$, $j \in \{1, \dots, n\}$

Figure 2.12: Typing rules for case-expressions



$\tau = \tau'$,
$\tau_i = \tau_{i+1} = \tau$ for all $i \in \{1, \dots, m-1\}$,
$\tau_i = C$ for $i > 1$

Figure 2.13: Typing rules for constraint expressions

Guarded expressions. Guarded alternatives, their guards and constraint expressions are all of type $\boldsymbol{C}$ (Figure 2.14). For the types of *ask*-constraints see Figure 2.8.

$$\begin{tikzcd}
 & & GAlt\colon \mathrm{C} \arrow[dll, no head] \arrow[d, no head] \arrow[dr, no head] & \\
guard_1\colon \mathrm{C} & \cdots & guard_n\colon \mathrm{C} & cexpr\colon \mathrm{C}
\end{tikzcd}$$

Figure 2.14: Typing rule for guarded expressions

2.4 Implementation

The checking of context conditions and the renaming of shadowing variables as discussed in Sections 2.1 and 2.2 are implemented in the program file *Checker.hs*. The module *TypeCheck.hs* realises type checking for CCFL programs as presented in Section 2.3.

Chapter 3

The target language LMNtal

In this chapter, we briefly introduce the target language LMNtal of the CCFL compiler, sketch on the abstract syntax representation of LMNtal code used in our implementation and discuss the LMNtal language concept of *links* in detail which turned out to strongly influence our implementation.

3.1 A very brief introduction to LMNtal

LMNtal [UK02, UK05, UKHM06, LMN07] is a concurrent graph rewriting language. Programs of this language describe *processes* which are multisets of atoms, cells, and rules. *Atoms* are one of the basic ingredients. They may be interconnected by logical *links* and build graphs in this way. *Rules* are used to describe the rewriting of graphs. A *cell* encloses a process, i.e. atoms, rules, and cells within a membrane and may encapsulate computations and express hierarchies. Besides atoms, *links* may interconnect as well cells and atoms with cells.

Program 3.1.1 shows two simple LMNtal rules for list concatenation and an atom *append* ([1,2],[4], *R*) to be reduced. LMNtal uses a PROLOG like syntax but there are fundamental differences. One important thing concerns logical links: What a PROLOG programmer may hold for logical variables in Program 3.1.1 are actually logical *links.* In LMNtal, links are restricted to occur exactly twice in a rule. This holds for Program 3.1.1, but of course not in general for PROLOG programs. Furthermore, there are other differences, e.g. LMNtal allows on the left-hand side of rules multisets of atoms, even cells including rules. This is expressed by patterns, so called *process templates.* LMNtal is a concurrent language and inherits properties from concurrent logic languages, for example guards and a non-deterministic rule choice. More complex examples of LMNtal programs are shown in Chapters 4 and 5, e.g. in the Programs 4.7.3, 4.7.4, and 5.2.1.

For a detailed description of LMNtal we refer to [UK02, UK05, UKHM06, LMN07]. We presuppose basic knowledge of LMNtal in the following. However, in Section 3.3, we will discuss aspects of logical links in LMNtal which are crucial to our implementation.

Program 3.1.1 List append in LMNtal

```
append([],Y,Z) :- Y = Z.
append([XH|XR],Y,Z) :- Z = [XH|ZR], append(XR,Y,ZR).

append([1,2],[4],R)
```

3.2 The abstract syntax of LMNtal

Program 3.2.1 shows the data types for the abstract representation of LMNtal programs as it is implemented in the file *LMNtalSyntax.hs*. Actually, we do not aim to represent every possible LMNtal program but instead just the subset which is potentially generated from CCFL programs.

An LMNtal program describes a *process* which is a multiset of atoms, cells, rules, and modules. We use *modules* to encapsulate rules to structure the generated programs. A *rule* consists of a rule head, a body, and a guard which is a multiset of so called type constraints. Rule heads and bodies are *process templates* describing a set of processes by patterns. *Cells*, in their most general form, encapsulate a process template. Additionally they may hold incoming and outgoing links. We use a so-called *stable flag* for the implementation of evaluation strategies. *Atoms* consist of a name and sub-atoms, mainly links represented by (*AVar linkname*) in our case. There are particular special atoms like connectors and operations.

3.3 CCFL variables and logical links in LMNtal

This section considers one concept of the language LMNtal more detailed: logical links. We discuss the correspondence and differences between variables in CCFL and links in LMNtal and the consequences for the code generation.

Program 3.1.1 illustrates that there is a narrow correspondence between LMNtal links and variables in declarative languages. Thus, it seems an obvious idea at first, to represent CCFL variables by LMNtal links. However, this point of view may become problematic. The reason is that links have a different intended meaning than variables.

CCFL uses declarative variables in the usual meaning, i.e. they stand for particular expressions resp. values. Once bound, CCFL variables stay bound throughout the computation and are indistinguishable from their value. LMNtal links as well connect to a structure or value. However, in general, they are used to interconnect two atoms, two cells, or an atom and a cell and they have, thus, (at most) two occurrences. In rules, logical links must occur exactly twice[1] [UK02, UK05]. Furthermore, link connections may change.

In our implementation, we examined two approaches to deal with the representation of CCFL variables in LMNtal. We discuss the general ideas in the following sections and consider the resulting compiler schemes in Chapters 4 and 5.

3.3.1 Functional CCFL programs and the copying of structures

The first approach is to persist in the viewpoint that links are just variables with certain restrictions. However, this means to accept a number of restrictions to CCFL.

LMNtal links establish connections. Thus, in a rule they must appear exactly twice. If there occurs a link once in the rule head and once in the body, then one can interpret it as variable which binding – a structure in the rule head – changes in a rewrite step into a new structure in the rule body. Since, moreover, LMNtal allows to copy (and delete) links in the body of a rule, if they occur once in the rule head, we can try to implement CCFL variables by means of LMNtal links.

In LMNtal, a structure without membranes, which is LMNtal-*ground*, i.e. it is a connected graph with exactly one free link, can be copied or deleted using the guard *ground*. Copying a structure containing links, however means, that the links are copied as well but they are not any more interconnected.

Example 3.3.1 Consider the following LMNtal process.

[1]Links may appear twice in the head and twice in the body [UK02], but this just "recycles" the link name and does not create "further entry points" into a link connection.

Program 3.2.1 Abstract representation of LMNtal programs

```
data Process =
  Process  { atoms    :: [LMNtalAtom],
             rules    :: [Rule],
             cells    :: [Cell],
             modules  :: [Module] }

data Module =
  Module   { moduleName :: String,
             moduleCtns :: [Process] }

data Rule =
  Rule     { ruleHead  :: ProcessTemplate,
             ruleGuard :: [LMNtalGuard],
             ruleBody  :: ProcessTemplate }

data ProcessTemplate =
  Template { templateAtoms :: [LMNtalAtom],
             ruleCtx :: [String],
             procCtx :: [ProcCtx],
             templateCells :: [Cell],
             templateRules :: [Rule] }

data Cell =
  Cell     { template :: ProcessTemplate,
             inLinks :: [InLink],
             outLinks :: [OutLink],
             stableTag :: Bool }

data InLink = InLink String
data OutLink = OutLink String

data LMNtalGuard =
  OpRelGuard  { ogname:: String,
                ogargs :: [LMNtalAtom] }
  | EqGuard   { eqgarg1 :: LMNtalAtom,
                eqgarg2 :: LMNtalAtom }
  | Ground String
  | GInt String
  | GFloat String
  | ...

data ProcCtx =
  ProcCtx     { name :: String,
                links :: [String] }

data LMNtalAtom =
  LMNtalAtom  { aname :: String,
                aargs :: [LMNtalAtom] }
  | Connector { arg1 :: LMNtalAtom,
                arg2 :: LMNtalAtom }
  | Operation { oname :: String,
                oargs :: [LMNtalAtom] }
  | AVar String
  | AInt Integer
  | AFloat Double
  | ...
```

```
f(A) :- ground(A) | g(A,A).
f(1), f(h(B))
```

The atoms reduce into the following.

```
g(1,1), g(h(B),h(B))
```

While this looks fine for our purposes at the first sight, actually, LMNtal completely copies the concerning structures including the link B. That is, there are finally two different links with name B in the atom $g(h(B),h(B))$.

Consider the following simple LMNtal program:

```
f(A) :- ground(A) | g(A), g(A), g(A).
g(A) :- A = 1.
g(A) :- A = 2.
f(B)
```

The atom $f(B)$ may e.g. reduce into the atoms B=1, B=2, B=1 (or others depending on the rule choice), which more clearly demonstrates the generation of link copies.

What follows is, that it is only safe to copy structures which are *ground* in the usual meaning of functional languages, i.e. structures which do not contain variables resp. free links. Thus, using the approach to represent variables by links, we may transform *functional* CCFL programs into LMNtal.

Note, that, in LMNtal, using the *ground* guard for copying is only possible for *structures without membranes.* To copy *structures containing membranes* LMNtal provides an API *nlmem.copy* which however is much more intricate and expensive and must be applied as many times as copies are needed.

We discuss the implementation of functional CCFL with the approach to represent CCFL variables by LMNtal links in Chapter 4.

3.3.2 Reaching full CCFL

Representing CCFL variables by LMNtal links limits us to functional CCFL programs. The restriction that links must occur exactly twice in a rule, yields to the necessity of copying which finally results in the situation that we can only deal with structures which are ground in the sense of functional languages.

This prevents not only the usage of free variables which are needed to implement CCFL constraint and guarded expressions, but also the realisation of a lazy evaluation strategy for functional CCFL since this demands a sharing (in contrast of copying) of structures.

Thus, to implement full CCFL we need a more sophisticated representation of variables by means of heap graph structures which are connected by links to the variable representations. We consider the implementation of CCFL based on this approach in Chapter 5.

Chapter 4

Compiling CCFL into LMNtal – a naive approach

The general structure of our CCFL compiler is shown in Figure 4.1, where we annotated the according sections of this report in parentheses. We sketched on the syntactic analysis which partitions the program into its elements, like keywords, variables, operators, numbers etc., and builds from these an abstract syntax tree (AST) representing the semantic components of the program in Section 1.2. In Chapter 2, we discussed the semantic analysis which verifies resp. computes context conditions and type information and augments in this way the AST.

While the syntactic and symbolic analyses build the so-called front-end, the generation of LMNtal code constitutes the compiler back-end. In a first step, we generate basic LMNtal code from the AST subtrees for the semantic program components, where we map CCFL functions to LMNtal rules, function applications, data terms, and guards to LMNtal atoms and guards and so on. Furthermore, we generate an additional rule set per function to realise the handling of higher-order functions and partial applications. CCFL variables are directly translated into LMNtal links. We discuss the transformations which are partially based on translation techniques [Nai91, Nai96, War82, CvER90] for functional into logic languages and the resulting code for a functional sub-language of CCFL in Sections 4.1–4.6. Since LMNtal does not fix a priori a particular evaluation strategy the generated LMNtal programs are evaluated non-deterministically.

In Section 4.7 we describe additional transformations on the generated LMNtal ASTs which allow the realisation of a call-by-value and a call-by-name evaluation strategy for CCFL programs by encapsulating computations in membranes.

By means of an alternative transformation scheme (discussed in Chapter 5) we finally allow the compilation of programs encompassing the complete functionality of CCFL including free variables, constraints and lazy evaluation. This is realised by a representation of CCFL variables by links on structures of a heap as runtime environment.

4.1 The compilation scheme for functional CCFL

Our naive approach for the transformation of CCFL programs into LMNtal code only applies to the functional sub-language of CCFL including all elements of functional expressions but excluding guarded and constraint expressions. This is caused by the representation of CCFL variables by LMNtal links and the restrictions on logical links in the target language LMNtal (as discussed in Section 3.3.1).

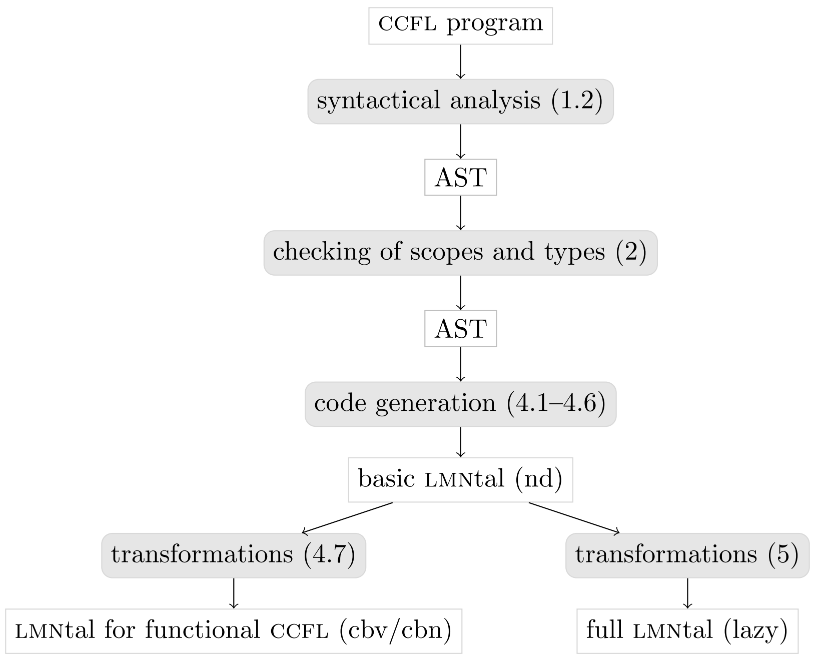

CCFL program
syntactical analysis (1.2)
AST
checking of scopes and types (2)
AST
code generation (4.1–4.6)
basic LMNtal (nd)
transformations (4.7)
transformations (5)
LMNtal for functional CCFL (cbv/cbn)
full LMNtal (lazy)

Figure 4.1: Overview of the CCFL compiler

4.1.1 Function definitions

A program is translated by considering each of its function definitions. Data type definitions and function declarations are used for type and context checking only (see Chapter 2). A CCFL function definition is translated into a set of LMNtal rules.

The transformation is implemented by means of a function *transform*. This function takes a CCFL AST (i.e. an element of type class *Transform*), a string to represent the link to access the functions result, and an already generated LMNtal rule. It computes, in general, a set of LMNtal rules.

The implementation of the function *transform* is monadic; it has an internal state (hidden within the monad *TR*) consisting of a map of all function and constructor declarations (i.e. the global environment) and a map associating every currently known variable with its LMNtal counterpart. Additionally, this state allows the generation of fresh links.

```
class Transform a where
    transform :: a -> String -> Rule -> TR [Rule]
```

The following problems are not considered in this section, but they are dealt with in proceeding and subsequent subsections resp.:

- higher-order functions, η-enrichment, and partial applications (Sections 4.2, 4.3, and 4.4),
- predefined functions (Section 4.1.3) and renamings (Section 4.5),
- restrictions on link occurrences in LMNtal (Sections 3.3 and 4.6), and
- evaluation strategies (Section 4.7).

Concerning the third item, LMNtal restricts links to occur exactly twice in a rule. While this may be natural for the representation of graphs in LMNtal, it complicates our transformations. However, initially we will consider LMNtal links just like variables (without occurrence restrictions) and use them to represent CCFL variables. This is also reflected by the abstract representation of links as "variables" (*AVar linkname*) in Program 3.2.1. We discussed the link restrictions, their effects, and how to solve these problems in Section 3.3.

In the following we sketch on the CCFL to LMNtal transformation and show finally generated code by means of examples.

A function definition itself has no result and, thus, its translation is called with an empty result link.

```
1  transform (FDef name pars expr) _ rule =
2    do -- build new result link
3       result <- ...
4       -- build rule "name(pars, result) :- | ."
5       let newRule = ... rule pars result ...
6       -- compute rule set from expr, result, and newRule
7       rules <- transform expr result newRule
8       return rules
```

An LMNtal rule for a function definition needs an additional link *result* (line 3) in the head to link to the result of the function. Since the initial call of *transform* on a function definition comes with an empty rule, the guard and the body of the rule are empty too (line 5). They are computed in the next step from the CCFL rule body expression (line 7).

Example 4.1.1 The transformation of a CCFL function *filter* with

```
fun filter :: (a -> Bool) -> (List a) -> (List a)
def filter pred list = >>expr<<
```

yields a set of rules computed by (*transform expr V_0 newRule*), where *V_0* is the new result link and the LMNtal rule *newRule* looks currently like this:

```
filter (Pred, List, V_0) :- | .
```

4.1.2 Functional expressions

For functional CCFL we need to consider functional expressions only. Constraint expressions and guarded expressions are discussed in Section 5.3, where we reach a translation for full CCFL.

- **Basic infix operations** are handled straightforward (the code is given below): First the sub-expressions are transformed yielding new LMNtal rule sets (lines 5 and 6). Then we build an atom expressing the basic infix operation in LMNtal (line 9), add it to the current rule body (line 10) and return the computed rule set (line 12).

  In a set of generated rules always the first rule carries the current function transformation information. Thus the transformation of *fexpr2* (line 6) is done on top of the first result rule *ruleFe1* of the transformation of *fexpr1*. Likewise the new LMNtal atom is added to the first rule *ruleFe2* originating from the translation of *fexpr2* (line 10).

```
1  transform (Op opName fexpr1 fexpr2) result rule =
2    do ...
3       -- generate rule sets from fexpr1 and fexpr2
4       -- with new result links linkFe1 and linkFe2
5       (ruleFe1 : rulesFe1) <- transform fexpr1 linkFe1 rule
6       (ruleFe2 : rulesFe2) <- transform fexpr2 linkFe2 ruleFe1
7       -- extend rule body by
8       --    atom "result = linkFe1 opName linkFe2"
9       bodyAtom = ...
```

```
        infixRule <- addAtomToRuleBody bodyAtom ruleFe2
        -- return generated rules
        return (infixRule : (rulesFe1 ++ rulesFe2))
```

Example 4.1.2 The arithmetic function *succ*

```
fun succ :: Int -> Int
def succ a = a + 1
```

compiles into the following rule, where the first atom of the right hand side results from the infix operator application $a + 1$.

```
succ (A, V_0) :-
  V_0 = V_1 + V_2,
  V_2 = 1,
  V_1 = A .
```

- **Constants, variables, and constructors** just generate connector atoms in LMNtal, an example is given by the second and third body atoms of the generated rule in Example 4.1.2.

```
transform (Const (Int i)) result rule =
  do -- build an atom "result = i"
     let bodyAtom = Connector (AVar result) (AInt i)
     -- extend rule body by bodyAtom
     let newRule = addAtomToRuleBody bodyAtom rule
     -- return generated rule
     return [newRule]

transform (FunVar (Var vName)) result rule =
  do -- build a connector atom "result = vName"
     -- and extend rule body by this atom
     ...

transform (Constr c) result rule =
  do -- build a connector atom "result = c"
     -- and extend rule body by this atom
     ...
```

- **Function applications** are translated by the treatment of their sub-expressions, the results of which are combined again.[1]

```
transform (Apply fexpr1 fexpr2) result rule =
  do -- flatten Apply tree into uncurried LMNtal style
     let (FlatApplyTree fexpr args) =
                          flattenApply (Apply fexpr1 fexpr2)
     ...
     -- build an atom "app_(nameLink, argLinks, result)"
     --   with new links nameLink and argLinks for name
     --   and arguments and add it to the current rule
     let applyAtom = ...
     let newRule = addAtomToRuleBody applyAtom rule
     ...
```

[1]For the application we use an atom name *app_*, such that the underbar "_" distinguishes it from valid function names.

```
12        -- generate rules from fexpr and args
13        (feRule : feRules) <- transform fexpr nameLink newRule
14        applyRules <-
15            foldM ... transform ... args argLinks feRule ...
16        ...
17        -- return generated rules
18        return (applyRules ++ feRules)
```

Example 4.1.3 Consider the definition of the CCFL function *foo*.

```
fun foo :: Int -> Int -> Int
def foo a b =
    bar a 2 3 (baz 4 b)
```

Its body expression is represented by the abstract syntax tree of Figure 4.2(a). The long chains of *Apply* nodes result from the currying of functions in CCFL. Since atoms in LMNtal are not curried, we flatten the chains of *Apply* nodes when generating LMNtal code from a CCFL AST (line 4). This yields the tree given in Figure 4.2(b) for our example. Finally, we receive the following resulting LMNtal code.

```
foo (A,B,V_0) :-
    V_7 = B,
    V_6 = 4,
    V_8 = baz,
    app_ (V_8,V_6,V_7,V_4),
    V_3 = 3,
    V_2 = 2,
    V_1 = A,
    V_5 = bar,
    app_ (V_5,V_1,V_2,V_3,V_4,V_0) .
```

- **Let-expressions** (***let*** *var* = *fexpr* ***in*** *cexpr*) are transformed into LMNtal atoms. We show this just by a simple example.

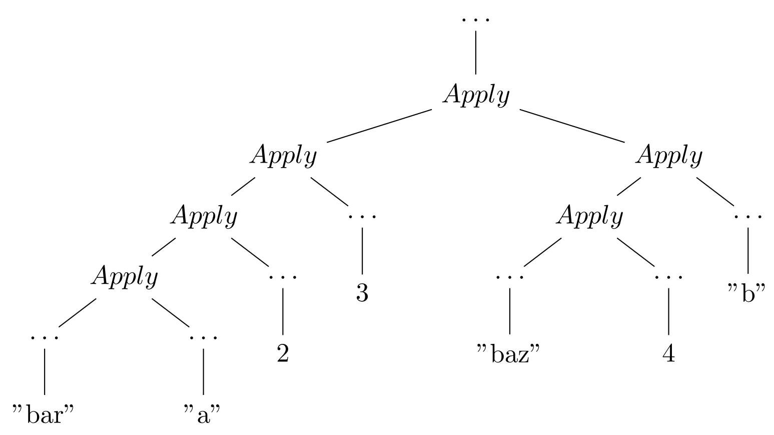
  Example 4.1.4 The CCFL function *flet* with

```
fun flet :: Int -> Int
def flet a =
    let b = (a + 9)
    in b * 2
```

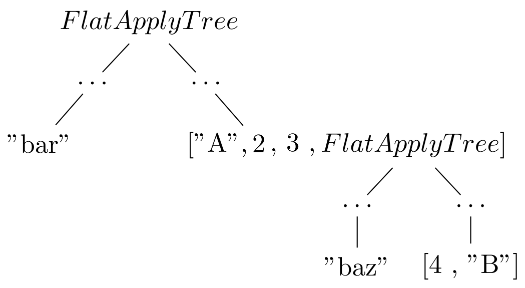
  compiles into the LMNtal rule

```
flet (A,V_0) :-
      V_0 = V_3 * V_4,
      V_4 = 2,
      V_3 = B,
      B = V_1 + V_2,
      V_2 = 9,
      V_1 = A .
```

- **Case-expressions** (***case*** *fexpr* ***of*** ...) are split up into distinct rules for each alternative. First, we transform the functional expression *fexpr* on which the appropriate alternative is chosen (line 4). In a second step, we build rules for each alternative (lines 11–14). Finally, an atom which matches the rule heads of these rules is added to the current rule (line 17).



(a) Chain of *Apply* nodes as CCFL AST ...



(b) ... and its (intermediate) representation in LMNtal

Figure 4.2: Abstract representation of function applications

```
 1 transform (Case fexpr branches) result rule =
 2   do ...
 3      -- generate rules from fexpr and a new link fexprLink
 4      (fexprRule : fexprRules) <- transform fexpr fexprLink rule
 5      -- build atom
 6      --     "case__ruleName(caseLink,allKnownLinks,result)" ...
 7      caseCall = ...
 8      -- and a new rule "caseCall :- | .
 9      newRule = addAtomToRuleHead caseCall emptyRule
10      -- generate list of rules from branches and newRule
11      branchRules <- ...
12        mapM
13          (\ branch -> transform branch result newRule)
14          branches
15      ...
16      -- add caseCall to current rule body
17      let caseRule = addAtomToRuleBody caseCall fexprRule
18      -- return new rule set
19      return ( caseRule : (fexprRules ++ branchRules))
```

This is again best illustrated by means of an example.

Example 4.1.5 The CCFL function *flist* takes a list and returns parts of it depending on the list structure.

```
fun flist :: List a -> List a
def flist a =
   case a of Nil -> a ;
             Cons b d -> case d of Nil -> a ;
                                   Cons e f -> d
```

This function is compiled into a number of rules. The initial rule just rewrites an atom *flist* (*A*,*V_0*) into an atom *case__flist* (*V_1*,*A*,*V_0*) (the *caseCall*, line 7). Its first link *V_1* represents the expression, i.e. the variable *a*, on which the appropriate case-alternative is chosen. This gets obvious by the heads of the second and third rules which perform a matching on *V_1* and whose bodies contain code generated from the different alternatives.

Additionally, the example illustrates a nested case-expression. Here we see, why we need to pass over all known variables resp. links in a "*caseCall*" (line 7). E.g. in the second case alternative of the CCFL function *flist* (and the fourth rule in the generated LMNtal code) the result is *a* (resp. *A*) which is not used in the distinguishing functional expression of this case-expression directly but instead known from before.

```
flist (A, V_0)  :-
   case__flist (V_1,A, V_0),
   V_1 = A .

case__flist (V_1,A, V_0),
nil (V_1)  :-
     V_0 = A .

case__flist (V_1,A, V_0),
cons (B,D, V_1)  :-
   case__case__flist (V_2,A,B,D, V_0),
   V_2 = D .
```

```
    case__case__flist (V_2,A,B,D, V_0),
    nil (V_2)   :-
      V_0 = A .

    case__case__flist (V_2,A,B,D, V_0),
    cons (E,F, V_2)   :-
      V_0 = D .
```

The compilation of a case-expression which alternatives are distinguished by natural numbers (i.e. *NBranch* alternatives) is realised by adding corresponding guards instead of head atoms to the LMNtal rules. Example 4.6.3 in Section 4.6 shows the compilation result for the *fibonacci* function definition of Program 1.1.1.

4.1.3 The prelude

Besides the actual program rules, we generate a prelude which provides rules for handling CCFL built-in functions (mainly based on LMNtal built-in operations).

4.2 Higher-order functions

To allow the usage of higher-order functions in CCFL we use a transformation scheme from [War82]. For every CCFL function (*f x1 ... xn = expr*) we generate a rewrite rule (*app_(f,x1 ,..., xn) :− f(x1 ,..., xn)*).

Example 4.2.1 Given the following CCFL program ...

```
fun goo :: Int -> Int
def goo x = x + 2

fun foo :: (Int -> Int) -> Int -> Int
def foo function value = function (value + 1)
```

... we obtain the LMNtal rules:

```
goo (X, V_5)   :-
  V_5 = V_6 + V_7,
  V_7 = 2,
  V_6 = X .

foo (Function, Value, V_8)   :-
  V_9 = V_11 + V_12,
  V_12 = 1,
  V_11 = Value,
  V_10 = Function,
  app_ (V_10, V_9, V_8) .

app_ (goo, V_9, V_10)   :-
  goo (V_9, V_10) .

app_ (foo, V_15, V_16, V_17)   :-
    foo (V_15, V_16, V_17) .
```

Notice, that the atom *app_(V_10,V_9,V_8)* in the last line of the *foo* rule has already been generated from the CCFL AST node (*Apply* ...) of the function application (*function* (*value* + 1)).

We show a derivation of the functional expression *foo goo* 3 and a corresponding derivation of the according LMNtal atom *foo*(*goo*,3,*Z*).[2]

A CCFL derivation:

foo goo 3 $\rightarrow$ *goo* (3+1) $\rightarrow$ *goo* 4 $\rightarrow$ 4 + 2 $\rightarrow$ 6

A corresponding LMNtal derivation:

foo(*goo*,3,*Z*) $\rightsquigarrow$ *app_*(*goo*,3+1,*Z*) $\rightsquigarrow$ *goo*(3+1,*Z*) $\rightsquigarrow$ *goo*(4,*Z*) $\rightsquigarrow$ *Z* = 4+2 $\rightsquigarrow$ *Z* = 6

To show a more complex example, we add a function *soo* to the CCFL program which takes three arguments: a function *hof* which is a higher-order function itself, a function *f* and a value *v* and just applies *hof* on *f* and *v*.

```
fun soo ::
  ((a -> a) -> a -> a) -> (a -> a) -> a -> a
def soo hof f v = hof f v
```

Function *soo* is translated into LMNtal code according to the scheme given in the Sections 4.1.1 and 4.1.2.

```
soo (Hof,F,V,V_0)  :-
  V_2 = V,
  V_1 = F,
  V_3 = Hof,
  app_ (V_3, V_1, V_2, V_0) .
```

Derivation sequences for the functional expression *soo foo goo* 3 and the corresponding LMNtal atom *soo*(*foo*,*goo*,3,*Z*) demonstrate again the use of higher-order functions:

A CCFL derivation:

soo foo goo 3 $\rightarrow$ *foo goo* 3 $\rightarrow$ ... (as above)

A corresponding LMNtal derivation:

soo(*foo*,*goo*,3,*Z*) $\rightsquigarrow$ *app_*(*foo*,*goo*,3,*Z*) $\rightsquigarrow$ *foo*(*goo*,3,*Z*) $\rightsquigarrow$ ... (as above)

4.3 η-enrichment

To simplify the generation of rules for the partial applications of functions, the CCFL compiler performs an η-enrichment of functions. This just means that additional arguments are appended to the left-hand sides and to the right-hand sides of function definitions according to the function type declaration.

Example 4.3.1 Consider the CCFL function *addOne*. According to its type declaration, it takes one argument of type ***Int*** which is, however, left out in the function definition.

```
fun addOne :: Int -> Int
def addOne = add 1
```

The η-enrichment of this function within the compiler yields (internally) the following definition which is used in the subsequent compilation process.

```
def addOne x = (add 1) x
```

[2]Currently no evaluation strategy is fixed; we just consider one possible derivation here. The encoding of evaluation strategies is discussed in Section 4.7.

4.4 Partial applications

The partial application of functions is realised by a number of additional LMNtal rules per CCFL function, where we apply a transformation given in [Nai91]. We start with an example.

Example 4.4.1 Let *add* and *addOne* be functions, where the latter is defined by a partial application of the former.

```
fun add :: Int -> Int -> Int
def add a b = a + b

fun addOne :: Int -> Int
def addOne = let x = add 1 in x
```

Function *addOne* is transformed by η-enrichment into

```
def addOne y = (let x = add 1 in x) y
```

A CCFL derivation sequence is the following:

$addOne\ 2 \rightarrow (add\ 1)\ 2 \rightarrow 1 + 2 \rightarrow 3$

The functions *add* and *addOne* are translated to LMNtal rules using our translation scheme. In the next step, we generate additional rules for each function for all possible cases of its partial application. These rules just generate constructor terms of the function name and allow in this way to keep the data and to suspend the computation until the function can be fully applied.[3]

```
add (A,B, V_0)  :-
   V_0 = A + B.

addOne ( V_3, V_4)  :-
   app_ (add,1 ,X),
   app_ (X, V_3, V_4) .

/* handling add resp. (V_5 = add) */
app_ (add, V_5)  :-  add ( V_5) .
/* handling (add V_0) resp. (V_1 = add V_0) */
app_ (add, V_0, V_1)  :- add ( V_0, V_1) .
/* handling ((add V_2) V_3) resp. (V_4 = (add V_2) V_3) */
app_ (add ( V_2), V_3, V_4)  :-  add ( V_2, V_3, V_4) .
/* handling add as hof */
app_ (add, V_6, V_7, V_8)  :-  add ( V_6, V_7, V_8) .

/* handling addOne resp. (V_11 = addOne) */
app_ (addOne, V_11)  :- addOne ( V_11) .
/* handling addOne as hof */
app_ (addOne, V_9, V_10)  :-  addOne ( V_9, V_10) .
```

Using these LMNtal rules we rewrite the atom *addOne*(2,*R*) which corresponds to the CCFL expression *addOne* 2:

addOne(2,*R*)

$\rightsquigarrow$ *app_*(*add*,1,*X*), *app_*(*X*,2,*R*)

$\rightsquigarrow$ *add*(1,*X*), *app_*(*X*,2,*R*) $\equiv$ *app_*(*add*(1),2,*R*)

$\rightsquigarrow$ *add*(1,2,*R*)

[3]For better readability we inline link bindings here.

Program 4.4.1 General scheme for the generation of rules for n-ary CCFL functions for the handling of partial applications and higher-order functions

Given the CCFL function f, where $a \neq \boldsymbol{C}$, with

```
fun f :: a_1 -> ... -> a_n -> a
def f x_1 ... x_n = expr
```

we generate the following LMNtal rules:

app_(f,X_1) :− f(X_1).	(1)
app_(f,X_1,X_2) :− f(X_1,X_2).	(2)
...	...
app_(f,X_1 ,..., X_n) :− f(X_1 ,..., X_n).	(n)
app_(f(X_1),X_2,X_3) :− f(X_1,X_2,X_3).	$(n+1)$
...	...
app_(f(X_1),..., X_n) :− f(X_1 ,..., X_n).	$(n+(n-2))$
app_(f(X_1),..., X_n,R) :− f(X_1 ,..., X_n,R).	$(n+(n-1))$
app_(f(X_1,X_2),X_3,X_4) :− f(X_1,X_2,X_3,X_4).	$(n+(n-1)+1)$
...	...
app_(f(X_1,X_2),..., X_n,R) :− f(X_1 ,..., X_n,R).	$(n+(n-1)+(n-2))$
...	...
app_(f(X_1 ,..., X_n−1),X_n,R) :− f(X_1,...,X_n−1,X_n,R).	$(\frac{n\times(n+1)}{2})$

$\rightsquigarrow R = 1{+}2$

$\rightsquigarrow R = 3$

The general scheme for rule generation to enable the partial application of functions (including the handling of higher-order functions) generates $\frac{n\times(n+1)}{2}$ rules for every n-ary function. Program 4.4.1 shows the generation scheme.

The following example is a bit more complex and contains also higher-order functions.

Example 4.4.2 Consider the functions *add* and *addOne* from Example 4.4.1 and the two functions defined as below:

```
fun foo ::
  (a -> b) -> (b -> b -> c) -> a -> a -> c
def foo f g x y = g (f x) (f y)

fun bar :: Int -> Int -> Int
def bar = foo addOne add
```

We show two example derivations using a call-by-value evaluation strategy for CCFL.

bar 2 3

$\rightarrow$ *foo addOne add* 2 3

$\rightarrow$ *add* (*addOne* 2) (*addOne* 3)

$\rightarrow^\star$ *add* 3 4

$\rightarrow^\star$ 7

bar 1

$\rightarrow$ *foo addOne add* 1

We compile the functions *foo* and *bar* according to our scheme including the rules of Program 4.4.1 and receive amongst others the following rules:

```
foo (F, G, X, Y, V_9)  :-
   app_ (F, Y, V_11),
   app_ (F, X, V_10),
   app_ (G, V_10, V_11, V_9) .

bar (V_17, V_18, V_19)  :-
   app_ (foo, addOne, add, V_22),
   app_ (V_22, V_17, V_18, V_19) .

app_ (foo, X_1, X_2, X_3) :- foo (X_1, X_2, X_3).
app_ (foo(X_1, X_2), X_3, X_4, R) :- foo (X_1, X_2, X_3, X_4, R).
...
app_ (bar, X_1, X_2) :- bar (X_1, X_2).
...
```

The following LMNtal derivations reflect the functional call-by-value CCFL derivations as given above, where the bold lines represent the intermediate results of the CCFL derivation steps. However note, that without a certain fixed evaluation strategy it is not necessarily the case that we always reach corresponding states within an LMNtal derivation.

$\boldsymbol{bar(2,3,X)}$
$\rightsquigarrow$ $app\_(foo, addOne, add, V\_22),\ app\_(V\_22, 2, 3, X)$
$\rightsquigarrow$ $foo(addOne, add, V\_22),\ app\_(V\_22, 2, 3, X)$
$\quad\equiv$ $app\_(foo(addOne, add), 2, 3, X)$
$\rightsquigarrow$ $\boldsymbol{foo(addOne, add, 2, 3, X)}$
$\rightsquigarrow$ $app\_(addOne, 3, V\_11),\ app\_(addOne, 2, V\_10),\ app\_(add, V\_10, V\_11, V\_9)$
$\quad\equiv$ $app\_(add, app\_(addOne, 2), app\_(addOne, 3), X)$
$\rightsquigarrow$ $app\_(add, app\_(addOne, 2), addOne(3), X)$
$\rightsquigarrow$ $app\_(add, addOne(2), addOne(3), X)$
$\rightsquigarrow$ $\boldsymbol{add(addOne(2), addOne(3), X)}$
$\rightsquigarrow$ $add(addOne(2), app\_(app\_(add, 1), 3), X)$
$\rightsquigarrow$ $add(addOne(2), app\_(add(1), 3), X)$
$\rightsquigarrow$ $add(addOne(2), add(1, 3), X)$
$\rightsquigarrow$ $add(addOne(2), 1+3, X)$
$\rightsquigarrow$ $add(addOne(2), 4, X)$
$\rightsquigarrow^{\star}$ $\boldsymbol{add(3,4,X)}$
$\rightsquigarrow$ $X = 3 + 4$
$\rightsquigarrow$ $\boldsymbol{X = 7}$

$\boldsymbol{app\_(bar, 1, Y)}$
$\rightsquigarrow$ $bar(1, Y)$ (the derivation suspends)

4.5 Renamings

The compiler performs a number of renamings according to the syntactical restrictions of CCFL and LMNtal.

Variables in CCFL start with a lower letter as typical for functional languages. Since we represent variables currently by links which start with an upper letter in LMNtal, the compiler performs an according renaming.

Similarly, data constructors from CCFL must be renamed since they start with an upper letter here while they are represented by atom names in LMNtal starting with a lower letter.

4.6 Representing CCFL variables by LMNtal links

Up to now, we represented CCFL variables just by LMNtal links regardless of their occurrence restrictions. In the implementation, however, they must, of course, be taken into consideration.

In LMNtal, links must occur exactly twice in a rule which is currently not the case in general for the code generated by our scheme. However, the language allows to copy and delete links (resp. their connected structures) if the concerning links have been introduced by the rule head. In the following, we briefly point out that this property is always ensured for functional CCFL and show how copying of links is realised then in our implementation.

4.6.1 New variables in CCFL rule bodies

For functional CCFL we only need to consider functional expressions in rule bodies. Thus, the variables on their right-hand sides are always introduced by the function heads *except* for two language constructs, namely let expressions and constructor branches and otherwise-branches in case expressions. However, they are actually unproblematic too.

A let-expression (*let x = fexpr in cexpr*) introduces a fresh variable x. Since x can be used multiple times in the expression *cexpr* following the keyword ***in***, we do not meet the link occurrence restrictions in LMNtal. This problem can be solved, however, by generating intermediate rules as illustrated by the following example.

Example 4.6.1 Consider the arithmetic function *foo*.

```
fun foo :: Int -> Int
def foo n =
  let a = n * n ;
      b = a + n / 2
  in a * (a - 1) + b
```

A naive compilation would yield a CCFL rule with four occurrences of link A on its right-hand side which is non-conform with the restrictions on links in LMNtal.

```
foo (N, V_0) :-
  A = N * N,
  B = A + N / 2,
  V_0 = A * (A - 1) + B.
```

Thus, we generate intermediate LMNtal rules such that each rule just contains exactly one equation originating from one local definition and a call to a continuing rule. This ensures that every newly introduced link is used exactly twice per rule. We gain the following LMNtal program:

```
foo (N, V_0)  :-
  A = N * N,
  let__foo (A,N, V_0).

let__foo (A,N, V_0)  :-
  B = A + N / 2,
  let__let__foo (B,A,N, V_0).

let__let__foo (B,A,N, V_0)  :-
```

```
  V_0 = (A * (A - 1)) + B.
```

Case-expressions are the second type of construct which may introduce fresh variables on the right-hand side of CCFL rules. This concerns variables in constructor patterns and the pattern variable of otherwise-branches. However, again this is unproblematic as demonstrated by the Examples 4.6.2 and 4.6.3.

Example 4.6.2 The CCFL function *area* computes areas of different geometric shapes. The particular formula is selected by means of a case-expression using constructor patterns for the alternatives *Circle*, *Rectangle*, and *Square*.

```
data Shape = Circle Float
           | Rectangle Float Float
           | Square Float

fun area :: Shape -> Float
def area shape =
  case shape of
    Circle radius ->
      3.14 *. radius *. radius ;
    Rectangle x y ->
      x *. y ;
    Square x ->
      x *. x
```

This is compiled into the following LMNtal rules:

```
area (Shape, V_0) :-
  case__area (V_1, Shape, V_0),
  V_1 = Shape .

case__area (V_1, Shape, V_0),
circle (Radius, V_1) :-
  V_0 = 3.14 *. Radius *. Radius.

case__area (V_1, Shape, V_0),
rectangle (X, Y, V_1) :-
  V_0 = X *. Y.

case__area (V_1, Shape, V_0),
square (X, V_1) :-
  V_0 = X *. X.
```

A case-expression on constructor patterns, like (***case*** *shape* ***of*** *Circle radius ...*) introduces one new rule per alternative. Each rule has two atoms on the left-hand side: the atom *case__area* (*V_1*,*Shape*,*V_0*) which is the entry point into the case-expression and an atom to represent the matching on the constructor pattern, e.g. *circle* (*Radius*, *V_1*). They are connected via the link *V_1* which always exactly occurs twice and, thus, satisfies the LMNtal link restrictions.

Furthermore, the atoms for pattern matching, like *circle* (*Radius*, *V_1*) may introduce fresh links, but they are introduced on the left-hand sides of the rules and occur there exactly once which is guaranteed by the CCFL syntax of case-expressions.

The second example concerns the introduction of new variables by otherwise-branches. Recall, that otherwise-branches are only allowed in case-expressions whose alternatives are distinguished by natural numbers.

Example 4.6.3 Consider the *fibonacci* function as defined in Program 1.1.1. The pattern variable m in the otherwise-branch just binds the remaining possible alternative values and takes them over into the right-hand side of the rule. Since LMNtal does not fix a certain rule choice strategy, we add guards which exclude the cases which are already handled by alternative rules. Thus, the link M appears only once in the rule head (and possibly multiply in the rule body).

```
fibonacci (N, V_0)   :-
   case__fibonacci ( V_1,N, V_0),
   V_1 = N .

case__fibonacci ( V_1,N, V_0)   :-
V_1 =:= 0  |
   V_0 = 0  .

case__fibonacci ( V_1,N, V_0)   :-
V_1 =:= 1  |
   V_0 = 1  .

case__fibonacci (M,N, V_0)   :-
M =\= 1,
M =\= 0  |
   let__case__fibonacci (F1,M,N, V_0),
   app_ (fibonacci ,M-1,F1)  .

...
```

4.6.2 Copying LMNtal links and structures

While in LMNtal links must occur exactly twice in a rule, for the presented code generation scheme this property is quiet often not satisfied. However, adding a *ground* guard for the concerning links to the rule, LMNtal allows to copy or delete the attached structures. This is possible provided that the link was introduced by the rule head before. We discussed the satisfaction of this condition in the previous section.

The following example shows, how to add *ground* guards to LMNtal rules generated by our scheme to obtain valid LMNtal code.

Example 4.6.4 The addition of *ground* guards to the LMNtal rules of Example 4.6.1 yields valid LMNtal code.

```
foo (N, V_0)   :-
ground (N)  |
   A = N * N,
   let__foo (A,N, V_0).

let__foo (A,N, V_0)   :-
ground (A), ground (N)  |
   B = A + N / 2,
   let__let__foo (B,A,N, V_0).

let__let__foo (B,A,N, V_0)   :-
ground (A), ground (N)  |
   V_0 = (A * (A - 1)) + B.
```

Program 4.7.1 A CCFL program: *foo*, *goo*, *hoo*

```
fun foo :: Int -> Int -> Int
def foo a b = a + b
fun goo :: Int -> Int
def goo a = a + 3
fun hoo :: Int -> Int
def hoo a = goo (goo a)
```

Program 4.7.2 The program *foo*, *goo*, *hoo* compiled into LMNtal

```
foo (A,B,Z) :- Z = A + B .
goo (A,Z) :- Z = A + 3 .
hoo (A,Z)  :- goo(A,R), goo(R,Z) .

hoo(2,Y), goo(1,X), foo(X,Y,Z)
```

4.7 Evaluation strategies

Since LMNtal does not a priori support certain redex selection strategies, these must be encoded in the generated LMNtal rules.

4.7.1 Call-by-value evaluation

A call-by-value evaluation strategy for (functional) CCFL programs can be incorporated into the generated LMNtal program by a subtle mechanism using membranes and the stable flag "/" (see below). The main idea is to delay computations by holding the atoms representing outer calls apart from the rules in extra membranes as long as there are inner reducible expressions (inner *redexes* in the following).

Consider the CCFL Program 4.7.1 and a functional expression *foo* (*goo* 1) (*hoo* 2). Since we want to allow a concurrent reduction of independent sub-expressions, there are several possible derivation sequences using the call-by-value evaluation principle. We show two possible sequences and underline, at this, the inner redexes.

$foo\ \underline{(goo\ 1)}\ \underline{(hoo\ 2)}$
$\rightarrow\ foo\ \underline{(1+3)}\ \underline{(hoo\ 2)}$
$\rightarrow\ foo\ 4\ \underline{(hoo\ 2)}$
$\rightarrow\ foo\ 4\ (goo\ \underline{(goo\ 2)})$
$\rightarrow^2\ foo\ 4\ \underline{(goo\ 5)}$
$\rightarrow^2\ \underline{foo\ 4\ 8}$
$\rightarrow^2\ 12$

$foo\ \underline{(goo\ 1)}\ \underline{(hoo\ 2)}$
$\rightarrow\ foo\ \underline{(goo\ 1)}\ (goo\ \underline{(goo\ 2)})$
$\rightarrow^2\ foo\ \underline{(goo\ 1)}\ \underline{(goo\ 5)}$
$\rightarrow^2\ foo\ 4\ \underline{(goo\ 5)}$
$\rightarrow^2\ \underline{foo\ 4\ 8}$
$\rightarrow^2\ 12$

We translate the CCFL program and the function call into LMNtal (see Program 4.7.2) according to our scheme. At this, we partially evaluated *app_* atoms for better understanding the principle idea.

The destructuring of the function call *foo* (*goo* 1) (*hoo* 2) and of the functional expressions on the right-hand sides of the rules by introducing connecting links between the sub-expressions supports the control of the computation as we need it here: On the one hand, it is possible to express dependencies between sub-expressions in this way, on the other hand, the computations can be held apart from each other.

The idea is now to delay the outer calls by holding them apart from the rules until the computation of the inner redexes they depend on is finished. In LMNtal one can annotate a membrane template on the left-hand side of a rule with the so-called *stable flag*, such that it can match only with a cell containing no applicable rules. However, this stability property of cells refers only to rules within the cell. Thus, we need to hold the rules together with the inner redexes within one membrane to allow to check, whether they have been reduced completely. To enable on the other hand a concurrent computation of independent expressions, we must assign each atom a separate membrane (including a copy of the rules). Example 4.7.1 illustrates again, why this complex organisation is necessary.

Example 4.7.1 A first (but not satisfying) idea is, just to hold all inner redexes within one membrane together with the rule set. Outer calls are hold within a second membrane which "protects" them against rule application. They are lifted, in case that the computation of the inner nodes is finished. However, this does not allow a full concurrency of the evaluation of sub-expressions.

Consider e.g. Figure 4.3 which visualises an according computation of $hoo(2,Y)$, $goo(1,X)$, $foo(X,Y,Z)$. Membranes are represented as enclosing ellipses. All atoms are held within separate membranes for organisational reasons. Dependencies of atoms in the order of the evaluation are represented by arrows. In the initial state the atoms $hoo(2,Y)$ and $goo(1,X)$ are inner redexes to be reduced first. We mark these by a light-gray color. The atom $foo(X,Y,Z)$ represents an outer call to be delayed until the computation of its sub-expressions has been finished. Thus, we put it into a protecting membrane. The overall computation performs within an enclosing membrane which contains furthermore the LMNtal rules *@rules* generated from the CCFL program.

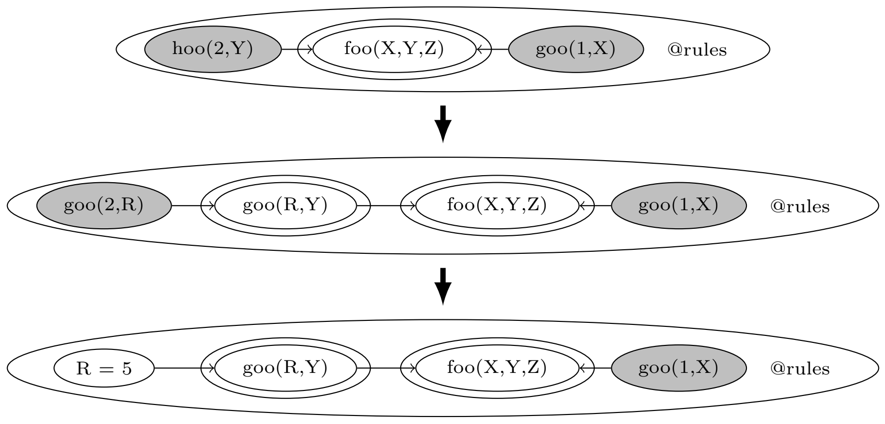


Figure 4.3: A computation sequence using an unsatisfactory strategy

Assume that in the first step the atom $hoo(2,Y)$ is chosen for rewriting (concurrently a reduction of $goo(1,X)$ would have been possible). This yields the second state, with the new inner redex $goo(2,R)$ and the new outer redex $goo(R,Y)$ which is delayed by an enclosing membrane. In the next step we chose $goo(2,R)$ for reduction which yields an irreducible atom $R = 5$.

Now we would like to allow a concurrent computation of the independent atoms $goo(1,X)$ and ($goo(R,Y)$, $R = 5$). However, since the stability property of cells only refers to the rules within a cell, the computation of ($goo(R,Y)$, $R = 5$) is delayed until the derivation of $goo(1,X)$ has been finished. Figure 4.4 shows a subsequent computation.

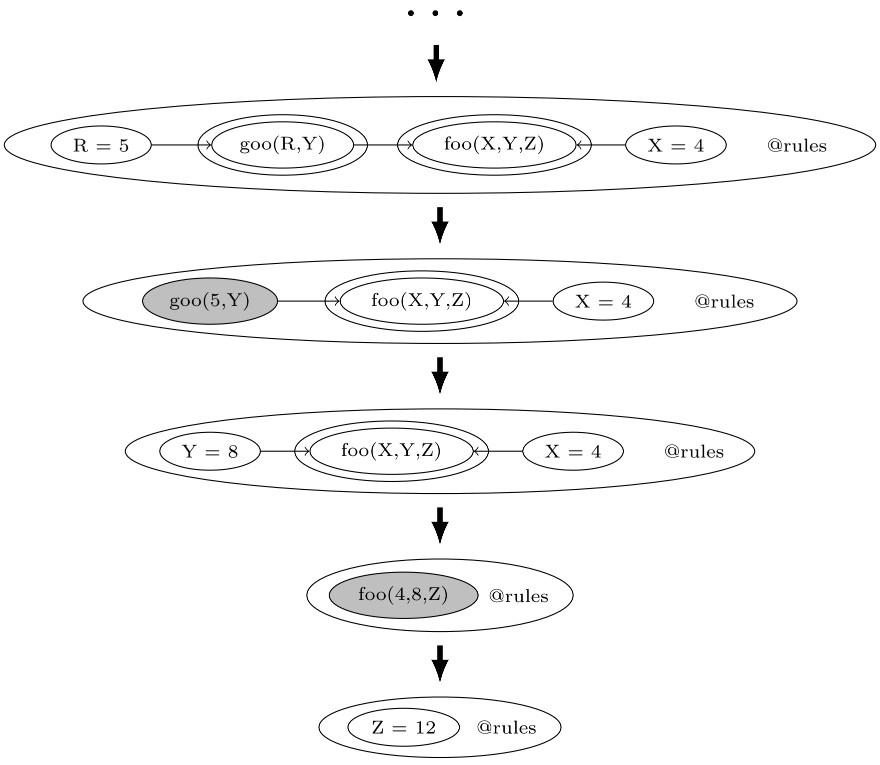


Figure 4.4: A computation sequence using an unsatisfactory strategy (continued)

To allow an actual concurrent computation of independent sub-expressions, we need to hold a copy of the rule set with each inner redex.

Figure 4.5 shows how this looks like for our example and the first computation steps. In the third sub-expression, now a concurrent computation of the independent atoms $goo(1,X)$ and $goo(5,Y)$ is enabled.

In Program 4.7.3 we display the LMNtal rules generated from Program 4.7.1 with membranes and further organisational elements for emulating a call-by-value strategy with concurrent evaluation of independent sub-expressions. The atoms $inner\_(i)$ and $outer\_(o)$ are used to distribute the rules into the membranes with inner redexes. The links between the membrane cells are used to explicitly express dependencies between sub-expressions.[4] The guards $ground(V)$ for every "argument" link is necessary, because LMNtal does not allow to "look into a membrane" for matching. This is possible only, when the arguments are LMNtal-*ground* atoms. This is ensured here, because all links are bound to values when a rule is applied because of the call-by-value redex selection. The atoms *preludeCBV.use* and *rules.use* load the prelude rule set and the generated rules from ***module***(*rules*) into the membranes of the initial call.

Program 4.7.4 shows the LMNtal program which controls our call-by-value strategy. The rules *cbv1* and *cbv2* organise the lifting of outer calls when the inner redexes they depend on have been completely reduced. The *unpack* rules are responsible for the distribution of

[4]This information is implicitly existent as well in the links of the atoms in many cases, but this does not apply in all cases; thus, we use extra links here.

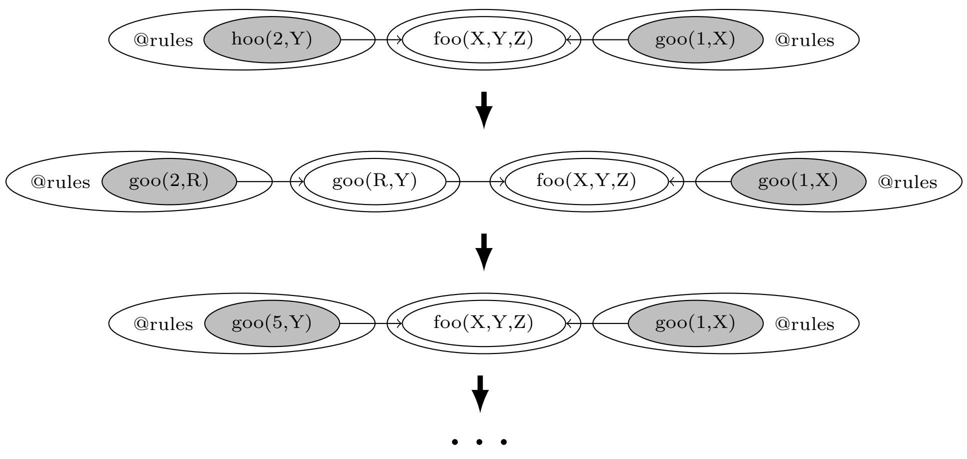


Figure 4.5: A call-by-value computation sequence

Program 4.7.3 An LMNtal program generated from the CCFL Program 4.7.1 for a call-by-value evaluation

```
{ module( rules ).

  { foo (A,B,Z), $p } :-
        ground(A), ground(B) |
        { Z = A + B, $p } .
  { goo (A,Z), $p } :-
        ground(A) |
        { Z = A + 3 , $p } .
  { hoo (A,Z), inLinks_(N), $p }  :-
        ground(A), ground(N) |
        { outer_(1), inner_(1),
          { goo(A,R), inLinks_(0), +L },
          { { goo(R,Z), inLinks_(N+1), -L, $p } }
        } .
}.

{{ hoo(2,Y), inLinks_(0), +L1 }, rules.use, preludeCBV.use},
{{ goo(1,X), inLinks_(0), +L2 }, rules.use, preludeCBV.use},
{{ foo(X,Y,Z), inLinks_(2), -L1, -L2 }}
```

the program rules into membranes of atoms resulting from the application of the rules of ***module***(*rules*).

Note: While the usage of membranes allows to implement evaluation strategies, it causes a considerable slow-down of the computation performance.

4.7.2 Call-by-name evaluation

A call-by-name evaluation strategy can be implemented in a similar way like the call-by-name reduction. However, there are three main differences to take into consideration.

Reordering of calls. A call-by-name evaluation strategy privileges the evaluation of outer redexes. Thus, we now need to hold inner calls within extra membranes to protect them against evaluation. Accordingly we receive a (nearly) inverse link structure for the destructured expressions in the rule bodies and the initial call.

It is no more necessary to explicitly compute with the number of expressions the evaluation of an expression depends on (as done for the call-by-value reduction using the $inLinks\_(N)$ atoms) because an expression can have one outer enclosing expression at most.

Unsuspendig inner redexes. As is well known, in certain cases it is necessary to evaluate an inner redex one or more steps to proceed with the evaluation of the enclosing outer expression, e.g. to chose between alternative cases or to compute an arithmetic expression. We implemented this by making available the program rules to the concerning sub-expression for exactly one computation step and protecting the newly received expressions afterwards again within membranes. This scheme already provides for the concurrent evaluation of sub-expressions, such that there is no need manage copies of the program rules here.

Figure 4.6 shows a visualisation of the computation of the LMNtal atoms $hoo(2, Y)$, $goo(1,X)$, $foo(X, Y,Z)$ wrt. Program 4.7.2 using the described mechanisms to emulate a call-by-name evaluation. One step in the figure represents one evaluation step and the lifting of a next inner redex to the evaluation level. The derivation shows that at every point only one function call can be evaluated, thus we do not need to copy the program rules. Moreover, one can see, that it is possible to alternate between the evaluation of independent sub-expressions to express concurrency, e.g. done here for the evaluation of $hoo(2, Y)$ and $goo(1,X)$.

Copying of structures. The main drawback in contrast to the call-by-value evaluation implementation is the necessity to copy graphs *including* membranes now: As it is well known, using a call-by-name strategy, function arguments are re-evaluated when they are used several times. This yields to the need to copy the concerning structures which may contain membranes in general.

LMNtal allows to copy structures without membranes simply (and as many times as needed) using the *ground* guard. To copy structures with membranes as it is necessary here, the language provides an API *nlmem.copy* which must be used as many times as copies are needed and which usage complicates and slows down again the performance of the program.

4.8 Implementation

The basic compilation scheme for functional CCFL programs with non-deterministic evaluation strategy is implemented in the file *Transform.hs*. Its main compilation function is *compileF* :: ***String*** −> ***IO*** ().

If the user has defined a function *main* with constant type to enclose the initial function call, then the compiler additionally generates an initial LMNtal atom *main* to run the

Program 4.7.4 An LMNtal program to emulate a call-by-value evaluation strategy

```
cbv1@@
{ @rules,
  {$procs1, +L, inLinks_(0)} }/,
{ {$procs2, -L, inLinks_(N)} } :-
 N =:= 1 |
   { @rules,
     {$procs1, $procs2, inLinks_(0)}
   }.

cbv2@@
{ @rules,
  {$procs1, +L, inLinks_(0)} }/ ,
{ {$procs2, -L, inLinks_(N)} } :-
 N > 1 |
{ { $procs2, inLinks_(N-1), $procs1} }.

unpack1@@
{ @rules,
  { outer_(N), $procs1,
    { { $procs2 } }
  }
} :- ground(N) |
{ @rules, { $procs1, outer_(N-1) } }, { { $procs2 } }.

unpack2@@
{ @rules,
  { outer_(0), inner_(N), {$proc1}, $procs1 }
} :- N > 1 |
{ @rules, { $proc1 } },
{ @rules, { outer_(0), inner_(N-1), $procs1 } }.

unpack3@@
{ @rules,
  { outer_(0), inner_(1), {$proc1} }
} :-
{ @rules, { $proc1 }}.
```

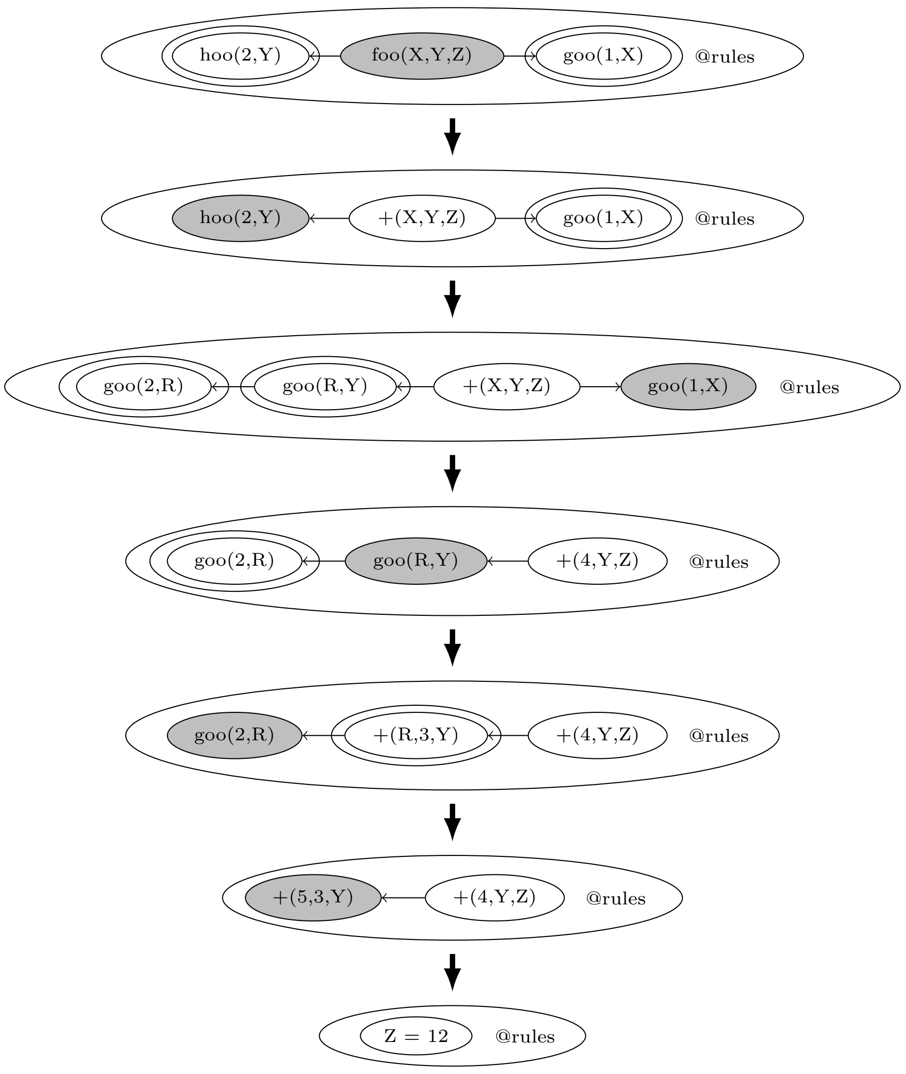


Figure 4.6: A call-by-name evaluation

program.

Functional CCFL programs for evaluation under a call-by-value strategy with concurrent computation of independent processes as described in Section 4.7.1 are translated into LMNtal code using the function *compileFcbv* :: ***String*** −> ***IO*** (). The compiler functionality is implemented in *TransformFpCbv.hs*.

As for the non-deterministic case, the compiler produces a main-call, if the user has defined a constant function *main* in the program. This call, however, is now enclosed by a membrane and organisational structures (for explanations cf. Program 4.7.3) to initiate a call-by-value evaluation:

$$\{\ rules.use,\ preludeCBV.use,\ \{main\ (Result),\ inLinks\_\ (0)\}\}$$

A prelude for functional programs to be evaluated non-deterministically resp. with a call-by-value strategy can be generated by the functions *prelude* :: ***IO*** () and *preludeCBV* :: ***IO*** () which are implemented in the file *CCFLPrelude.hs*.

The file *LMNtalPrettyPrint.hs* provides a pretty printer to generate LMNtal code from LMNtal ASTs.

The usage of membranes and the API *nlmem.copy* to implement evaluation strategies as described in this chapter cause a considerable slow-down of the computation performance of the programs with the current LMNtal version (see Section 7.2).

Chapter 5

A heap as CCFL runtime environment

In Chapter 4 we discussed a naive approach for the compilation of CCFL into LMNtal. The basic idea was to represent CCFL variables as LMNtal links which, however, yield strong restrictions for our language.

Links are one-to-one connections. Using links to represent CCFL variables we could transfer their bindings between atoms in rules. However, it was not possible to reuse bindings in different places. This was realised using LMNtal constructs for copying structures. However, there remained two main problems which could not be handled in this way: (1) *sharing of structures* and (2) *using unbound variables*. Restriction (1) excluded the realisation of a lazy evaluation strategy. Limitation (2) restricted us to functional programming and prevented the usage of constraints because these typically contain unbound variables. The representation of CCFL variables by LMNtal links, thus, constrained the compilation of CCFL to a functional sub-language with call-by-value or call-by-name evaluation strategies (Section 4.7).

To overcome these limitations we implemented a heap as CCFL runtime environment. Links, cells, and LMNtal atoms are used to represent heap structures and variables can be bound to them using links again.

This chapter is structured as follows: We consider the representation of structures in the heap in Section 5.1. Section 5.2 discusses new options and consequences from the introduction of a heap for functional CCFL. Using matching of heap structures now the compilation of CCFL guards becomes possible. We implemented a unification algorithm for heap structures in LMNtal. Using unification allows the realisation of CCFL *tell*-equality-constraints such that we finally reach the full power of CCFL (see Section 5.3).

5.1 The representation of the heap

We distinguish two kinds of data represented in the heap: variables and structures. For both we use an LMNtal atom with name *el* whose arguments are the name of the variable resp. the structure head symbol and links onto and from the element.[1]

5.1.1 CCFL variables

A *variable representation* consists of an atom $el(var(N),O,I)$ with a link N to the name(s) of the variable, a link O to a cell $\{+O,...\}$ which connects to the structure the variable is (possibly) bound to and a link I from a cell $\{on(I),...\}$ which manages links onto the variable.

[1]For the heap representation see also Section 7.2.

Links onto some element are marked by the atom *on* which is 0-ary in the case of an unbound variable. A variable atom may stand for a number of variables, in case they have been unified. Thus, we manage the multiple names again by a common enclosing membrane.

Figure 5.1(a) shows the representation of the free CCFL variable *a*. The two links $+A1$ and $+A2$ are links from some heap structures onto it, i.e. the variable *a* is shared between two other structures here. Figure 5.1(b) represents a variable *b* which has been unified with a variable *c*. It links by L onto some other structure.

$$\begin{tikzcd}
name(a,NameL) \arrow[d] & & {} \arrow[d, shift left] \arrow[d, shift right] \\
\{+NameL,+N\} & & \{on(I),+A1,+A2\} \arrow[dl] \\
& el(var(N),O,I) \arrow[ul] \arrow[d] & \\
& \{+O,on\} &
\end{tikzcd}$$

(a) A free variable *a*

$$\begin{tikzcd}
name(b,NameL1) \arrow[dr] & & name(c,NameL2) \arrow[dl] & \\
& \{+NameL1,+NameL2,+N\} & & \{on(I),...\} \arrow[dl] \\
& & el(var(N),O,I) \arrow[ul] \arrow[d] & \\
& & \{+O,on(L)\} \arrow[d] & \\
& & {} &
\end{tikzcd}$$

(b) A bound variable *b*

Figure 5.1: The representation of variables in the CCFL heap

5.1.2 Non-variable terms

Non-variable CCFL terms or structures are represented by an atom $el(cons(F),OL,I)$, where F is the root symbol of the term, OL is a list of links to cells connecting to the structures of the arguments and I is a link from a cell $\{on(I),...\}$ which manages links onto the term.

As an example consider the functional expression *mult x x* and assume that *x* has been bound before to the term *succ* 2. That is, we consider the term *mult* (*succ* 2) (*succ* 2), where (*succ* 2) is a shared element. Figure 5.2(a) shows an according graph structure as illustration. Figure 5.2(b) shows the according heap structure. The links $L1$ and $L2$ connect into the same cell and realise sharing in this way.

5.2 Functional CCFL with heap

In this section we discuss the effects for functional CCFL using a heap for the representation of data.

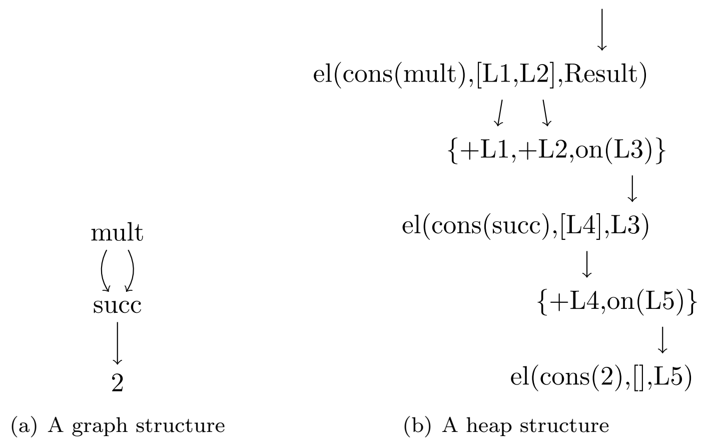


Figure 5.2: The representation of a non-variable term in the CCFL heap

5.2.1 Adaptions to the compilation scheme

The integration of heap structures into the CCFL compiler is realised by a transformation (*heap transformation* in the following) of the LMNtal AST generated by the scheme of Section 4.1. The main idea is to translate the atoms of the rule heads and rule bodies into heap structures and to transform the guards into matching expressions on heap elements resp. again into LMNtal guards.

The compilation of functions

Starting from an AST representing an LMNtal program generated using the scheme of Section 4.1, we transform each rule. For simplification, we mainly suspend the handling of multiple link occurrences and discuss this in detail on page 64.

The head of a rule generated from a CCFL function by our general compilation scheme is a process template consisting of one LMNtal atom resp. two LMNtal atoms resulting from case-expressions. Rule heads are used to chose a rewrite rule by matching against an atom. Thus, we just need to translate a rule head into atoms and cells representing the according heap structures.

It is important to copy data representations from the head into the body, because we need to maintain the heap for further computations.

Example 5.2.1 Consider the generated LMNtal rules from the CCFL function *flist* from Example 4.1.5.

```
/* result of the general compilation scheme: */

flist (A, V_0)  :- ...

case__flist (V_1,A, V_0),
nil (V_1)  :- ...

case__flist (V_1,A, V_0),
cons (B,D, V_1)  :- ...
```

The heap transformation yields the following rule heads for the first three rules.

```
/* result of the heap transformation */
```

```
el (cons (flist),[A], V_0) :- ...

el (cons (case__flist),[V_1,A], V_0),
el (cons (nil),[], V_1_),
{on (V_1_),$pV_1,+V_1} :-
  // maintain the heap data
  el (cons (nil),[], V_1_),
  {on (V_1_),$pV_1},
  // the rule body transformation follows
  ...

el (cons (case__flist),[V_1,A], V_0),
el (cons (cons),[B,D], V_1_),
{on (V_1_),$pV_1,+V_1} :-
  // maintain the heap data
  el (cons (cons),[B,D], V_1_),
  {on (V_1_),$pV_1},
  // the rule body transformation follows
  ...
```

The atoms are translated according to the representation discussed in Section 5.1. The link *V_0* of each rules connects to the result of the rewrite process and remains, thus, unchanged. The arguments *A*, *V_1*, *B*, and *D* are links to some heap structures and remain unchanged. However, *V_1* links onto a heap element which is used for matching in the second and third rule. This part of the heap structure must be rebuild in the rule body, while the connecting link *V_1* disappears. The process context *$pV_1* stands for further potentially existing links to *el* (*cons* (*nil*),[], *V_1_*) resp. *el* (*cons* (*cons*),[*B*,*D*], *V_1_*) in the heap.

The body of a rule generated from a CCFL function by our scheme is a process template consisting of LMNtal atoms and connector atoms. The heap transformation just translates them into atoms, cells, and links building valid heap structures.

Example 5.2.2 Consider the third LMNtal rule generated from the CCFL function *flist* of Example 4.1.5 again.

```
/* result of the general compilation scheme: */

case__flist (V_1,A, V_0),
cons (B,D, V_1)  :-
  case__case__flist (V_2,A,B,D, V_0),
  V_2 = D .

/* result of the heap transformation */

el (cons (case__flist),[V_1,A], V_0),
el (cons (cons),[B,D], V_1_),
{on (V_1_),$pV_1,+V_1},
{$pB,+B},
{$pD,+D}  :-
  // maintain the heap data
  el (cons (cons),[B,D], V_1_),
  {on (V_1_),$pV_1},
  // rule body transformation
  el (cons (case__case__flist),[V_2,A,B_1,D_1], V_0),
  V_2 = D_2,
  {$pB,+B,+B_1},
  {$pD,+D,+D_1,+D_2}  .
```

Again links connecting to rewrite results, like *V_0* and *V_2* remain unchanged. One point which is considered in more detail later are multiple occurrences of links, like *D* and *B*. This is handled now by multiple links into cells containing also links to the according heap elements. For example, the links +*B* and +*B_1* originate both from *B* and are, thus, hold in the cell {*$pB*,+*B*,+*B_1*} which connects to the same heap element.

The guard of an LMNtal rule generated from a functional CCFL program can only originate from case expressions on natural numbers, like in Example 4.6.3. The heap transformation translates equality and disequality guards on natural numbers into guards for the according heap structures. This is again best illustrated by means of an example.

Example 5.2.3 Consider the heads and guards of the third and fourth LMNtal rules generated from the *fibonacci* function in Example 4.6.3. The heap transformation generates heap representations and respective guards as follows.

```
/* result of the general compilation scheme: */

case__fibonacci (V_1,N, V_0)  :-
V_1 =:= 1 |
   ...

case__fibonacci (M,N, V_0)  :-
M =\= 1,
M =\= 0 |
   ...

/* result of the heap transformation */

el (cons (case__fibonacci),[V_1,N], V_0),
el (cons (V_1_V_),[], V_1_),
{on (V_1_),$pV_1,+V_1} :-
V_1_V_ =:= 1  |
   // maintain the heap data
   el (cons (V_1_V_),[], V_1_),
   {on (V_1_),$pV_1},
   // the rule body transformation follows
   ...

el (cons (case__fibonacci),[M,N], V_0),
el (cons (M_V_),[],M_),
{on (M_),$pM,+M}  :-
M_V_ =\= 1,
M_V_ =\= 0  |
   // maintain the heap data
   el (cons (M_V_),[],M_),
   {on (M_),$pM},
   // the rule body transformation follows
   ...
```

Multiple link occurrences again

One reason for the introduction of the heap was the need to accordingly handle multiple occurrences of variables in CCFL programs resp. of links in generated LMNtal rules. Using the heap as CCFL runtime environment now allows the sharing of structures such that the need to copy data vanishes.

Links representing multiple occurrences of variables must be renamed and connected to a common cell which contains one link onto the common structure. Examples for this have already been given in Example 5.2.2 and in Figure 5.1.

Example 5.2.4 Consider the simple CCFL program building a list of three elements and its (simplified)[2] transformation.

```
/* the CCFL program */
fun p :: a -> a -> List a
def p a b =
    Cons a (Cons a (Cons b Nil))

/* result of the heap transformation: */
el (cons (p),[A,B], V_0),
{$pA,+A}  :-
   el (cons (nil),[], V_8_),
   V_7 = B,
   el (cons (cons),[V_7, V_8], V_5_),
   V_4 = A_L_1,
   el (cons (cons),[V_4, V_5], V_2_),
   V_1 = A_L_2,
   el (cons (cons),[V_1, V_2], V_0),
   {on (V_8_),+V_8},
   {on (V_5_),+V_5},
   {on (V_2_),+V_2},
   {$pA,+A_L_1,+A_L_2}  .
```

The three occurrences of the variable *a* in the CCFL function are represented by the links *A* in the rule head and *A_L_1* and *A_L_2* in the rule body, both linking into a common cell {*$pA*,+*A_L_1*,+*A_L_2*}. This cell contains furthermore the process context *$pA* which contains a link to the structure the variable *a* is bound to.

Higher-order functions and partial applications

Since the heap elements for non-variable terms are represented in a uniform way by atoms *el*(*cons*(*F*),*OL*,*I*), we can abstract the *app_* rules from Program 4.4.1 for higher-order functions and partial applications of functions within one LMNtal rule as given in Program 5.2.1.

A rewrite rule for *duplicatePointer_* duplicates a list of links on argument structures twice; *conc_* is used to concatenate two lists of links into one.

The rule of Program 5.2.1 replaces a heap structure representing the call

*app_*(*f*(*B1* ,..., *Bm*),*A1*,...,*An*,*Z*)

by the two new structures

f(*B11* ,..., *Bm1*,*A1*,...,*An*,*Z*) and *f*(*B12* ,..., *Bm2*,*LF*).

[2]We left out the generation of *app_* nodes.

Program 5.2.1 A common rule for handling partial application and higher-order functions in CCFL programs with heap management

```
el(cons(app_),[FN| ArgList],Z), {on(Z),$lz},
{+FN, on(LF),$lf}, el(cons(F),FArgList,LF) :-
    ground(F) |
    {on(Z),$lz}, {on(LF),$lf},
    el(cons(F),conc_(FArgList1,ArgList),Z),
    el(cons(F),FArgList2,LF),
    duplicatePointer_(FArgList,[FArgList1,FArgList2]).
```

The first structure is the result of the application of the partial call *f*(*B1* ,..., *Bm*) on the further arguments *A1*, . . ., *An*. The second structure must be maintained for the case that a partial application is reused itself.

Figure 5.3 visualises the rule of Program 5.2.1 including the copying of links to the arguments. At this, the link connections from *A1* ,..., *An* and *F* to other heap elements were left out to improve readability.

$$\begin{tikzcd}
 & \{on(Z),\$lz\} \arrow[d] & \\
 & el(cons(app\_),[FN \mid [A1,...,An]],Z) \arrow[d] & \\
 & \{+FN,on(LF),\$lf\} \arrow[d] & \\
 & el(cons(F),[B1,...,Bm],LF) \arrow[dl] \arrow[dr] & \\
\{+B1,\$pB1\} & \cdots & \{+Bm,\$pBm\}
\end{tikzcd}$$

rewrites into

$$\begin{tikzcd}
\{on(Z),\$lz\} \arrow[d] & & \{on(LF),\$lf\} \arrow[d] \\
el(cons(F),[B11,...,Bm1,A1,...,An],Z) \arrow[d] \arrow[drr] & & el(cons(F),[B12,...,Bm2],LF) \arrow[dll] \arrow[d] \\
\{+B11,+B12,\$pB1\} & \cdots & \{+Bm1,+Bm2,\$pBm\}
\end{tikzcd}$$

Figure 5.3: Visualisation of the generalised *app_* rule

5.2.2 Lazy evaluation

The newly introduced heap as CCFL runtime environment realises the sharing of common data by different expressions. Combining the heap transformation and the translation scheme for a call-by-name evaluation as sketched in Section 4.7.2 implements *lazy evaluation* for CCFL.

Program 5.3.1 The introduction of a new "nameless" variable

```
el (cons (produce),...) :-
   el (var (NoNameBuf1),OnBuf1,InBuf1),
   {on,+OnBuf1}, {+NoNameBuf1}, {on (InBuf1),...},
   ...
```

5.3 Full CCFL with heap

The representation of data in a heap, in particular the sharing of common data by different structures, the storage of possibly unbound variables, and a unification mechanism for heap data allows the compilation of CCFL guards and the realisation of *tell*-constraints such that we finally reach the full power of CCFL.

5.3.1 The compilation of constraint and guarded expressions

Considering the CCFL syntax there are four concepts which extend functional CCFL to full CCFL: the computation with *free variables*, *guards* for a non-deterministic, but conditional, rule choice, *equality-constraints* on functional expressions, and *user-defined constraints*. We consider these items in the following.

Free variables

A characteristic property of constraints is the presence of free variables in expressions. Valid bindings resp. sets or intervals of these for free variables are computed at runtime and – in case of CCFL – by possibly concurrent communicating processes.

As an example consider Program 1.1.3, where the two processes (*dice x1*) and (*dice y1*) throw a dice and communicate the result value via free variables to processes summing these up. Another example is Program 1.1.4, where free variables are used to represent a buffer which is filled by a producer and read by a consumer.

In CCFL free variables are introduced by the ***with*** construct. Their representation in the heap has been described in Section 5.1.1. Free variables in rules (except for the main rule as initial call), however, are represented in the heap without a name. Since rules may be applied several times, we would get variables with different meaning but the same name otherwise. Anyway, this in no problem, because variables introduced by rules are local such that their names are never needed.

Consider Program 5.3.1 which results from the translation of the *produce* function resp. user-defined constraint of Program 1.1.4. It introduces the free variable *buf1*. The transformation yields a "nameless" variable representation in the heap. By links into the cell {*on* (*InBuf1*),...} one can refer to this variable.

Guards

Guards are used for a conditional choice of alternatives within the definition of a user-defined constraint. For different matching guards the choice of the alternative to be applied is non-deterministic. If none of the guards of a rule is satisfied, the computation suspends, either infinitely or until another concurrent process provides a satisfying binding.

While alternatives of guarded expressions are simply translated each into one according LMNtal rule, the translation of the guard atoms resp. *ask*-constraints is a more interesting point. Guard atoms in CCFL are a bound-constraints and match-constraints.

Bound-constraints are used to check whether a variable is bound to a non-variable constructor term or value.

A guard ***bound*** *x*, where *x* is of user-defined type or of type ***Bool***, is realised by an additional LMNtal atom in the head of the generated rule to implement an according matching of heap structures. For variables *x* of type ***Int*** or ***Float*** we directly use the respective LMNtal guards.

A match-constraint *x =:= expr* checks on the binding of a variable *x* to a concrete constructor or value. Match-constraints are implemented by additional LMNtal atoms in the rule heads matching for the according heap structures.

Example 5.3.1 The user-defined constraint *isIn* of Example 1.1.3 non-deterministically chooses values from a given list. We either take the first element as result value or initiate a further computation on the rest of the list.

```
fun isIn :: List a -> a -> C
def isIn l x =
    l =:= Cons y ys -> x =:= y |
    l =:= Cons y ys -> case ys of Nil -> x =:= y ;
                                  Cons z zs -> isIn ys x
```

The compilation yields identical rule heads as given below for both alternatives, such that the rule choice is finally non-deterministically decided at runtime by the LMNtal system.

```
el (cons (isIn),[L,X], VC_0),
el (cons (cons),[Y, Ys], L_),
{on (L_), $pL,+L} :-
    ...
```

Equality-constraints on functional expressions

Tell-constraints *x =:= fexpr* introduce an equality between the expression bound to the variable *x* and the functional expression *fexpr*. Since both are functional expressions (of base type) their evaluation is deterministic.

The constraint *x =:= fexpr* is satisfied, if both expressions can be reduced to the same ground data term. This interpretation of equality is called *strict equality* [HAB^{+}06]. Thus, the implementation must perform an evaluation of both expressions to data terms and then perform a unification of the resulting heap expressions. The call-by-value strategy conforms nicely to strict equality. For the lazy strategy the evaluation to constructor terms is interleaved with unification.

The transformation of a CCFL equality-constraint into concerning LMNtal code is simple. We just produce a *unify* atom on both heap structures which initiates a unification procedure for heap structures of data terms implemented in LMNtal.

Example 5.3.2 Consider again the producer of the producer-consumer example in Program 1.1.4. The constraint *buf =:= Cons item buf1* just generates a concerning *unify* atom.

```
el (cons (produce),[Buf], VC_0) :-
  // generate fresh variable buf1
  el (var (NoNameBuf1), OnBuf1, InBuf1),
  {on,+OnBuf1}, {+NoNameBuf1}, {on (InBuf1),+Buf1},
  // generate structure for item
  ...
  // generate expression (Cons item buf1)
  el (cons (cons),[Item, Buf1], VC_2_),
  {on (VC_2_),+VC_2},
  // unify call: buf =:= Cons item buf1
```

```
unify (Buf , VC_2 , VC_0),
...
```

User-defined constraints

User-defined constraints correspond to functions but are of type ***C*** and may contain guards, free variables and *tell*-constraints.

The compilation of a user-defined constraint is composed of the compilation of its subelements. At this, it is important to note, that a user-defined constraint also needs a "result" link on its heap representation, even if constraints are only checked for their satisfiability (in contrast to a function which actually computes a result). The reason for this is, that CCFL is intended to support the partial application of user-defined constraints and their usage as arguments of higher-order functions.[3]

Example 5.3.3 The link *VC_0* in the head and body of the program rule of Example 5.3.2 connects to the heap structure of a user-defined constraint resp. of the result of an equality-constraint.

5.4 Implementation

The compilation of CCFL constructs according to the general transformation scheme is implemented in the file *Transform.hs.* In a second step, the thereby generated LMNtal AST is transformed into a new LMNtal AST taking the heap representations into consideration; this is implemented in the file *TransformCp.hs.* The compilation function is *compileCCFL* :: ***String*** −> ***IO*** ().

The compiler generates a main-call, if the user has defined a constant function *main* in the program. This call is again enriched by organisational structures:

$$\{\ rules.use,\ preludeHeap.use,\ app.use,\ unify.use,\ structure\ (main, [], Result)\}$$

The atom *rules.use* loads the rules generated from the CCFL user program into the membrane. The atoms *preludeHeap.use*, *app.use*, and *unify.use* supply a prelude with predefined functions and constraints, the handling of partial applications and higher-order functions, and the unification of heap structures, resp. The main call *structure* (*main*, [], *Result*) generates an according initial heap representation.

The prelude mentioned above for built-in functions realising the residuation strategy for arithmetic functions is provided in the file *preludeHeap.lmn.* The unification of heap data has been implemented in the file *unify.lmn.* Furthermore, there is a module *show.lmn* which allows to gain a user-friendly representation of the heap structure of the final result of a computation.

Note, that the representation of the heap using membranes as discussed in this chapter causes a slow-down in the performance which is typical for LMNtal programs with membranes for the current LMNtal version (see Section 7.2).

[3]Currently, we use these links only for the reduction of *app_* nodes resulting from the application of user-defined constraints (generated as for function applications, see Section 4.1.2).

Chapter 6

Extending CCFL by new constraint domains

CCFL provides *ask*-constraints in the guard of rules and *tell*-constraints in the rule bodies. While all *ask*-constraints check on the binding of variables to constructor terms, *tell*-constraints are either itself applications of user-defined constraints or are equality constraints on functional expressions. An extension of CCFL by new constraint domains (and solvers) as it is implemented in many declarative languages [FHGSP03, CHS+03, Lux01, Moz06] is desirable.

In this chapter, we sketch on ideas for the introduction of new constraint domains into CCFL (and LMNtal).

6.1 Two ways of constraint integration

The extension of CCFL by new constraint domains and solvers can be realised in two ways.

On the one hand one may integrate constraints whose solving mechanisms can be implemented directly using the target language LMNtal of the CCFL compiler. The approach fits well for domains which can be realised in a natural way in LMNtal itself, for example set-constraints [Aik94, PP97]. We also followed this approach when providing *tell*-equality-constraints for CCFL. We implemented a unification algorithm for terms in LMNtal as runtime environment for CCFL, where *unify*-calls are incorporated into the compilation result. Even the arithmetic built-in functions can be considered as representatives of this first approach in its simplest case, because they as well base on their equivalents in LMNtal.

The second approach is to integrate external constraints and solvers into both languages LMNtal and CCFL. However, this demands an extension of syntax and semantics of *both* languages. We sketch on this approach with an arithmetic domain as example in the following sections.

6.2 External *tell*-constraints

The first step is to allow external constraints in the body of user-defined constraints in CCFL programs.

We consider an example inspired by [Lux01]. Program 6.2.1 combines external arithmetic constraints and user-defined constraints in CCFL. The constraint symbols of the external domain are annotated with "#". For better understanding we use the list syntax from HASKELL (currently not supported by CCFL) and simplify the declaration of logical variables in function *main*.

Program 6.2.1 Computation of a schedule

```
fun main :: C
def main =
  with a, b, c, d, e, f, end :: Int
  in schedule a b c d e f end

fun schedule :: Int -> Int -> Int -> Int -> Int -> Int -> Int -> C
def schedule a b c d e f end =
  with maxEnd :: Int
  in
    a+3 #<= b & a+3 #<=c & a+3 #<= d & b+1 #<=e &
    c+3 #<=e  & c+3 #<=f & d+2 #<= f & e+1 #<=end &
    f+3 #<=end  & a #=0 & end #<= maxEnd &
    maxEnd =:= sum ([3,1,3,2,1,3]) &
    farm (#elem [0,...,maxEnd]) ([a,b,c,d,e,f,end])

fun farm :: (a -> C) -> List a -> C
def farm c (head : rest) =
  case rest of []      -> c head ;
               (rh:rr) -> c head & farm c rest
```

The user-defined constraint *schedule* takes the variables a, b, c, d, e, f, and *end* which represent the starting times of according tasks to be scheduled. The tasks must be performed in the left-to-right order given in Figure 6.1, each task taking a certain time as given. Task a starts at time point 0.

The user-defined constraint *farm* takes a constraint, here the partial application (*#elem* [0,..., *maxEnd*]) :: *a* −> ***C*** of an external constraint and applies it to a list of variables. The equality constraint *maxEnd* =:= *sum* ([3,1,3,2,1,3]) determines an upper bound for the final time of the schedule by summing up the task durations.

$$\begin{tikzcd}
& \text{b/1} \arrow[r] & \text{e/1} \arrow[dr] & \\
\text{a/3} \arrow[ur] \arrow[r] \arrow[dr] & \text{c/3} \arrow[ur] \arrow[dr] & & \text{end} \\
& \text{d/2} \arrow[r] & \text{f/3} \arrow[ur] &
\end{tikzcd}$$

Figure 6.1: Tasks and durations, ordered from left to right

We show a derivation for the goal *main*, where an additional *constraint store C* holds the information of the external solver.

main

$\rightsquigarrow$ *schedule a b c d e f end*
with store $C = \{\}$

$\rightsquigarrow$ *a*+3 #<= *b* & *a*+3 #<=*c* & *a*+3 #<= *d* & *b*+1 #<=*e* & *c*+3 #<=*e* & *c*+3 #<=*f* & *d*+2 #<= *f* & *e*+1 #<=*end* & *f*+3 #<=*end* & *a* #=0 & *end* #<= *maxEnd* & *maxEnd* =:= *sum* ([3,1,3,2,1,3]) & *farm* (#*elem* [0,...,*maxEnd*]) ([*a*,*b*,*c*,*d*,*e*,*f*,*end*])
with store $C = \{\}$

$\rightsquigarrow^\star$ *maxEnd* =:= *sum* ([3,1,3,2,1,3]) & *farm* (#*elem* [0,...,*maxEnd*]) ([*a*,*b*,*c*,*d*,*e*,*f*,*end*])

with store $C = \{\ 3 <= b,\ 3 <= c,\ 3 <= d,\ b+1 <= e,\ c+3 <= e,\ c+3 <= f,$
$d+2 <= f,\ e+1 <= end,\ f+3 <= end,\ a = 0,\ end <= maxEnd\}$

$\rightsquigarrow^\star$ *farm* (#*elem* [0,...,13]) ([a,b,c,d,e,f,end])
with store $C = \{\ 3 <= b,\ 3 <= c,\ 3 <= d,\ b+1 <= e,\ c+3 <= e,\ c+3 <= f,$
$d+2 <= f,\ e+1 <= end,\ f+3 <= end,\ a = 0,\ end <= 13,\ ...\}$

$\rightsquigarrow^\star$ *a* #*elem* [0,...,13] & ... & *end* #*elem* [0,...,13]
with store $C = \{\ 3 <= b,\ 3 <= c,\ 3 <= d,\ b+1 <= e,\ c+3 <= e,\ c+3 <= f,$
$d+2 <= f,\ e+1 <= end,\ f+3 <= end,\ a = 0,\ end <= 13,\ ...\}$

$\rightsquigarrow^\star$ *Success*
with store $C = \{\ 3 <= b,\ 3 <= c,\ 3 <= d,\ b+1 <= e,\ c+3 <= e,\ c+3 <= f,$
$d+2 <= f,\ e+1 <= end,\ f+3 <= end,\ a = 0,\ b \in \{3,...,11\},$
$c \in \{3,...,7\},\ d \in \{3,...,8\},\ e \in \{6,...,12\},\ f \in \{6,...,10\},\ end \in$
$\{9,...,13\},\ ...\}$

The derivation mainly underlines two things: First, the external constraint store C must exchange information with the CCFL program about restrictions on the bindings of the variables of common sorts. And second, currently, we just check the satisfiability of the constraints and narrow the solution sets of the variables. However, we do not provide actual solutions (as far as there is not exactly one unique solution).

Problems with similar structure are, e.g. the well-known send-more-money problem or the 8-queens problem.

6.3 External *ask*-constraints

The second extension by external constraints concerns the CCFL guards. Currently, match- and bound-constraints allow to perform an entailment test of constraints on the structure of (potentially incompletely bound) terms. An extension of the guards by arithmetic constraints realises similar tests on numbers which can as well be used for the coordination of processes.

Program 6.3.1 allows the sorting of lists with potentially unbound variables of type ***Int***. For these, however, there must be sufficient information to decide about their position within the resulting ordered list. The constraint abstraction *filter* is defined by four alternatives with exclusive guards. Again, we mark external constraint symbols by "#". The function *concat* concatenates a list of lists into one list. Since the new arithmetic *ask*-constraints are able to deal with unbound variables (in contrast to the similarly looking guard atoms in LMNtal), *filter* is able to deal with unbound variables.

We consider two traces of representative derivations for illustration:

quicksort [7, *a*,2] *r*
$\rightsquigarrow$ *filter* 7 [7, *a*,2] (*Triple* [] [] []) (*Triple* *l* *e* *g*) &
quicksort *l* *resl* & *quicksort* *g* *resg* & *r* =:= *concat* [*resl*, *e*, *resg*]
$\rightsquigarrow$ *filter* 7 [*a*,2] (*Triple* [] [7] []) (*Triple* *l* *e* *g*) &
quicksort *l* *resl* & *quicksort* *g* *resg* & *r* =:= *concat* [*resl*, *e*, *resg*]
$\rightsquigarrow$ (the computation suspends)

quicksort [7, *a*,2] *r* & *a* #> 8
$\rightsquigarrow$ *filter* 7 [7, *a*,2] (*Triple* [] [] []) (*Triple* *l* *e* *g*) &
quicksort *l* *resl* & *quicksort* *g* *resg* & *r* =:= *concat* [*resl*, *e*, *resg*] & *a* #> 8
$\rightsquigarrow$ *filter* 7 [*a*,2] (*Triple* [] [7] []) (*Triple* *l* *e* *g*) &
quicksort *l* *resl* & *quicksort* *g* *resg* & *r* =:= *concat* [*resl*, *e*, *resg*] & *a* #> 8
$\rightsquigarrow$ *filter* 7 [*a*,2] (*Triple* [] [7] []) (*Triple* *l* *e* *g*) &
quicksort *l* *resl* & *quicksort* *g* *resg* & *r* =:= *concat* [*resl*, *e*, *resg*]

Program 6.3.1 Quicksort with arithmetic *ask*-constraints

```
data Triple a = Triple (List a) (List a) (List a)

fun quicksort :: List Int -> List Int -> C
def quicksort i o =
  with g, e, l, resl, resg
  in case i of
       Nil -> o =:= Nil;
       x : xs ->
         filter x i (Triple [] [] []) (Triple l e g) &
         quicksort l resl &
         quicksort g resg &
         o =:= concat resl : (e : (resg : []))

fun filter :: Int -> List Int -> Triple Int -> Triple Int -> C
def filter x list (Triple a b c) result =
     list =:= [] ->
       result =:= Triple a b c |
     list =:= y : ys & x #> y ->
       filter x ys (Triple (y : a) b c) result |
     list =:= y : ys & x #= y ->
       filter x ys (Triple a (y : b) c) result |
     list =:= y : ys & x #< y ->
       filter x ys (Triple a b (y : c)) result
```

with store $C = \{a > 8\}$

$\rightsquigarrow$ *filter* 7 [2] (*Triple* [] [7] [*a*]) (*Triple* *l e g*) &
quicksort *l resl* & *quicksort* *g resg* & *o* =:= *concat* [*resl*, *e*, *resg*]
with store $C = \{a > 8\}$

$\rightsquigarrow^\star$ *filter* *a* [*a*] (*Triple* [] [] []) (*Triple* *l' e' g'*) & ...
with store $C = \{a > 8, ...\}$

$\rightsquigarrow^\star$ *Success*
with store $C = \{a > 8, r = [2, 7, a], ...\}$

6.4 Finding concrete solutions

The integration of external constraints as sketched above, allows to use constraints for the process coordination and to check the satisfiability of sets of constraint. At this, free variables are assigned sets of values representing possible solutions.

However, as already discussed in the scheduling example in Section 6.2, the programmer is often not only interested in the general satisfiability of a problem expressed by constraints but also in finding a concrete solution or in labeling all solutions. Depending on the constraint domain, often a computation of a minimum or a maximum wrt. a certain goal function is desired.

There are certain approaches to be considered for this purpose including *encapsulated search* [HAB$^+$06], *monads* [Lux01], or the introduction of particular higher-order constraints.

6.5 Handling external constraints in LMNtal

The integration of external constraints into CCFL demands an extension of the compilation scheme and of the target language LMNtal. LMNtal allows to use processes in rule bodies which build on inline code [LMN07]. Thus an implementation of external CCFL *tell*-constraints based on this option seems possible. However, *ask*-constraints may require an actual extension of the LMNtal compiler.

An external solver to be integrated needs a constraint propagation function for the *tell*-constraints, (preferably) an entailment test (otherwise implemented on top of the propagation function) for the *ask*-constraints and functions to handle the constraint store. Appropriate solvers are e.g. the CHOCO constraint solver [Cho08] on finite-domain constraints written in JAVA or GECODE [GeC07] for finite-domain constraints and finite-set constraints implemented in C++ (with bindings to several languages).

Chapter 7

Conclusion

This report discusses the design and implementation of the Concurrent Constraint Functional Language CCFL which combines concepts from the functional and the constraint paradigms and allows for concurrent computation of processes.

We presented a compiler for CCFL programs written in HASKELL which generates LMNtal code. We implemented three schemes for code generation caused by different approaches for the presentation of CCFL variables by LMNtal links. The first and second versions persist in the viewpoint that links are just variables with certain restrictions. However, this constrains the compilation of CCFL to a functional sub-language and disallows lazy evaluation. To overcome these limitations we implemented a heap in LMNtal as CCFL runtime environment such that we finally reached the full power of CCFL. The current version of our CCFL compiler consists of about 10000 lines of Haskell code and 1500 lines of LMNtal code realising the runtime environment.

Our implementation based on LMNtal demonstrates the embedding of the concurrent functional paradigm into the LMNtal language model which allowed to equip CCFL with advanced language features, like constraints for communication and synchronisation of process systems, and to express non-deterministic computation.

7.1 Related work

The language design of CCFL lends from a diploma thesis [Lor06] supervised by the author in the run-up and as preparatory work for the development of CCFL. CCFL is an extension of the therein proposed language FATOM which compiled into code for an abstract machine ATAF. The syntax of CCFL borrows from the (sequential) functional languages HASKELL and OPAL (see e.g. [Bir98, Hud00, PH06]).

Functional languages allowing for concurrent computation of processes are e.g. EDEN [LOMP05] and ERLANG [AVWW07], both using explicit notions for the generation of processes and their communication or CONCURRENT HASKELL [PGF96] which supports threads via the IO monad. The constraint functional language GOFFIN [CGKL98] strongly separates the functional sub-language from the constraint part which is used as coordination language.

CURRY [HAB$^+$06] is a functional-logic language combining the functional and the logic paradigms and builds on the evaluation principle narrowing in contrast to residuation and non-deterministic choice in CCFL. A language similar to CURRY is TOY [FHGSP03] for which the integration of different arithmetic constraint domains has been examined over the recent years.

The CCFL coordination of concurrent processes is based on the concurrent constraint programming (CCP) model [SR90], where we also lend the notions of *ask*- and *tell*-constraints from.

7.2 LMNtal as target language of the CCFL compiler

The language LMNtal developed by Kazunori Ueda and his group at Waseda University in Tokyo is the target language for the CCFL compiler. The CCFL compiler builds on the LMNtal version 1.00.20071019.

The ability of LMNtal to unify and to model computation paradigms proved to be very useful for our compiler implementation. CCFL allows to combine functional and constraint programming and to express concurrent and non-deterministic computations. Our implementation shows, that modelling combined language paradigms in LMNtal is possible in a convenient way and by means of clear transformations.

There have been some drawbacks in LMNtal wrt. our implementation which, however, are mainly under consideration for the current LMNtal implementation and conceptual work.

One problem is, that the usage of cells enclosed by membranes, like it was necessary for the implementation of evaluation strategies and to represent heap structures, causes a considerable slow-down in the computation performance. The currently ongoing work on the SLIM LMNtal system aims to improve this situation.

Secondly, as discussed in Section 3.3 and others, the restrictions on logical links in LMNtal constrains our implementation very strongly. This was also the reason for the multiple approaches to the code generation scheme. According to a conversation with Kazunori Ueda, an introduction of multiple links is currently under consideration for LMNtal. Depending on its realisation, this could significantly simplify the representation of CCFL variables by means of links in the generated LMNtal code, maybe even up to getting rid of an explicit heap representation.

A third thing which turned out to be hindering is that the rule choice in LMNtal is completely based on pattern matching of the rule head (plus the application of guards).

The heap representation as presented in Chapter 5 was actually a simplification; for the following reasons: Links to arguments of terms may connect to the same or as well to different heap elements. The necessity to match such heap structures with a rule head forces us to either to provide an in general exponential number of rule copies (depending on the number and types of arguments) or to introduce additional indirections into the heap representation (which is the more general and thus, better, solution but slows down the performance again).

Consider the term (*foo a b*) resp. its representation (*el* (*cons* (*foo*) [*A*,*B*],*R*)). The links *A* and *B* may connect onto the same or different heap structures. To match on this term, we need two according rules (or the above mentioned indirections) as shown below. The links of the head of the first rule connect into different membranes, while the links in the second rule head connect in one common membrane.

```
el (cons (foo) [A,B],R),
{$pA,+A},
{$pB,+B} :-
  ...

el (cons (foo) [A,B],R),
{$pA,+A,+B} :-
  ...
```

A possible solution (which is, of course, very rough at the moment and needs a deeper consideration) may lead into the direction of additional guard atoms which "compute" on links and membranes resp. cells but do without pattern matching. At this, *ACell* and *BCell* may stand now for the same cell.

```
el (cons (foo) [A,B],R) :-
  link_in_cell (A,ACell), link_in_cell (B,BCell),
  ACell' = ACell \ {+A,+B},
BCell' = BCell \ {+A,+B} |
   ...
```

Finally, it turned out that programming without a type system yields to many avoidable but often hard to find programming mistakes.

7.3 Future work

There remain a number of tasks to consider in the future:

At first, there are many things to improve and to polish in the language design of CCFL and the compiler implementation which is currently just a simple prototype. Moreover, it will be interesting to consider the transfer of the implementation onto the new SLIM LMNtal system as compiler target. An extension of LMNtal by multiple links as mentioned above may yield a number of changes and simplifications for our compilation scheme.

Another source of important work in the future is the integration of new constraint solvers as sketched on in Chapter 6 either by the implementation of constraint solvers in LMNtal itself or by extending LMNtal by external solvers.

The ability to express a migration of rules within a process system to control the program evaluation has proved to be an interesting feature of LMNtal. We used this opportunity on the one hand when implementing evaluation strategies for CCFL. On the other hand we applied this method in the LMNtal module *show.lmn* (mentioned in Section 5.4) to ensure that the actual CCFL computation has been completely finished when the computation of user-readable representations from the result heap structure starts. We think that this feature of LMNtal could also be used to implement other extended language concepts like e.g. such from aspect-oriented programming.

Acknowledgement

The work presented in this report has been supported by a postdoctoral fellowship No. PE 07542 from the Japan Society for the Promotion of Science (JSPS) and my host professor Kazunori Ueda, Waseda University, Tokyo.

Bibliography

[Aik94] Alexander Aiken. Set Constraints: Results, Applications, and Future Directions. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, pages 326–335, 1994.

[AVWW07] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang.* Prentice Hall, 2nd edition, 2007.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, 2nd edition, 1998.

[CGKL98] Manuel M.T. Chakravarty, Yike Guo, Martin Köhler, and Hendrik C. R. Lock. Goffin: Higher-Order Functions Meet Concurrent Constraints. *Science of Computer Programming*, 30(1-2):157–199, 1998.

[Cho08] Choco: A Java Library for Constraint Satisfaction Problems, Constraint Programming and explanation-based Constraint Solving. `http://sourceforge.net/projects/choco`, 2008. last visited 07 March 2008.

[CHS+03] A.M. Cheadle, W. Harvey, A.J. Sadler, J. Schimpf, K. Shen, and M.G. Wallace. ECL^{i}PS^e – An Introduction. Technical Report IC-PARC-03-1, IC-PARC, Imperial College London, 2003.

[CvER90] Mantis H. M. Cheng, Maarten H. van Emden, and B. E. Richards. On Warren's Method for Functional Programming in Logic. In Peter Szeredi David H. D. Warren, editor, *Logic Programming, Proceedings of the Seventh International Conference, Jerusalem, Israel, ICLP 1990*, pages 546–560. MIT Press, 1990.

[FHGSP03] A.J. Fernández, T. Hortalá-Gonzáles, and F. Sáenz-Perez. Solving Combinatorial Problems with a Constraint Functional Logic Language. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium – PADL*, volume 2562 of *LNCS*, pages 320–338. Springer, 2003.

[GeC07] Gecode – Generic Constraint Development Environment. `http://www.gecode.org/`, 2007. last visited 07 March 2008.

[HAB+06] Michael Hanus, Sergio Antoy, Bernd Braßel, Herbert Kuchen, Francisco J. Lopez-Fraguas, Wolfgang Lux, Juan Jose Moreno Navarro, and Frank Steiner. Curry: An Integrated Functional Logic Language. Technical report, 2006. Version 0.8.2 of March 28, 2006.

[Hud00] Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

[Lei01] Daan Leijen. Parsec, a fast Combinator Parser, 4 Oct 2001. `http://legacy.cs.uu.nl/daan/parsec.html`, last visited 07 March 2008.

[LMN07] LMNtal PukiWiki. `http://www.ueda.info.waseda.ac.jp/lmntal/`, 2007. last visited 07 March 2008.

[LOMP05] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[Lor06] Florian Lorenzen. Eine abstrakte Maschine für eine nebenläufige (und parallele) Constraint-funktionale Programmiersprache, 2006. Diplomarbeit. Technische Universität Berlin.

[Lux01] Wolfgang Lux. Adding Linear Constraints over Real Numbers to Curry. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 2001, Proceedings*, volume 2024 of *LNCS*, pages 185–200. Springer, 2001.

[Moz06] The Mozart Programming System. `http://www.mozart-oz.org/`, 2006. last visited 07 March 2008.

[Nai91] Lee Naish. Adding equations to NU-Prolog. In *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 15–26. Springer, 1991.

[Nai96] Lee Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Department of Computer Science, University of Melbourne, 1996.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[PH06] Peter Pepper and Petra Hofstedt. *Funktionale Programmierung. Sprachdesign und Programmiertechnik.* Springer, 2006.

[PP97] Leszek Pacholski and Andreas Podelski. Set Constraints: A Pearl in Research on Constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–562. Springer, 1997.

[Smo93] Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming. Selected Research*, pages 405–419. The MIT Press, 1993.

[SR90] Vijay A. Saraswat and Martin C. Rinard. Concurrent Constraint Programming. In *17th ACM Symposium on Principles of Programming Languages – POPL*, pages 232–245, 1990.

[UK02] Kazunori Ueda and Norio Kato. Programming with Logical Links: Design of the LMNtal Language. In *Proceedings of the Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, pages 115–126, 2002.

[UK05] Kazunori Ueda and Norio Kato. LMNtal: a Language Model with Links and Membranes. In *Proceedings of the Fifth International Workshop on Membrane Computing (WMC 2004)*, volume 3365 of *LNCS*, pages 110–125. Springer, 2005.

[UKHM06] Kazunori Ueda, Norio Kato, Koji Hara, and Ken Mizuno. LMNtal as a Unifying Declarative Language. In Tom Schrijvers and Thom Frühwirth, editors, *Proceedings of the Third Workshop on Constraint Handling Rules*, Technical Report CW 452, pages 1–15. Katholieke Universiteit Leuven, 2006.

[War82] D.H.D. Warren. Higher-order extensions to PROLOG: Are they needed? *Machine Intelligence*, 10:441–454, 1982.