9 Pattern matching 24
  9.1 Guards and if-then-else statements ......................................... 24
  9.2 Abstractions .............................................................................. 24
  9.3 Cases ......................................................................................... 24

10 Reduction 27
  10.1 Data structures .......................................................................... 27
  10.2 Automaton .................................................................................. 27
  10.3 Environments .............................................................................. 27

III Appendices 29
A Standard prelude 29

List of Figures

1 An example program. ........................................................................ 4
2 Type inference rules ........................................................................ 20
3 Pattern inference rules ..................................................................... 21
4 Transforming patterns ...................................................................... 26

List of Tables

1 State transitions ............................................................................... 27
Part I
Language

1 Syntax

1.1 General structure

A program typically consists of declarations and an expression. This pair can be expressed in two ways: either a sequence of declarations followed by an expression, or as an expression followed by a where clause. The where clause will be explained later on in section 1.6.1. A short example program is shown in figure 1.

Program \rightarrow Expression
| Declarations in Expression.

1.2 Lexing

1.2.1 Alphanumeric identifiers

There are two types of identifiers. Identifiers of values, lower-id in the grammar, begin with a lower-case letter. Identifiers of constructors and types, upper-id in the grammar, begin with an upper-case letter. After the initial letter, alphanumeric and some punctuation characters are permitted.

\[\begin{align*}
\text{idchar} & = \text{[a-zA-Z0-9]} \\
\text{lower-id} & = \text{[a-z]}(\text{idchar})^* \\
\text{upper-id} & = \text{[A-Z]}(\text{idchar})^*
\end{align*}\]

1.2.2 Infix identifiers

There are four categories of infix identifiers, which correspond to four levels of precedence. An infix identifier is an infix character of a certain precedence level, followed by zero or more infix characters from any precedence level, e.g., the string :+= is an infix identifier of precedence 3; the string *|* is parsed as three tokens: *|* (an infix identifier of precedence 4), `,` (a comma), and * (an infix identifier of precedence 4).

\[\begin{align*}
\text{Infix-Id} & \rightarrow \text{infix1-operator} \\
& \mid \text{infix2-operator} \\
& \mid \text{infix3-operator} \\
& \mid \text{infix4-operator}.
\end{align*}\]

\[\begin{align*}
\text{infixchar} & = \text{[=<>\&!]*+/\%]} \\
\text{infix4-operator} & = [\text{*|}](\text{infixchar})^* \\
\text{infix3-operator} & = [\text{+\%}](\text{infixchar})^* \\
\text{infix2-operator} & = [\text{<>\&}])(\text{infixchar})^* \\
\text{infix1-operator} & = [\text{\&|}](\text{infixchar})^*
\end{align*}\]

1.2.3 Character and string literals

Character literals and string literals can be described together, as they expressed similarly. The difference between them is that character literals are delimited by ' (apostrophe) characters, whereas string literals are delimited by " (quotation mark) characters. Further, \ character must be escaped in character literals (but not in string literals), and the same vice versa holds for " characters. Finally, character literals consist of exactly one character; string literals consist of a non-negative number of characters.

\[\begin{align*}
\text{char-literal} & = '\text{\'} or \" or (basicchar)'' \\
\text{string-literal} & = "\text{\'} or \" or (basicchar)""
\end{align*}\]

A basic character literal, then, consists of any "printable" character (minus the ' and " characters, as described above), plus some escaped characters.

\[\begin{align*}
\text{basicchar} & = \text{(printable) or \" or [abfntrv]}
\end{align*}\]

The escape characters have the following meanings:
Branch :: [Tree a] -> Tree a.
Leaf :: a -> Tree a.

BTBranch :: (BTTree a, BTTree a) -> BTTree a.
BTLeaf :: a -> BTTree a.

to_binary_tree :: Tree a -> BTTree a =
% (Leaf x) => BTLeaf x
| (Branch [x]) => to_binary_tree x
| (Branch [x, y]) => BTBranch (x', y') where {
x' = to_binary_tree x.
y' = to_binary_tree y
} | (Branch (x:y:z)) =>
to_binary_tree (Branch (Branch [x, y]:z))
| (Branch []) => Undefined.

fold_tree :: (((b, b) -> b), (a -> b), BTTree a) -> b =
% (_, 1, BTLeaf x) => 1 x
| (br, 1, BTBranch (x, y)) => br (x', y') where {
x' :: b = fold_tree (br, 1, x).
y' :: b = fold_tree (br, 1, y)
}.

sum_tree :: Tree Int -> Int =
% t => fold_tree (%(a, b) => a + b,
%x => x, to_binary_tree t).

build_tree :: Int -> Tree Int =
% x => Branch [Branch [t1, t2, t3]] where {
t1 :: Tree Int = Branch [Leaf 3, Leaf 9].
t2 = Branch [Leaf 7].
t3 = Leaf (-15 * (if x < 0 then 1 else x))
} in

build_tree 5 ; sum_tree ; fact where {
    /* note fact is deliberately undefined for n < 0 */
    fact :: Int -> Int =
      % 0 => 1
    | n | n > 0 => n * fact (n - 1)
    | ... => Undefined
}
1.3 Types

Each value must be declared explicitly with a type.

```
Type    →  Type-2  →  Type  
  |       Type-2. 
Type-2  →  upper-id  Type-3+ 
  |       Type-3. 
Type-3  →  upper-id  
        |  lower-id  
        |        (  
        |         ( Type , TypeList )  
        |            [ Type ]  
        |            ( Type ).  
TypeList →  Type , TypeList  
        |  Type. 
```

The seven types of types are:

1. **Arrow type** — a function taking one argument and returning one value, e.g., `Int` → `Bool`;

2. **Basic type** — where the leading type name is polymorphic, instantiate it with a serious of other types, e.g., `Tree` `Int`; gives a `Tree of Ints`. Otherwise, an atomic type name, e.g., `Char`;

3. **Polymorphic type** — a polymorphic type name to be substituted by any type, e.g., `a`;

4. **Unit type** — the unit type, i.e., the type of the empty tuple;

5. **Tuple type** — tuples, e.g., `(Int , a → [b] , Bool)`;

6. **List type** — special syntax for the `List` type.

1.3.1 Types of infix operators

Both constructors (see section 1.4.1) and values (see section 1.4.2) allow declaration of new infix operators. The types of these must be severely restricted. Specifically, since infix operators are necessary binary, the infix operator must be binary:

```
InfixType  →  ( Type , Type ).
```

A newly constructed infix operator must be of type `InfixType` → `Type` or `InfixType` → `ConsType` appropriately.

1.4 Declarations

Declarations in a series are separated (note, not terminated), by the period character “.”. All declarations are omnipresent in their scope, i.e., one need not declare a symbol before referencing it.

```
Declarations  →  Declaration ( . ) Declarations  
  |       Declaration ‘,’  
  |       Declaration.
Declaration  →  Constructor  
  |  Value.
```
1.4.1 Data type declarations

Data types are not declared explicitly. Rather, constructors are declared, which look very much like function declarations but without any body. Data types are therefore defined by all the constructors gathered with the appropriate return type.

\[
\text{Constructor} \rightarrow \text{upper-id :: ConsType} \\
| \text{upper-id :: Type-3 -> ConsType} \\
| \{? \text{Infix-Id } \} :: \text{Infix-Type -> ConsType.}
\]

\[
\text{ConsType} \rightarrow \text{upper-id} \\
| \text{upper-id lower-id*.}
\]

That is, the type of a constructor must be restricted. First, it must take only either zero or one argument, the reason being that currying constructors is conceptually invalid. Secondly, it must be well-defined which type we’re actually trying to define, so returning, e.g., a tuple, or some other complex type, is nonsensical.

Infix constructors Infix constructors must be binary, and are declared by wrapping the constructor name in braces, e.g.,

\[\{++\} :: (\text{Int}, \text{String}) \rightarrow \text{X.}\]
\[A :: \text{X.}\]

\[
\text{case } 3 \text{ ++ "foo" of} \\
\text{+:: x => some_expression}
\]

1.4.2 Value declarations

Very much like constructors, but each has an associated expression which gives it its “value”.

\[
\text{Value} \rightarrow \text{lower-id :: Type = Expression} \\
| (\text{Infix-Id } ) :: \text{InfixType -> Type = Expression} \\
| \text{lower-id} = \text{Expression} \\
| (\text{Infix-Id } ) = \text{Expression.}
\]

Note that, unlike constructors, values need not be explicitly typed, in which case the type must be derived.

Infix value declarations Infix operators are wrapped in parentheses when declared, e.g.,

\[(++) = \% (\[], b) \Rightarrow b \\
| (a:as, b) \Rightarrow a:(as ++ b).
\]

1.5 Patterns

Pattern-matching is done when writing abstractions (see section 1.6.2) and cases (see section 1.6.3). Patterns between the two is slightly different — namely abstractions use level-2 patterns and cases use level-1 patterns — but this is just a syntactic anomaly to avoid ambiguities in the grammar. The reason for this is that abstraction patterns allow a sequence of patterns separated by spaces (as a convenience for abstractions with more than one argument), but constructors themselves allow sub-patterns separated by spaces; thus, level-2 patterns — those used in abstractions — force parentheses around constructor patterns with sub-patterns. Level-1 patterns form a superset of level-2 patterns syntactically, and they are equivalent in expressiveness. At their core, the patterns are similar, though.
Each pattern matches a datum as given:

1. **Complex constructor type** — a constructor with associated data, i.e., a constructor with an argument

2. **Simple constructor type** — a constructor with no associated data

3. **Infix constructors**

4. **Variable** — matches any datum, and binds the given variable in the pattern's associated expression

5. **Wild card** — matches any datum

6. **Unit** — the empty-tuple

7. **Tuple**

8. **Empty list type** — convenience syntax for matching the list’s Nil constructor

9. **List type** — convenience syntax for matching lists

10. **Integer** — matches a integer datum against a specific value

11. **Character** — matches a character datum against a specific value

12. **String** — convenience syntax for matches lists of Char

1.5.1 **Pattern Expression Lists**

**Function Pattern Expression Lists** Pattern expression lists used in abstractions (or "functions") allow multiple patterns per pattern-expression in order to accommodate multiple arguments via currying. Thus, to avoid ambiguity, Pattern2 patterns must be used.
FxnpatternExprList \rightarrow FxnPatternExpr \mid FxnPatternExpr \mid FxnPatternExpr ‘|’ FxnPatternExprList
| FxnPatternExpr.
FxnpatternExpr \rightarrow PatternList \Rightarrow Expr1
| PatternList GuardList.
PatternList \rightarrow Pattern2 PatternList
| Pattern2.

Following are two examples of function pattern expressions:

\[
\begin{align*}
\% \_ \_ (\text{Branch } x) & \Rightarrow \text{expr1} \\
\mid \_ \_ & \Rightarrow \text{expr2}.
\% 0 & \Rightarrow 0 \\
\mid n & \mid n < 0 \Rightarrow n + 5 \\
\mid \_ & \Rightarrow f n
\end{align*}
\]

1.5.2 Case Pattern Expression Lists

Case patterns take a sole pattern per expression, which allows patterns on constructors to be written out without explicit parentheses. Examples of case pattern expressions:

\[
\begin{align*}
case \ e \ of \\
& \text{Branch } [x] \Rightarrow \text{expr1} \\
& \mid \text{Leaf} \Rightarrow \text{expr2} \\
& \mid \_ \Rightarrow \text{expr3}
\end{align*}
\]

\[
\begin{align*}
case \ e' \ of \\
& 0 \Rightarrow x + 4 \\
& \mid n & \mid n < 0 \Rightarrow \text{expr4} \\
& \mid \_ & \Rightarrow \text{Undefined}
\end{align*}
\]

PatternExprList \rightarrow PatternExpr \mid PatternExpr \mid PatternExpr ‘|’ PatternExprList
| PatternExpr.
PatternExpr \rightarrow Pattern1 \Rightarrow Expr1
| Pattern1 GuardList.

1.5.3 Guards

Guards have been touched on briefly via example. Guards allow “pattern-matching” based on predicates instead of constructors. In the case:

\[
\begin{align*}
case [3, 5] \ of \\
& \_ \Rightarrow 0 \\
& \mid [x] & \mid x < 3 \Rightarrow x \\
& & \mid \_ \Rightarrow x + 5 \\
& \mid [x, y] & \mid y < x \Rightarrow y + x \\
& & \mid y > x \Rightarrow y - x \\
& & \mid \_ \Rightarrow \text{Undefined}
\end{align*}
\]

the third pattern \([x, y]\) matches the expression. The expression then matches each of the predicates of the guards and evaluates to the expression corresponding to the appropriate guard.

GuardList \rightarrow \text{Guard+ ‘|’ } \ldots \Rightarrow \text{ExprInfl},
Guard \rightarrow ‘|’ \text{ExprInfl} \Rightarrow \text{ExprInfl}.
1.6 Expressions

Expression → Abstraction
  | CaseExpr
  | IfExpr
  | Expr1.
Expr1 → ExprInf1 where \{'? Declarions \}'
  | ExprInf1.
ExprInf1 → ExprInf2 infix1-operator ExprInf1
  | ExprInf2.
ExprInf2 → ExprInf3 infix2-operator ExprInf2
  | ExprInf3.
ExprInf3 → ExprInf4 infix3-operator ExprInf3
  | ExprInf4.
ExprInf4 → Expr2 infix4-operator ExprInf4
  | Expr2.
Expr2 → AppList
  | Exp3.
Expr3 → Application
  | upper-id Expr4
  | Expr4.
Expr4 → lower-id
  | upper-id
  | Undefined
  | Literal
  | ( Expression )

1.6.1 Where clause

A where clause allows one declarations which exist only in the scope of the associated expression. Each symbol declared within the associated declarations is present within the entirety of the where clause, i.e., the associated expression and all of the associated declarations.

1.6.2 Abstractions

Abstractions are similar to a case. When applied to a sequence of expressions (its arguments), the abstraction will match the arguments against the patterns given and will accordingly evaluate to an expression.

The \% token was chosen due to its semblance to the \(\lambda\) from \(\lambda\)-calculus.

Abstraction → % FxnPatternExprList.

1.6.3 Cases and ifs

CaseExpr → case ExprInf1 of PatternExprList.
IfExpr → if ExprInf1 then ExprInf1 else ExprInf1.

Cases and ifs are treated identically. if \(a\) then \(b\) else \(c\) is exactly synonymous to case \(a\) of True = \(b\) | _. = \(c\).

1.6.4 Applications

An application applies an abstraction to one argument. As an application often yields an abstraction, these can be chained together, with implicit binding to the left, to form an application with more than one argument.

One should note that the syntax for constructions is deceptively complex. Complex constructions — those with an argument — are at level 3 precedence so that applications involving complex constructions
force parentheses and avoid ambiguity. Simple constructions — those without any argument — offer no ambiguity and so are at level 4 precedence.

\[
\begin{align*}
\text{AppList} & \rightarrow \text{AppList} \text{'} ; \text{Expr} 3 \\
& | \text{Expr} 3 \text{'} ; \text{Expr} 3 . \\
\text{Application} & \rightarrow \text{Expr} 4 \text{Expr} 4 +
\end{align*}
\]

That is, apply an abstraction or constructor to all of its arguments. The ;, or flip, syntax allows one to write chains of combinators in sequence, e.g.,

\[
a \ (b \ (c \ (d \ (e \ (f \ (g \ (h \ (i \ (j \ (k \ (l \ (m \ (n \ (o \ p)))))))))))]))
\]

can be better rewritten as:

\[
op ; mn ; l ; k ; ij ; fg ; h ; e ; cd ; b ; a
\]

which can be read as “perform \(op\), then perform \(mn\) on the result, then perform \(l\) on the result of that, and so on.”

1.6.5 Literals

\[
\begin{align*}
\text{Literal} & \rightarrow \text{string-literal} \\
& | \text{char-literal} \\
& | \text{int-literal} \\
& | \text{ListLiteral} \\
& | \text{Tuple}.
\end{align*}
\]

The lexical structure of string literals and character literals are explained in section 1.2.3. Int literals are expressed as expected, with the - character acting as unary negation.

\[
\begin{align*}
\text{ListLiteral} & \rightarrow [ ] \\
& | [ \text{ExprList} ]. \\
\text{Tuple} & \rightarrow ( ) \\
& | ( \text{Expression} , \text{ExprList} ). \\
\text{ExprList} & \rightarrow \text{Expression} , \text{ExprList} \\
& | \text{Expression}.
\end{align*}
\]

1.7 Pre-processing

Pre-processing of each program loaded in is processed by the standard Unix macro processor m4. For example, to include another file, such as the standard prelude (see section 4), the source file might contain:

```c
#include('prelude.lambda')
/* rest of program file */
```
2 Built-in datatypes and functions

2.1 Undefined expressions

The keyword Undefined can be used in place of any expression. An Undefined expression shall not affect the
typing of an expression as a whole; i.e., Undefined is of "every" type, as the situation warrants. Undefined
applied to another expression, as in an application or construction, shall yield Undefined.

The concept behind Undefined is that all patterns in the language, whether in abstractions or cases,
or in guards, must be complete. That is, every case that is undefined must be explicit. A program that
evaluates to Undefined shall print out an error message: Undefined on line y, where y is the line number
where the given Undefined was written in the source.

2.2 Datatypes

There are four built-in datatypes, the Char, Int, Bool and List. The type String also exists, but is an
alias for type [Char], i.e., List Char.

2.2.1 Char

The Char datatype represents all possible characters. One can consider the following pseudo-code implicit
in every program file:

'\a' :: Char.
'\b' :: Char.
...
'\v' :: Char.
', ' :: Char.
'!' :: Char.
...
'\1' :: Char.
'0' :: Char.
...
'9' :: Char.
'A' :: Char.
...
'Z' :: Char.
'a' :: Char.
...
'z' :: Char.

That is, every printable character and every defined control character is implicitly defined as type Char.

2.2.2 Int

The Int datatype represents all integers. Without resorting to unary representing (defining Int by constructors Zero and Succ), it is actually impossible to explicitly define Int: the set of integers is infinite. Thus,
each integer literal can be thought of as implicitly being a constructor to the type Int, as in:

... 
-2 :: Int.
-1 :: Int.
0 :: Int.
1 :: Int.
2 :: Int.
...
2.2.3 Bool
The Bool type represents the boolean values of truth and falsehood. It consists of the two constructors True and False.

2.2.4 List
The list type is built-in due to its importance in functional programming, and due to its connection to the syntax—the types List a and [a] are synonymous—and the String type.

Nil :: List a.
{()} :: (a, List a) \to List a.

# or, using alternative syntax:
Nil :: [a].
{()} :: (a, [a]) \to [a].

3 Functions
There are few functions built in. Any functions which can be defined in the language itself are defined as part of the standard prelude (see section 4). The following signatures represent built-in functions, and their semantics are self-explanatory:

/* general functions */
(==) :: (a, a) \to Boolean.
(<>): (a, a) \to Boolean.

/* integer functions */
(+) :: (Int, Int) \to Int.
(->) :: (Int, Int) \to Int.
(*) :: (Int, Int) \to Int.
(/) :: (Int, Int) \to Int.
(<) :: (Int, Int) \to Boolean.
(<=) :: (Int, Int) \to Boolean.
(>) :: (Int, Int) \to Boolean.
(>=) :: (Int, Int) \to Boolean.

4 Prelude
Loaded before any program file should be the standard prelude (see section 1.7 on loading the standard prelude), which defines commonly used functions. See appendix A for the standard prelude.
Part II
Implementation

5 Data structures

5.1 Syntax tree

Before any semantics calculations, the following syntax tree is used, which follows directly from the parse tree:

\[
\begin{align*}
\text{LambdaExpr} & = \text{WhereExpr of [LambdaDec]}, \text{LambdaExpr} \\
& \mid \text{GuardExpr of [LambdaExpr], LambdaExpr} \\
& \mid \text{AbsExpr of [[LambdaPatt], LambdaExpr]} \\
& \mid \text{AppExpr of LambdaExpr, LambdaExpr} \\
& \mid \text{ConsExpr of String, Option LambdaExpr} \\
& \mid \text{VarExpr of String} \\
& \mid \text{CaseExpr of LambdaExpr,} \\
& \quad [\text{LambdaPatt}, \text{LambdaExpr}] \\
& \mid \text{TupleExpr of [LambdaExpr]} \\
& \mid \text{CharLiteral of Character} \\
& \mid \text{IntLiteral of Integer} \\
& \mid \text{UndefinedExpr of Integer}. \\
\text{LambdaType} & = \text{ArrowType of LambdaType, LambdaType} \\
& \mid \text{BasicType of String, [LambdaType]} \\
& \mid \text{PolyType of String} \\
& \mid \text{TupleType of [LambdaType].} \\
\text{LambdaDec} & = \text{Constructor of String, Option LambdaType,} \\
& \quad (\text{String, [String]}) \\
& \mid \text{Value of String, LambdaExpr,} \\
& \quad \text{Option LambdaType}. \\
\text{LambdaPatt} & = \text{ConsPatt of String, Option LambdaPatt} \\
& \mid \text{TuplePatt of [LambdaPatt]} \\
& \mid \text{Wildcard of Option String} \\
& \mid \text{ListPatt of [LambdaPatt]} \\
& \mid \text{IntPatt of Integer} \\
& \mid \text{CharPatt of Character}. \\
\end{align*}
\]

5.2 Internal data structures

These data structures are based largely on those given for the syntax in Marlow's thesis[Mar95].

\[
\begin{align*}
\text{Term id} & = \text{Variable of id} \\
& \mid \text{Abstraction of id, Term id} \\
& \mid \text{Application of Term id, Term id} \\
& \mid \text{Construction of id, Option (Term id)} \\
& \mid \text{Tuple of [Term id]} \\
& \mid \text{IntegerValue of Integer} \\
& \mid \text{Case of Term id, [PatternExpr id],} \\
& \quad \text{Pattern id, Term id.} \\
\text{PatternExpr id} & = \text{Pattern id, Term id.} \\
\text{Pattern id} & = \text{ConstructorPattern of id, Option (Pattern id)} \\
& \mid \text{TuplePattern of [Pattern id]} \\
& \mid \text{IntegerPattern of Integer} \\
& \mid \text{WildcardPattern of Option id}. \\
\end{align*}
\]

They are designed to be terribly simple, hardly more complex than \(\lambda\)-terms. One major advantage to this is that evaluation of a program is quite simple, too, not much more complicated than a \(\beta\)-reduction.
id acts as the unique identifier key in the data structure. In the implementation of the parse tree, strings are used for id, corresponding to the variable names.

5.2.1 Terms
As said before, the term data structure is derived from the λ-calculus. Only a few extra terms are needed.

Variable $x$ — a variable that is bound in an abstraction or case
Abstraction $x, t$ — the term $t$ is bound by $x$
Application $u, t$ — $u$ applied to the term $t$
Construction $a, t$ — a datum where $a$ is the constructor and $t$ is the argument to the constructor, if there is one
Tuple $ts$ — the tuple value made up of the given sub-terms
IntegerValue $t$ — representing integers in unary, i.e., via the constructors Zero and Succ or similar, is unwieldy, so integers are stored natively as a special case
Case $t, ps$ — case on $t$, matching the patterns $ps$ with the result
6 Lambda Lifting

The transformation of lambda lifting is designed to "lift" (hence its name) an abstraction up to the top scope. For example:

\[ f = \% n \Rightarrow g \cdot n \cdot n \text{ where } \begin{cases} \ g = \% m \Rightarrow n / 2 + m \end{cases} \text{ in } \]
\[ f \ 4 \]

The definition of \(g\) in this example depends on the variable \(n\), which is defined in a higher scope. Specifically, it is bound by the abstraction assigned to \(f\). Via lambda lifting, this is transformed as:

\[ g = \% n \cdot m \Rightarrow n / 2 + m. \]
\[ f = \% n \Rightarrow g \cdot n \cdot n \text{ in } \]
\[ f \ 4 \]

The transformed program is semantically equivalent to the earlier example. However, all abstractions are now given on the top level. This step is critical to flattening, which will be described in section 7.

The process of lambda lifting is broken up into three functions — representing three processes — abstract, rename and collectSCs. Johnson-style lambda lifting is implemented. The process of simple lambda lifting consists of composing these three functions together: \texttt{lambdaLift = collectSCs \circ rename \circ abstractrename}. It should be of note that Peyton Jones and Lester\texttt{[JL92]} leave off the final rename function; i.e., Johnson-style is implemented analogous to \texttt{collectSCs \circ abstract \circ rename}. However, the abstraction stage does introduce new supercombinators for unbound abstractions, which clearly need to be renamed before being collected.

6.1 Abstracting

The process of abstraction is to disassociate abstractions from any bindings. In the example given above, \(g\) needs to be abstracted of the binding \(n\). Hence, it gets transformed to:

\[ g = \% n \cdot m \Rightarrow n / 2 + m \]

For this, not only are the free variables of \(g\) needed, but also the free variables of any functions \(g\) might call. This is done by calculating the reflexive transitive closure.

Abstraction takes place only upon where clauses: once all named abstractions (i.e., declared values with abstractions as their right-hand-sides) are further abstracted, all calls within the declarations and in the associated expression itself are transformed such that all calls to declared abstractions include further arguments, e.g., consider how the example above gets transformed:

\[ f = \% n \Rightarrow g \cdot n \cdot n \text{ where } \begin{cases} \ g = \% n \cdot m \Rightarrow n / 2 + m \end{cases} \text{ in } \]
\[ f \ 4 \]

For expressions other than where clauses, no transformation happens save recursing on sub-expressions. One exception to this is unnamed abstractions. For example, consider the expression:

\[ f \ (\% f \cdot n \Rightarrow f \cdot n + f \cdot (n + 2)) \]

The abstraction is unnamed, but still needs to be lifted. This is transformed as follows:

\[ f \ sc \text{ where } \begin{cases} \ sc = \% f \cdot n \Rightarrow f \cdot n + f \cdot (n + 2) \end{cases} \]

What is needed now is a mechanism to bring the abstractions to the top level.

6.2 Renaming

The step of renaming is a necessary, if terribly mundane, step in the process of lambda lifting. The process of abstracting naïvely assigns the name of \(sc\) to every supercombinator it creates. The rename functions recursively descends on each expression, ensuring every symbol, excepting those bound by abstractions and cases, is given a unique name.
6.3 Collecting

The `collectSCs` function — mnemonically “collect supercombinators” — collects all abstractions defined in a where clause and moves it up one scope. This is implemented abstractly as:

```haskell
collectSCs e =
case scs of
  [] => e'
  _ => WhereExpr (scs, e') where
    scs = local-scs ++ scs-of-subexpressions
    e' = e after scs of subexpressions have been collected
```

As the renaming process exists, there is no possibility of name collision.

The only non-obvious part of this process is finding local supercombinators. If the expression is not a where expression, this is actually trivial: there are none. In the case of a where clause, though, this is accomplished by filtering out those declarations (after recursively lambda-lifting them): any abstraction is a supercombinator and is lifted out.

After abstracting, renaming and collecting, the example given at the top of the section transforms to:

```haskell
sc = % m n => n / 2 + m.
f = % n => g n * n where { g = sc n } in
f 4
```
7 Flattened form

Before any typing (see section 8) or reduction can take place, the syntax tree must be converted to flattened form. A program in flattened form consists of a set of declarations and a single expression, where each of the declarations and the expression is “flattened”, i.e., without any where clauses (see section 1.6.1). This is accomplished by flattening all declarations declared at the top-level into a single top-level where clause:

\[
\begin{align*}
\text{flatten}_\text{program} \ e &= \text{flatten}_\text{expr} \\
\text{flatten}_\text{decl} \ \text{Constructor} \ c &= [\text{Constructor}] \\
\text{flatten}_\text{decl} \ \text{Value} \ (s, e, t) &= \text{Value} \ (s, e', t) : d \ \text{where} \ (d, e') = \text{flatten}_\text{expr} \ e
\end{align*}
\]

\text{flatten}_\text{expr} \ then \ flattens \ all \ where \ clauses, \ and \ recursively \ flattens \ where \ clauses \ in \ sub-expressions, \ to \ return \ a \ list \ of \ declarations \ and \ a \ “flattened” \ expression — one without any where clauses or where clauses in sub-expressions.

7.1 Removing where clauses

For our definition of \text{flatten}_\text{expr} \ WhereExpr \ (d, e), the assumption is made that program has already renamed as described in section 6.2 with respect to lambda lifting. This is practical since lambda lifting precedes flattening in the implementation.

We will consider the following example:

\[
\begin{align*}
% \ x & \Rightarrow g \ y \ 3 + g \ y \ 4 \ \text{where} \ \\
& \{ \\
& \quad v = x * x, \\
& \quad y = v * v \\
& \}
\end{align*}
\]

The process of flattening a where clause is broken into three stages.

7.1.1 Abstracting

As in lambda lifting in section 6.1, declarations in a where clause are viewed as a group, and the reflexive transitive closure of their free variables is calculated. In this example, the free variables of \(v\) are \(\{x\}\) and the free variables \(y\) are \(\{v, x\}\); note that \(x\) is free in \(y\) transitivity through \(v\).

Any values in the where clause are then abstracted according to the free variables, e.g., \(v\)'s body becomes \% \(x \Rightarrow x * x\) and \(y\)'s body becomes \% \(v \Rightarrow v * v\).

7.1.2 Applying

After the abstracting stage, the program is not even semantically correct anymore. What is needed is to transform references to the local symbols into applications that pass in unbound variables. To avoid unnecessary computation, it is crucial at this stage to perform the application in the correct spot. The application is placed inside any binding. As it is assumed that lambda lifting has already happened, there is only one place where this happens, and thus there will not be multiple applications.

Our above example gets transformed as:

\[
\begin{align*}
% \ x & \Rightarrow C1 \ v \ y \Rightarrow g \ y \ 3 + g \ y \ 4) \ (v \ x) \ (y \ v \ x) \ \text{where} \ \\
& \{ \\
& \quad v = \% \ x \Rightarrow x * x, \\
& \quad y = \% \ v \ x \Rightarrow (\% \ v \Rightarrow v * v) \ (v \ x) \\
& \}
\end{align*}
\]
7.1.3 Collection

As each local definition is no longer dependent on local bindings, it can be collected and floating to the
top-level.

Our above example gets transformed to:

\[ v = \% x \Rightarrow x \ast x. \]
\[ y = \% v x \Rightarrow (\% v \Rightarrow v \ast v) (v x) \in \]
\[ \% x \Rightarrow (\% v y \Rightarrow g y 3 + g y 4) (v x) (v y x) \]

It should be of note that although \( v \) is not used in the body of \( x \), even though it is bound there. This
kind of sucks.

7.2 Other expressions

Performing \( flatten_{expr} \) on any kind of expression other than a \( WhereExpr \) becomes trivial, as none has any
declarations at the top scope. The operation then becomes recursing on sub-expressions if necessary.
8 Typing

To ensure semantic correctness, typing of the program is performed on the parse tree. The framework for typing, and the unification algorithm, comes from Vesely's work on Charity [Ves97].

The general method of typing is to collect information given in the program about possible types. An attempt is then made to prove that the program has a type. If any proof can be found, the program is typed correctly; if no proof can be found, then obviously the program cannot be typed.

Typing a program is centred around type equations. The following data structures are used to represent the type equations:

Type = Arrow of Type, Type
      | Tuple of [Type]
      | Basic of String, [Type]
      | Abstraction of [Integer], Type
      | Variable of Integer.

Constraint = String, Type.
TypeEquation = TypeVar, TypeVar.

8.1 Collecting type constraints

Explicit type information may be given in the declarations of a program. Type information gleaned from constructors, as shown in the table below, is used in the context when deriving proofs.

\[ C :: t \Rightarrow (C, t) \]
\[ C :: a \to t \Rightarrow (C, \text{Arrow}(a, t)) \]

In addition to any type constraints that can be derived from the program definition, built-in definitions must also be considered a part of the context when deriving a proof. Appropriate definitions for : (aka Cons), Nil, True and False are part of the context. This "standard constraint set", denoted by \( \Gamma \), from hereon, is equal to \( \{(a, [a]) \Rightarrow [a], \text{Nil} : [a], \text{True} : \text{Bool}, \text{False} : \text{Bool}\} \).

Type constraints given for functions are not used when deriving proofs. Rather, these constraints are used during unification merely to ensure the derived type for a given function matches its explicit constraints.

8.2 Symbol table

All functions are assigned type variables. These type variables are not part of the context, but rather form a symbol table. When equations are derived for the body of a function, the are assigned equal to the variable that is given to the function name in the symbol table.

However, when a function call is found, it is incorrect to assume that the type of function call must be equal to the type of the function definition. Rather, it is sufficient that the type of the function call be at least as specific as the function definition. For example, if function \( f \) has type \( F \), and while deriving the proof we discover that a call to function \( f \) must be of type \( T \), we create a new type inequality \( F \gg T \), read as "\( F \) is at least as general as \( T \)".

8.3 Collecting type equations

A new type variable \( A \) is assigned to the expression, which forms a sentence in the context collected in section 8.1.

For example, consider the flattened program:

\[ A :: T_1. \]
\[ B :: (T_1, \text{Int}, T_1) \to T_1. \]
\[ f :: \text{Int} \to T_1 = \% x \Rightarrow B (A, x, A). \]
\[ g = \% x y \Rightarrow x + y \text{ in} \]
\[ f (g 3 4) \]
<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Inference Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projection</td>
<td>( \Gamma, x : A \vdash x : B \vdash A : B )</td>
</tr>
<tr>
<td>Integer projection</td>
<td>( \Gamma \vdash i : A \vdash A = \mathbb{Z} )</td>
</tr>
<tr>
<td>Character projection</td>
<td>( \Gamma \vdash \alpha' : A \vdash A = \text{Char} )</td>
</tr>
<tr>
<td>Abstraction branching</td>
<td>( \Gamma \vdash \lambda \frac{p_1}{p_n} : A )</td>
</tr>
<tr>
<td>Abstraction</td>
<td>( \Gamma, x_1 : C_1, \ldots, x_n : m : A \vdash B, C_1 \vdash A = C_1 \rightarrow \cdots \rightarrow C_n \rightarrow B )</td>
</tr>
<tr>
<td>Case branching</td>
<td>( \Gamma \vdash e : B \rightarrow B, C \vdash A = B \rightarrow C )</td>
</tr>
<tr>
<td>Guard branching</td>
<td>( \Gamma \vdash q_i : B, \Gamma \vdash e_i : A \vdash B \vdash B = \text{Bool} )</td>
</tr>
<tr>
<td>Application</td>
<td>( \Gamma \vdash m : B \rightarrow A, \Gamma \vdash n : B \vdash B \rightarrow m n : A )</td>
</tr>
<tr>
<td>Tuple</td>
<td>( \Gamma \vdash x_i : B_i \vdash (x_1, \ldots, x_n) : A \vdash B_i \vdash A = B_1 \times \cdots \times B_n )</td>
</tr>
<tr>
<td>Function call</td>
<td>( (f :: F \in \text{SymbolTable}) \Gamma \vdash f : A \vdash F \vdash A )</td>
</tr>
<tr>
<td>Undefined</td>
<td>( \Gamma \vdash \text{Undefined} : A )</td>
</tr>
</tbody>
</table>

Figure 2: Type inference rules
This forms the sentence:
\[ \Gamma_{st} : A : T1, B : (T1, Z, T1) \Rightarrow T1 + f(3, 4) : A \]

This base sentence is the final statement in the proof. The goal of collection is then to attempt to build a proof on top of this that is consistent with the information in the program, and concludes the statement.

Figure 8.3 shows the rules of inference used to construct a proof. Note that the abstraction rule places patterns in the context. These patterns need to be deconstructed further. The inference rules for constructing contexts are shown in figure 8.3.

8.4 Unification

Before unification starts, we transform our list of inequalities into lists of lists of inequalities. Each inequality stems from one function call, and it is necessary to keep all inequalities separate which refer to the same function call.

At the time of unification, we have three data: a list of type equalities and inequalities, a list of explicit type constraints, and the variable of the final type we’re interested in. The process of unifying a type equality in with already-collected type equation — i.e., the unify function — will not be discussed here, is it verbatim the algorithm discussed in [Ves97]. However, this is just one small part of the unification process.

From unify we get a list of type equation, of the form \( A = T \) where \( A \) is necessarily a type variable and \( T \) is a possibly complex type. Before unifying in another type equality, it is necessary to perform substitutions on the newly got type equations.

Consider that we wish to substitute \( A = T \). For the type equality \( U = V \), this is simply \( U[T/A] = V[T/A] \). For substituting into inequalities of the form \( U > V \), this is done in two parts. First, this is transformed simply into \( U > V[T/A] \); next, if \( A \) is \( U \), then the inequality must be broken down. \( V' \) is constructed as \( V'[T/A] \) with all fresh type variables and the type equality \( T = V' \) is added to the list of type equalities yet to consume.

Further, if \( T \) is a type variable, then the inequality \( T > V' \) is added. Otherwise, i.e., \( T \) has structure of the form \( S(T_1, \cdots, T_n) \), then the inequalities must be formed as:

- if \( V' \) is of the form \( S(V_1, \cdots, V_n) \), then \( T_1 > V_1, \cdots, T_n > V_n \) are added and broken down recursively, as needed;

\[ A = Y \Rightarrow Z \]
• if \( V' \) is a type variable, then we create \( n \) fresh type variables \( W_1, \ldots, W_n \), add the type inequalities \( T_1 \triangleright W_1, \ldots, T_n \triangleright W_n \) and break them down recursively as needed, and further we add the type equality \( V' = S(W_1, \ldots, W_n) \);

• otherwise, i.e., if \( V' \) is of a different structure from \( T \), then a type mismatch has occurred and unification fails.

Further, if \( A \triangleright T \) and \( A \triangleright U \) in the same list of inequalities, i.e., for the same function call, then \( A \triangleright U \) is removed from the list of inequalities, and \( U = T \) is added to the list of equalities.

Substitution into type constraints is similar to substitution into inequalities except that new equalities to try to make the constraints more general can never occur. For example, substitution \( A = T \) into the type constraint \( F : U \) takes the form of:

• if \( A \neq F \), no action takes place;

• if \( A = F \) and \( T = S(T_1, \ldots, T_n) \) and \( U = S(U_1, \ldots, U_n) \) for some structure \( S \), then the type constraints \( T_1 : U_1, \ldots, T_n : U_n \) are added and broken down recursively until a type variable appears on the left-hand-side of the ::;

• otherwise, a constraint violation has occurred, and unification fails.

### 8.4.1 Consistency checking

In order to guarantee termination, a cycles among type equalities and inequalities needs to be considered. Consider, for example, the list of equalities \([A = S(B), C = S(D)]\) and the list of inequalities \([A :> D], [C :> B]\). Performing unification as described will not terminate. Two more lists of inequalities are kept for each type variable \( A \): those type variables that are greater than (containing more structure) \( A \), and those that are at least as great as \( A \). Every time a new equality or inequality is introduced, the lists are augmented via transitivity.

For example, \([A = S(B), C = S(D)], [A :> D], [C :> B]\) yields the list of inequalities \( A > B, C > D, D \geq A, B \geq C \). Working through these inequalities in order, we build up the following table of lists:

<table>
<thead>
<tr>
<th></th>
<th>( &gt; )</th>
<th>( \geq )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>B, D</td>
</tr>
<tr>
<td>D</td>
<td>A, B, C, D</td>
<td></td>
</tr>
</tbody>
</table>

It is an error for \( X > X \) for any \( X \). In this example we find that \( D > D \) and signal an error.

One may question why consistency checking is used in place of more traditional algorithms, e.g., generating call graphs and considering equations from strongly connected components as a whole, such as those used in Haskell implementations. The net effect from treating connected components as a single unit is that any derived types of function calls are interpreted as a definition for the type itself. This leads to types which are unnecessarily strong. For example, consider the following Haskell program:

\[
g \_ \_ = 0
f \ (a:as) \ (b:bs) = g \ a \ b + f \ bs \ as
\]

This yields the type of \([a] \rightarrow [a] \rightarrow \text{Int}\), which is too strict. One could, for example, consider calling this function with a list of Ints and a list of Chars and have the program compute successfully. Because the call to \( f \) is given within the definition of \( f \), it in part defines the type of \( f \).

For contrast, consider the analogous program:

\[
g = \% \_ \_ \_ \_ \_ \rightarrow 0.
f = \% \ (a:as) \ (b:bs) \_ \rightarrow g \ a \ b + f \ bs \ as
\]

This yields the more correct type of \([a] \rightarrow [b] \rightarrow \text{Int}\). By not treating the call to \( f \) as a definition for the type of \( f \), the resultant types can be more liberal.
8.5 Final substitution

After unification, we are left with a list of type equations $E$ of the form $A = T$, where $A$ is necessarily a type variable, and a type variable $Z$ representing the final type of the expression. We have the guarantee that if $A = T \not\in E$ for any $T$, then no type information about $A$ is present in the expression.

We define a function

$$B(X) = \begin{cases} 
X & \text{if } X \text{ is some concrete type} \\
Y & \text{if } X = T \not\in E, \text{ where } Y \text{ is a fresh type variable} \\
S(B(X_1), \ldots, B(X_n)) & \text{if } X = S(X_1, \ldots, X_n) \text{ for some structure } S 
\end{cases}$$

The final type of the expression thus is $B(Z)$.
9 Pattern matching

The patterns given in section 5.2 have to be transformed to atomic patterns in order to be evaluated. Specifically, each case should have exactly one matching pattern for each constructor. Abstractions, guards, if-expressions and cases all must be transformed towards a single case construct, that allows exactly one constructor per branch and either zero or one bound variables per branch, so to project the argument of the constructor if it has one.

9.1 Guards and if-then-else statements

Guards are transformed to possibly deeply nested if-then-else statements. If-then-else statements are, in turn, transformed into case statements, with one branch for True and one for False.

9.2 Abstractions

Abstractions are transformed by introducing bound variables and creating cases. For example,

```
% Foo 3 => e1
| (Bar x) _ => e2
| y z => e3
```

is transformed to:

```
% a => (case a of
  Foo => (% 3 => e1)
  Bar x => (% _ => e2)
  y => (% z => e3))
```

Note that the inner abstractions have not been resolved yet. Transformations on abstractions and cases are mutually recursive.

9.3 Cases

Cases are transformed by building up a list of discriminants. Each time a discriminant distinguishes between constructors, an atomic case statement is built enumerating each constructor of the appropriate type.

Consider the following example:

```
case e of
  p1 => e1
  | p2 => e2
```

The algorithm needs a list of discriminants, which need to be resolved, and a list of pattern-list-expressions. The initial list of discriminants from this case is [e] and the list of pattern-list-expressions is [[(p1), e1], [(p2), e2]].

Should the algorithm ever run into a case with a non-empty list of discriminants, but an empty list of patterns, it is an error. This would occur if a case construct were built that did not cover all constructors.

9.3.1 Characteristics

A characteristic of a list of patterns describes whether the patterns discriminate on constructors, tuples, characters, integers or variables (none of the above), denoted respectively by one of \{C(T), T(n), H, N, V\}. The method for finding the characteristic is simple: search the list of patterns in order until the first non-variable pattern is found. If all patterns are variable patterns, then the characteristic is V. Because the program has typed correctly, there will never be patterns of the same list having different types — e.g., it will never be the case where one pattern matches a tuple and another matches a constructor — but variable patterns can be mixed with any other sort of pattern.
9.3.2 Building cases

If the characteristic of the first set of patterns $p_{1,1}, \cdots, p_{1,n}$ is $\mathbb{C}(T)$ for some type $T$, we build a case. Consider where our list of discriminants is $d : D$ and the constructors for $T$ are $A, B$ (i.e. $A$ takes no argument, $B$ takes an argument). We construct a case statement as:

\[
\begin{align*}
\text{case } d \text{ of } & \quad A \Rightarrow [r_A(p_{1,1}), \cdots, p_{m,1}], e_1 \\
& \quad \vdots \\
& \quad [r_A(p_{1,n}), \cdots, p_{m,n}], e_n \\
& \quad B z \Rightarrow [r_B(z)(p_{1,1}), \cdots, p_{m,1}], e_1 \\
& \quad \vdots \\
& \quad [r_B(z)(p_{1,n}), \cdots, p_{m,n}], e_n
\end{align*}
\]

The $r$ function reduces the pattern according to the constructor given. It works as follows:

- if the pattern conflicts with the constructor (e.g., a pattern matches $B$ will conflict with $r_A$, as the constructors are different), the entire pattern-list-expression is dropped, as it can never be matched;
- if the pattern matches and the constructor takes no argument (e.g., a pattern matches $A$ on $r_A$), then that pattern is dropped (because it's been matched) and pattern-matching continues on the remaining discriminants and pattern-list-expressions if any;
- if the pattern matches and the constructor takes an argument, the pattern is dropped, but the pattern’s argument is re-added to the front of the list, and $z$ (where $z$ is the fresh variable added to the built case) is added to the front of the list of discriminants;
- if the pattern is a variable, the pattern is dropped and expression $e$ is transformed to $(\lambda x.e)C$, where $C$ is the constructor being matched against and $x$ is the variable of the pattern.

9.3.3 Destroying tuples

If the characteristic of the first set of patterns is $\mathbb{T}(n)$ for some non-negative integer $n$, i.e., we’re casing on a tuple, we destroy the tuple into its component elements. Simply, where $d$ is the discriminant, we remove $d$ and add $\pi_0 d, \cdots, \pi_{n-1} d$ to the front of the list of discriminants. Similarly, we destroy the patterns so that any tuple pattern $(p_1, \cdots, p_n)$ becomes the list of patterns $p_1, \cdots, p_n$.

If the pattern is a variable, we destroy it to the $n$-set of underscores. In other words, the pattern $x$ is interpreted as matching the tuple $(_{\cdots}, _{\cdots})$, which is then destroyed according to usual tuple-destruction rules. Further, because the variable $x$ has been removed from any patterns, the expression $e$ is transformed to be $(\lambda x.e)d$. An abstraction is built in lieu of performing a substitution to avoid the performance degradation in case $x$ is not linear in $e$.

9.3.4 Character and integer rules

Where the characteristic is $\mathbb{H}$ or $\mathbb{N}$, we build if-else-then constructs (which are in turn eventually transformed into case statements), testing for equality between the first discriminant and the patterns. The exception is for the first variable pattern, which will take the place of the “else”.

If there are no variable “catch-all” patterns, it is an error.

9.3.5 Shifting

If the characteristic is $\mathbb{V}$, there is no way to discriminate on this list of patterns. Thus, for each variable pattern, we apply the substitution $[d/x]$. For an underscore pattern, no substitutions need to be made, and it’s sufficient to simply shift.
Figure 4: Transforming patterns.
10 Reduction

10.1 Data structures

When reduction, or evaluation, is to take place, the program is represented in the following data structures:

| Var = Reference Expression. |
| Constructor = Integer. |
| Expression = Abstraction of Var, Expression |
| Case of [Constructor, Option Var, Expression] |
| Application of Expression, Expression |
| Construction of Constructor, Option Expression |
| Tuple of [Expression] |
| Variable of Var |
| Projection of Integer, Integer (hereafter \( \pi_{i,h} \)) |
| TupleConstruction of Integer (hereafter \( \rho_{a} \)) |
| Builtin of Integer, BuiltinFunction |
| Undefined of Integer. |

Program = Expression.

10.2 Automaton

A four-tuple automaton is used to reduce the program. The four columns are:

1. Current state, of type Expression;
2. Environment, of type [Expression];
3. Substitutions, of type Set \( (a \rightarrow b) \) (references to expressions);

The transition table for the automaton is seen in table 1. The initial state is \( (e, [], \emptyset, []) \), where \( e \) is the final expression of the program. The machine halts when there is no rule to apply, as defined in table 1.

<table>
<thead>
<tr>
<th>Initial state</th>
<th>Resultant state</th>
</tr>
</thead>
<tbody>
<tr>
<td>((xy, v; s, n))</td>
<td>((x, y; v; s, n))</td>
</tr>
<tr>
<td>((\lambda x, y; v; s, n))</td>
<td>((e, v, {x \rightarrow y} \cup s, n))</td>
</tr>
<tr>
<td>((\text{case } C_i \rightarrow \lambda x. e, \text{ Cons } (i, z); v; s, n))</td>
<td>((e, v, {x \rightarrow z} \cup s, n))</td>
</tr>
<tr>
<td>((\text{case } C_i \rightarrow e, \text{ Cons } i; v; s, n))</td>
<td>((e, v; s, n))</td>
</tr>
<tr>
<td>((x, v; s, y; n))</td>
<td>((e, v, s, x; n))</td>
</tr>
<tr>
<td>((e, v; s, y; n))</td>
<td>((e, v, {y \rightarrow e} \cup s, n))</td>
</tr>
<tr>
<td>((\pi_{i,1}, \ldots, \pi_{i,n}; v; s, n))</td>
<td>((e_{i,1}, v, s, n))</td>
</tr>
<tr>
<td>((\rho_{m}, e_1; \ldots; e_n; v; s, n))</td>
<td>((e_{1,1}, \ldots, e_{n,1}; v, s, n))</td>
</tr>
<tr>
<td>((\text{Builtin } (n, f), e_1; \ldots; e_n; v; s, n))</td>
<td>((f(e_1, \ldots, e_n), v, s, n))</td>
</tr>
</tbody>
</table>

Table 1: State transitions

10.3 Environments

The given automaton has the problem in that abstractions and case branches with bound variables have a single reference to the bound variable. When multiple separate instances of a bound variable need be in memory at one time, as is the case generally in recursion, one reference is unacceptable.

Thus, in the current implementation, each bound variable is “instantiated” before being bound. The associated expression is transformed to reference a new reference.
Part III
Appendices

A  Standard prelude

/*
 * list functions
 */
fold :: (a -> b -> b) -> b -> [a] -> b =
  % _ n [] => n
| c n (x:xs) => c x (fold c n xs).

map :: (a -> b) -> [a] -> [b] =
  % f => fold (% x y => f x:y) □.

(++) :: ([a], [a]) -> [a] =
  % (□, y) => y
| (x:xs, y) => x:(xs ++ y).

/*
 * boolean functions
 */
not :: Bool -> Bool =
  % True => False
| False => True.

(&&) :: (Bool, Bool) -> Bool =
  % (True, True) => True
| _ => False.

(||) :: (Bool, Bool) -> Bool =
  % (False, False) => False
| _ => True.

References

