Transformation of Imperative Statements into Functional Expressions

Tim Sheard and Leonidas Fegaras
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 89-112
October 25, 1989

Transformation of Imperative Statements into Functional Expressions

Tim Sheard and Leonidas Fegaras

Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

October 25, 1989

1 Introduction

During the last few years there has been a strong interest in high level functional programming languages, such as Hope, Miranda, and ML. In contrast to the imperative programming languages where side effects are supported as the primary method of changing the state of a task, these languages force referential transparency by using the lambda abstraction construct as the only way to introduce variable bindings. These functional languages are built on top of the lambda calculus, but they also provide some advanced features, like polymorphic type checking and pattern matching, found in modern high level programming languages.

The lambda calculus is as expressive as conventional imperative languages and has the advantage of a simple, uniform syntax and semantics. Simplicity and uniformity are highly desirable because they reduce the chances of errors during programming and often make the code easy to understand and debug. These features also facilitate the proofs about programs. Functional languages are also suitable for concurrent systems, because parallel graph reduction methods are very simple and efficient. There are some cases though, where imperative programming seems more appropriate and natural. For example, it is difficult to express in functional notation a sequence of actions that must be performed in a strict order. To specify such a sequence of actions a complicated expression with lambda abstractions must be used, that may hide the programmers intentions and produce unclear code.

In this paper we will present a macro-like language with features similar to imperative languages, that can be translated to lambda calculus. By providing this alternative way of programming, we encourage traditionally trained people to work in a functional environment, gaining all the benefits of using such a model, without the need to sacrifice the convenience of the imperative style of programming.

Although the functional language ADABTPL will be used to give some examples throughout this paper, and Miranda to describe the program transformations, these methods can be applied to any functional programming language. ADABTPL is a strongly typed database specification language that uses type inference algorithms to deduce type checking information.

2 Overview

In this paper we will provide some macro-like capabilities that will improve the usage of functional languages. Two features will be introduced.

1. The first is an alternative method for explicit function calls that is self-documented and can reduce the number of lambda abstractions. This feature is called a **keyword expression**. An example of a keyword expression is

   ```
   ( ALL x IN s WHERE x>0 )
   ```

   which is translated to the function call
   $select(s,\ function(x : number) \rightarrow boolean\ (x > 0))$.

2. The second feature emulates the imperative style of programming, providing that all side effects are upon local variables defined in the body of the enclosing function. Each such function (which is called an **imperative function**) is translated into a functional expression. This can be done by gathering all variables defined in this function, as well as a place holder to keep the result of the function, into a special vector called the **state vector**. In this way the emulation problem is reduced to the problem of finding a lambda abstraction from the state vector to the state vector such that when it is applied to the actual parameters of the imperative function it produces a new state vector describing the state of the function after the application of its body to its arguments. To get the resulting value of this function, we need to return the value of the place holder from the new state vector.

Only three primitive constructs are provided for imperative functions: **assignment statements**, **begin-end statements**, and **keyword statements**. Keyword statements are abbreviations of lambda abstractions from a state vector to a state vector and serve a similar purpose to that of the keyword expressions.

In the database paradigm, transactions are special types of functions that take the current database as a parameter, as well as some optional parameter values, and produce a new database. Transactions are like imperative functions, in which the state vector includes the database. This similarity to transfer transactions into functional expressions is also exploited.

3 Alternative Calling Sequences

ADABTPL provides four alternative ways of making function calls: **normal prefix, unary prefix, infix,** and **keyword expressions**. The following examples illustrate how one can define functions in ADABTPL and call them using different forms of calling sequences.

```
function not (x : boolean) : boolean ;
prefix sequence ~ x ; prec 450 ;
body
  if x then false else true
end ;
```

In this example function *not* is defined to be from *boolean* to *boolean*. One example of a normal prefix calling sequence (where the list of arguments follow the function name) is *not*(*true*). The second line indicates that besides the normal prefix calling sequence, a unary prefix operation form may be used. This line is optional and may only be used for unary function definitions. The name of the operator in this example is defined to be $\sim$. In general, unary prefix operators (as well as binary infix operators) can be any sequence of special symbols or an identifier enclosed in vertical bars (like |*not*|). One example of the use of this prefix operator is $\sim x$, which is translated to *not*(x). The **prec** declaration part denotes that the prefix operator will have precedence 450. Precedence values run from a low of zero to a high of 500.

```
function(alpha)
     set_member ( elem : alpha ; s : set(alpha) ) : boolean ;
infix sequence elem |in| s ;
prec 400 ; associativity non ;
body ... end ;
```

This is an example of the definition of a binary calling sequence for a function that describes the membership predicate over sets. *alpha* is a type variable that can be bound to any type. *set_member* is a function from *alpha* $\times$ *set*(*alpha*) to *boolean*, for any type *alpha*. For example, the type variable *alpha* in expression *set_member*(3, *insert*(3, *emptyset*)) is bound to *number*. The third line describes the syntax of the alternative infix calling sequence. This form can only be used for functions that have two parameters. Here |*in*| is the infix operator. Using this notation, the last function call may be written as 3 |*in*| *insert*(3, *emptyset*).

The fourth line defines the precedence and associativity of the infix operator. The associativity of an operator can be **left**, **right**, or **non**. For example, the '$-$' operator is normally defined with left associativity, because we want to interpret a sequence like $a - b - c$ as $(a - b) - c$ and not as $a - (b - c)$ (which is how it would be interpreted if it had right associativity). If no associativity is wanted then the associativity *non* is used (for example, $x < y < z$ is an error since the operation '$<$' does not associate).

```
function(alpha)
    exists ( s : set(alpha) ;
             p : function(alpha)->boolean ) : boolean ;
keyword sequence EXISTS x@local IN s@value WHERE p@function(x) ;
body ... end ;
```

Function *exists* takes a set s as a first argument, and a predicate (a function to boolean) p as a second argument. *exists* returns true if there is an element x in s such that $p(x)$ is true. An example of a call of such function is: *exists*(*sn*, #*oddp*), where *sn* is of type *set*(*number*), and *oddp* is defined as a function from *number* to *boolean* and returns true if a number is odd. The notation #*oddp* is used to emphasize that *oddp* is a globally defined function (in contrast with the parameter p of *exists* which is a locally defined function). Another example of a call is: *exists*(*sn*, *function*(z : *number*) $\rightarrow$ *boolean* $(z > 5)$). Here we use a lambda abstraction to form a predicate that has not been defined globally. This function is from *number* to *boolean* and returns true if its argument is greater than 5. The keyword sequence line in this function definition describes an alternative simpler way of calling this function. Reserved names with all uppercase characters are keyword names and other names are regular identifiers. Using this notation the last call can be written as:

```
( EXISTS z IN sn WHERE z>5 )
```

If we compare this with the definition for the keyword sequence attached to the function *exists*, we can see that a local variable x is introduced which is bound to z, s is a parameter of *exists* bound to *sn*, and p is bound to *function*(z : *number*) $\rightarrow$ *boolean* $(z > 5)$. The last binding is inferred by the compiler from the fact that p is from *number* to *boolean* and it is defined as *function*(x) (a function that uses the local variable x). The specifiers **local**, **value**, and **function** in the keyword sequence definition indicate the role of each variable in the pattern. x@*local* means that x must be bound to a variable name; s@*value* means that s can be bound to any expression; and p@*function*(x) means that p can be matched with any expression and that it will be bound to the lambda abstraction with only one parameter x and the matched expression for its body. The call *exists*(*sn*, #*oddp*) can be written as

```
( EXISTS z IN sn WHERE oddp(z))
```

4 Imperative Functions

The second feature that ADABTPL provides is imperative function definitions. For example consider a simple imperative function that makes some arithmetic computations.

```
imperative function test ( a : number ) : number ;
begin test:=a+10;
      test:=test*5
end;
```

Here the name *test* serves two purposes: as the function name and as a variable that will hold the result. The above imperative function will be translated into the following regular function:

```
function test ( a : number ) : number ;
body (a+10)*5 end ;
```

A more interesting example comes from the function *union*, that returns the union of two sets. It can be written as follows:

```
imperative function(alpha)
    union ( a, b : set(alpha) ) : set(alpha) ;
initialize union := a ;
begin
  [ FOREACH x IN b DO [ INSERT x INTO union ] ]
end ;
```

The name *union* serves also as the name of the function and as a variable that will hold the returned value. This variable is initialized to be equal to *a*. In the body of the function *union* every element *x* from *b* is inserted into the *union*. Keyword statements have a similar form with the keyword expressions. The difference is that keyword statements are enclosed between square brackets and can have other statements as components. We will see later how one can define such statement patterns. In the *union* definition the last part of the FOREACH statement is another statement. This statement, which inserts an element *x* into *union*, is repeated for every element *x* in *b*. Before trying to describe INSERT and FOREACH, it is useful to see the translation of the *union* into a regular function:

```
function(alpha) union ( a, b : set(alpha) ) : set(alpha) ;
body foreach(b, function($:STATE1(alpha) with [x:alpha])->STATE1(alpha)
                    ( [$.a, $.b, insert($.x, $.union)] ),
             [a, b, a] ).union end ;
```

where STATE1 is the state vector of *union*:

```
 STATE1(alpha) = [ a, b, union: set(alpha) ] ;
```

The state vector includes all the parameters of *union*, as well a place to keep the result to be returned. The *with* type constructor extends the state vector with a value x that will hold the local variable introduced by FOREACH. The function *foreach*, appeared in the translation of *union*, iterates over all elements of the set *dest* of type *set(alpha)*, and use a function *oper* to change the initial value of state x of type *beta*. For example, the call *foreach*($\{1,2,3\}$, *oper*, x) is equivalent with the expression *oper*(*oper*(*oper*(x *with* [1]) *with* [2]) *with* [3]).

```
function(alpha, beta)
  foreach ( dest : set(alpha) ;
            oper : function(beta with [ext:alpha])->beta;
            x : beta ) : beta ;
body
  case dest of
    emptyset : x
    others   : foreach(rest(dest), oper, oper(x with [choose(dest)]))
  end
end ;
```

For example, suppose that the function *union* is called with $a = \{1,2\}$ and $b = \{3,4\}$. Then, the function *foreach* is called with

```
oper = function($:STATE1(alpha) with [x:alpha])->STATE1(alpha)
          ( [$.a, $.b, insert($.x, $.union)] )
```

which remains the same during the execution. Here is a trace of the parameters *dest* and x in *foreach*:

```
1) dest = {2,4}
   x    = [ {1,2}, {3,4}, {1,2} ]
2) dest = {4}
   x    = oper( [ {1,2}, {3,4}, {1,2} ] with [3] )
        = [ {1,2}, {3,4}, insert(3, {1,2}) ]
        = [ {1,2}, {3,4}, {3,1,2} ]
3) dest = {}
   x    = oper( [ {1,2}, {3,4}, {3,1,2} ] with [4] )
        = [ {1,2}, {3,4}, insert(4, {3,1,2}) ]
        = [ {1,2}, {3,4}, {4,3,1,2} ]
--> [ {1,2}, {3,4}, {4,3,1,2} ]
```

Now we present the imperative statement definition for INSERT.

```
imperative stmt(alpha)
     INSERT a:alpha@value INTO s:set(alpha)@component
      === update $ by [ s := insert(a,s) ] ;
```

In imperative statement definitions we have a pattern part and a body part (at the right of symbol '==='). The pattern part describes the form of the keyword statements and it is similar to the keyword sequence definition. In this case, we need to denote the type of each variable name in the pattern, because imperative statement definitions are not defined in a context of a regular function definition. Also the possible roles of variables (from value, local, and function) are extended to include **stmt** and **component** roles. The INSERT imperative statement definition has the form: *INSERT a INTO s*, where *a* is an expression and *s* is a component of the state tuple (usually the result variable). When such a statement is found, the variables *a* and *s* are bound and are used to translate the form into its body. In the body of INSERT, $ means the value of the state vector at this point. The *update* statement is an operation that returns a new state vector from the old, after replacing the component *s* by the expression *insert*(*a*, *s*). That is, this type of updates is defined from the following axiom:

$$(\textit{update}\ \$\ \textit{by}\ [s := E]).r \;=\; \begin{cases} E & r = s \\ \$.r & r \neq s \end{cases}$$

The imperative statement definition for FOREACH has the following form:

```
imperative stmt(alpha)
     FOREACH x:alpha@local IN s:set(alpha)@component
             DO f:function(state with [ext:alpha])->state@stmt(x)
     === foreach(s,f,$) ;
```

This imperative statement definition introduces a new local variable *x* that is used to hold an element from the set *s*, when there is an iteration over *s*. The DO part of FOREACH is another statement. Therefore, it is a function from state to state (the name *state* is the type of the state vector $ at this point). The input state of *f* is extended to include *x*, because this statement may use the value of *x*. The body of FOREACH is a function from state to state that calls *foreach*(*s*, *f*, $).

A more complex example is the *intersection* function that returns the set of all common elements in *a* and *b*. This function uses the IF-THEN imperative statement.

```
imperative function(alpha)
    intersection ( a, b : set(alpha) ) : set(alpha) ;
initialize intersection := emptyset ;
```

```
begin
  [ FOREACH x IN a DO
      [ IF x |in| b THEN [ INSERT x INTO intersection ] ] ]
end ;
```

where the IF-THEN imperative statement is defined as:

```
imperative stmt
     IF cond:boolean@value THEN thens:function(state)->state@stmt
     === if cond then ~thens($) else $ ;
```

intersection is translated into the following function:

```
STATE2(alpha) = [ a, b, intersection : set(alpha) ] ;

function(alpha) intersection ( a, b : set(alpha) ) : set(alpha) ;
body foreach(a, function($:STATE2(alpha) with [x:alpha])->STATE2(alpha)
                    ( if $.x |in| $.b
                        then [$.a, $.b, insert($.x, $.intersection)]
                            with [$.x]
                        else $ ),
             [a, b, emptyset] ).intersection
end ;
```

REMOVE and UPDATE are examples that use imperative statement definitions with functional parameters.

```
imperative function testit ( a : set(number) ) : set(number) ;
initialize testit := a ;
begin [ REMOVE x FROM testit WHERE x<0 ] ;
      [ UPDATE x IN testit WHERE x<8 BY x+1 ]
end ;
```

where REMOVE and UPDATE are defined as:

```
imperative stmt(alpha)
     REMOVE x:alpha@local FROM s:set(alpha)@component
            WHERE test:function(alpha)->boolean@function(x)
     === update $ by [ s := remove(s, test) ] ;

imperative stmt(alpha)
     UPDATE x:alpha@local IN s:set(alpha)@component
            WHERE test:function(alpha)->boolean@function(x)
            BY oper:function(alpha)->alpha@function(x)
     === update $ by [ s := update(s, oper, test) ] ;
```

testit is translated into the following function:

```
function testit ( a : set(number) ) : set(number) ;
body update( remove( a, function(x:number)->boolean ( x<0 ) ),
             function(x:number)->number ( x+1 ),
             function(x:number)->boolean ( x<8 ) )
end ;
```

5 Transactions

Transactions use the database definition as an implicit defined parameter. One example of a transaction is the following:

```
database DBtype : [ a, b : set(number) ] ;

transaction test ( z : number) ;
begin
   [ INSERT z INTO b ] ;
   [ FOREACH x IN a DO [ INSERT x INTO b ] ]
end ;
```

The database type *DBtype* defines a collection of all objects that one want to be persistent. This collection is the only persistent variable that can be defined in a program (a database definition must be appeared only once in a program). Transaction *test* is used to perform some side effects to the database object. Every component of *DBtype* can be accessed and/or changed inside the body of a transaction, by using its name. Transaction *test* can be translated into the following imperative function:

```
imperative function test ( DB : DBtype; z : number) : DBtype ;
initialize test := DB ;
begin
   [ INSERT z INTO test.b ] ;
   [ FOREACH x IN test.a DO [ INSERT x INTO test.b ] ]
end ;
```

6 Translation and Role Playing in Keyword Expressions

The translation of keyword expressions to lambda calculus is straightforward. When a keyword expression is found, it is unified with the definition of a keyword sequence.

The selected candidate definition is the one whose list of keywords is the same as the keys in the expression. To test whether there is a perfect match one may use the function call $MATCH\ K\ P\ L$. In this call, K is the keyword expression which is a list of keys and expressions; P is the pattern which is the keyword sequence of the candidate definition that forms a list of keys and variable specifications; and L is a list of bindings of all variables in the keyword sequence definitions found so far. $MATCH$ returns a new list L, which is the list of bindings of all variables in P bound to the expressions in K. If the matching fails then $MATCH$ returns the value *fail*. $MATCH$ will be described using a notation similar to that used in Miranda language. In that notation, if the parameters of a function call matches with the pattern in the left side of a definition (which is written as an equation), then all variables that appeared in this pattern are bound and used to execute the right part of the equation.

$$MATCH\ []\ []\ L = L \quad (1)$$
$$MATCH\ (key : k)\ (key : p)\ L = (MATCH\ k\ p\ L),\ Keyp(key) \quad (2)$$
$$MATCH\ (e : k)\ (\{x@value\} : p)\ L = MATCH\ k\ p\ L[x = e] \quad (3)$$
$$MATCH\ (v : k)\ (\{x@local\} : p)\ L = (MATCH\ k\ p\ L[x = v]),\ Varp(v) \quad (4)$$
$$MATCH\ (e : k)\ (\{x@function(v_1, ...)\} : p)\ L$$
$$= MATCH\ k\ p\ L[x = (\lambda v_1.\lambda v_2 ...\ e)/L] \quad (5)$$
$$MATCH\ x\ y\ L = fail \quad (6)$$

In this set of rules, as well as in the following rules, we will use squiggly brackets to inclose pieces of ADABTPL programs. Everything else is in Miranda form. The notation $L[x = e]$ means that the binding $[x = e]$ is added to L. The ':' operator is the infix list constructor, while the '/' operator is the substitution operator (for example, e/L uses the binding list L to change the variables in e).

One can see that $MATCH$ is a simple tail-recursive pattern matcher that checks whether an element in the keyword expression and the corresponding element in the pattern matches. The first rule describes the trivial case where we finish comparing the elements of the two lists. In that case the list of bindings that has been accumulated up to here as the parameter L is returned back. The second rule is redundant because it tests if a keyword in the expression matches with the associated keyword in pattern. This rule is fired whenever an identifier *key* satisfies the *Keyp* predicate, that is whenever it is a keyword. Rules three, four, and five use the roles of variables in the pattern to extract meaningful bindings. More specifically, the third rule indicates that variable x has been assigned the role *value*, which means that it expects any expression e. In that case the binding $[x = e]$ is added to L. The fourth rule indicates that x plays a role of a local variable. In that case v must be a variable name (that satisfies the predicate *Varp*) which must not be defined previously as a local variable in the same keyword expression. The fifth rule make explicit some implicit lambda abstractions. Here x must be defined as a function with some of the local variables defined previously as parameters. x is not bound to e but to the lambda abstraction that has body e. The last rule catches every call that does not fall into any of the previous categories, and returns *fail*.

Suppose now that the candidate keyword sequence for the keyword expression K is defined in the context of the function:

$$function\ N\ (v_1 : T_1;\ ...\ v_n : T_n)\ :\ T$$

then K is translated into $N(v_1,\ ...\ v_n)/(MATCH\ K\ P\ [])$, where P is the keyword sequence definition of function N.

For example, the keyword expression (`EXISTS z IN sn WHERE z>5`) has the same keys with the keyword sequence in function *exists*. Therefore, *exists* is used as the candidate function. In that case, a trace of the $MATCH$ function is:

```
MATCH [EXISTS,z,IN,sn,WHERE,z>5]
      [EXISTS,x@local,IN,s@value,WHERE,p@function(x)] []                  (2)
MATCH [z,IN,sn,WHERE,z>5] [x@local,IN,s@value,WHERE,p@function(x)] []     (4)
MATCH [IN,sn,WHERE,z>5] [IN,s@value,WHERE,p@function(x)] [x=z]            (2)
MATCH [sn,WHERE,z>5] [s@value,WHERE,p@function(x)] [x=z]                  (3)
MATCH [WHERE,z>5] [WHERE,p@function(x)] [s=sn,x=z]                        (2)
MATCH [z>5] [p@function(x)] [s=sn,x=z]                                    (5)
MATCH [] [] [p=l z.(z>5),s=sn,x=z]                                        (1)
-> [p=l z.(z>5),s=sn,x=z]
```

Therefore the keyword expression is translated to:

$$exists(s,\ p)/[p = \lambda z.(z > 5),\ s = sn,\ x = z]$$

which is $exists(sn,\ \lambda z.(z > 5))$.

7 Translation of Imperative Functions

In order to translate imperative functions into regular functions one has to define program transformation rules that will translate any statement into a function (lambda abstraction) from state to state. This is done with the function $TS\ S\ L\ T$. Here S is the statement to be translated; L is the list of bindings of variables in keyword statements that have been defined before S; and T is the state vector (an ADABTPL type). In order to make the produced lambda expressions more readable, the lambda calculus is extended to include tuples. The form $< e_1,\ e_2, ... >$ is used to denote tuple construction. It is translated into the lambda expression: $\lambda x.(if\ x\ =\ 1\ then\ e_1\ else\ ...\ else\ if\ x\ =\ n\ then\ e_n\ else\ (error))$. Using this notation, projections like $x.a_i$ become $(x\ i)$. The function call $(setval\ A\ I\ E)$ takes an array A, an index I, and an expression E and returns a new array that has the same elements with A, except the I_{th} element which is set to E.

FindPattern and *FindBody* return the pattern and the body of the imperative statement definition that matches the keyword statement. In order to avoid name conflicts, a copy of the imperative statement definition with new parameter names is always returned. The *MATCH* function that compares the keyword statement and the pattern is very similar to that used for the keyword expressions.

$$TS\ \{begin\ S\ end\}\ L\ T = TS\ S\ L\ T \quad (1)$$
$$TS\ \{begin\ S_1;\ \dots\ S_n\ end\}\ L\ T$$
$$= \lambda x.\ (TS\ \{begin\ S_2;\ \dots\ S_n\ end\}\ L\ T)\ (TS\ S_1\ L\ T)\ x \quad (2)$$
$$TS\ \{D\ :=\ D\}\ L\ T = \lambda x.x \quad (3)$$
$$TS\ \{a_i\ :=\ E\}\ L\ \{[a_1 : T_1;\ \dots\ a_i : T_i;\ \dots a_n : T_n]\}$$
$$= \lambda x. < (x\ 1),\ \dots\ E/[a_1 = (x\ 1), \dots, a_n = (x\ n)],\ \dots (x\ n) > \quad (4)$$
$$TS\ \{D.a_i\ :=\ E\}\ L\ \{[a_1 : T_1;\ \dots\ a_i : T_i;\ \dots]\}$$
$$= \lambda x. < (x\ 1),\ \dots\ (TS\ \{D\ :=\ E\}\ L\ T_i)\ (x\ i),\ \dots > \quad (5)$$
$$TS\ \{D[I]\ :=\ E\}\ L\ \{array\ [T_1]\ of\ T_2\} = TS\ \{D\ :=\ (setval\ D\ I\ E)\}\ L\ T_2 \quad (6)$$
$$TS\ \{[K]\}\ L\ T = (BODY\ (FindBody\ (Name\ K))\ T$$
$$(MATCH\ K\ (FindPattern\ (Name\ K))\ L\ T)) \quad (7)$$
$$TS\ S\ L\ T = error \quad (8)$$

For example the call $TS\ \{begin\ n := 3;\ m[10] := n + 1\ end\}\ []\ \{[n : number;\ m : array[1..100]\ of\ number]\}$ produces the lambda expression:
$\lambda x.\lambda x. < (x\ 1),\ (setval\ (x\ 2)\ 10\ ((x\ 1) + 1)) > \ \lambda x. < 3,\ (x\ 2) > \ x.$

The first two rule of *TS* describe how compound statements (formed by the begin-end construct) are translated into lambda abstractions. The first rule is the base case, where we have a compound statement with only one component *S*. In the second rule there are more than one statements. In that case we return the function composition of the translation of the compound statement without the first statement, with the translation of the first statement. The rules three through six are used to compile assignments of the form *dest* := *source* where *dest* is a component of the state vector while *source* is any expression. The third rule returns the identity lambda expression whenever the *source* and the *dest* are the same. This will cause the state vector to remain the same. The fourth rule examines the case where the destination is a simple component of the state tuple (the i_{th} component). In that case a lambda expression is constructed that returns a new state vector. This vector must have the same elements with the old state vector, except of the i_{th} element which must be set to the value of source. Note that every component name that appears in the source must be substituted with its real value, that is with the proper projection of the state vector. The fifth rule describes a projection for a destination. Here the projection name a_i must be a component of the state vector. In that case we take T_i, the type of a_i, as the new state vector and try to find a lambda expression for $D := E$. Again a new state vector similar to the old need to be returned, except for the i_{th} element which must be the application of the lambda abstraction for $D := E$ to the old i_{th} element of the state. The sixth rule describes the case of an array reference in the destination. Here the assignment is changed in that way that the destination will become the whole array D instead of an array reference, and the source will return a new array that has the same elements with D except for the i_{th} element which is set to E. The seventh rule is the most interesting one. This rule translates a keyword statement K into a lambda abstraction. $(NAME\ K)$ returns the list of keywords in the body of K. These keywords will help to identify the right imperative statement definition to match this expression. Finally, the last rule catches all errors.

The $MATCH$ function is very similar to the matcher for keyword expressions. In this case though, the state vector T must be passed as a parameter.

$$
\begin{array}{lr}
MATCH\ []\ []\ L\ T = L & (1) \\
MATCH\ (key : k)\ (key : p)\ L\ T = (MATCH\ k\ p\ L\ T), \qquad Keyp(key) & (2) \\
MATCH\ (e : k)\ (\{x : T_0@value\} : p)\ L\ T = MATCH\ k\ p\ L[x = e]\ T & (3) \\
MATCH\ (v : k)\ (\{x : T_0@local\} : p)\ L\ T & \\
\quad = (MATCH\ k\ p\ L[x = v]\ T), \qquad Varp(v) & (4) \\
MATCH\ (v : k)\ (\{x : T_0@component\} : p)\ L\ T & \\
\quad = (MATCH\ k\ p\ L[x = v]\ T), \qquad Componentp(v,\ T) & (5) \\
MATCH\ (e : k)\ (\{x : function(v_1 : T_1; ...) \rightarrow T_0@function(v_1, ...)\} : p)\ L\ T & \\
\quad = MATCH\ k\ p\ L[x = (\lambda v_1.\lambda v_2 ...\ e)/L]\ T & (6) \\
MATCH\ (s : k)\ (\{x : function(state\ with\ [n_1 : T_1; ...]) & \\
\qquad\qquad \rightarrow state@stmt(v_1, ...)\} : p)\ L\ T & \\
\quad = MATCH\ k\ p\ L[x = (TS\ s\ L\ \{T\ with\ [v_1 : T_1; ...]/L\})]\ T & (7) \\
MATCH\ x\ y\ L\ T = fail & (8)
\end{array}
$$

One can see that there are two new roles for variables. Rule five says that variable x must be a component of the state vector. This rule is similar to the fourth rule, but here we have to check if v is actually a component of the state. The fifth rule describes the case where there is another statement as a component of a keyword statement (one example is nested keyword statements). In that case s must be translated to a lambda expression that will construct a new state vector. Therefore, function TS must be used again to translate s. Note that the role of x is $stmt(v_1, ..)$, where v_i is a local variable defined before. This role is similar to the function role, with the only difference that now the implicit lambda abstraction is from state to state. The old state vector need to be expanded to include places for the values of the local variables. This is done with the **with** construct that form the new T in the call of TS.

$BODY$ is the translation of the right part (after the symbol ===) of the imperative statement definition. It has the rules:

$$\begin{aligned}
&BODY\ \{update\ \$\ by\ [a\ :=\ E]\}\ \{[a_1 : T_1;\ ...\ a_i : T_i;\ ...]\}\ L\\
&\quad = \lambda x.\ <(x\ 1),\ ...\ (E/L[\$ = x])/[a_1 = (x\ 1), ...],\ ...>,\ \text{ where } a_i = a/L \quad (1)\\
&BODY\ \{update\ \$\ by\ [D.a\ :=\ E]\}\ \{[a_1 : T_1;\ ...\ a_i : T_i;\ ...]\}\ L\\
&\quad = \lambda x.\ <(x\ 1),\ ...\ (BODY\ \{update\ \$\ by\ [D\ :=\ E]\}\ T_i\ L)\ (x\ i),\ ...>,\\
&\qquad\qquad \text{where } a_i = a/L \quad (2)\\
&BODY\ \{update\ \$\ by\ [D[I]\ :=\ E]\}\ \{array\ [T_1]\ of\ T_2\}\ L\\
&\quad = BODY\ \{update\ \$\ by\ [D\ :=\ (setval\ D\ I\ E)]\}\ T_2\ L \quad (3)\\
&BODY\ E\ \{[a_1 : T_1;\ ...\ a_i : T_i;\ ...]\}\ L\\
&\quad = \lambda x.(E/L[\$ = x])/[a_1 = (x\ 1), ...] \quad (4)
\end{aligned}$$

The first three rules in the *BODY* are very similar to the rules four through six in *TS*, because we need to change a component of the state vector again. The fourth rule describes the case where the body of an imperative statement definition is any expression other than *update*. In that case the lambda abstraction that has body E is returned.

The general form of an imperative function is the following:

$$\begin{aligned}
&imperative\ function\ N\ (v_1 : T_1;\ ...\ v_n : T_n)\ :\ T\ ;\\
&var\ l_1 : LT_1 := e_1;\ ...\ var\ l_m : LT_m := e_m;\\
&initialize\ N := e;\\
&S;
\end{aligned}$$

Here the second line with the *var* definitions, introduces some local variables that are mainly used to hold temporary results. Each such variable l_i has type LT_i and it is initialized to value e_i. The state vector is $[v_1 : T_1;\ ...\ v_n : T_n;\ l_1 : LT_1;\ ...\ l_m : LT_m;\ N : T]$. The initial value of this vector is $< v_1,\ ...\ v_n,\ e_1,\ ...\ e_m,\ e >$. Therefore, the translation of S is:

$$\begin{aligned}
&SIMPL((TS\ S\ []\ \{[v_1 : T_1;\ ...\ v_n : T_n;\ l_1 : LT_1;\ ...\ l_m : LT_m;\ N : T]\})\\
&\qquad < v_1,\ ...\ v_n,\ e_1,\ ...\ e_m,\ e >)(m + n + 1)
\end{aligned}$$

where SIMPL is a simplification function that uses the following rules:

$SIMPL\ \lambda x.x\ e = SIMPL\ e$
$SIMPL\ \lambda x.e_1\ e_2 = e_1/[x = (SIMPL\ e_2)]$, when the right part is shorter than the left
$SIMPL\ < e_1,\ e_2,\ ...\ e_n >\ i = SIMPL\ e_i$
$SIMPL\ (if\ c\ then\ e_1\ else\ e_2)\ i = SIMPL\ (if\ c\ then\ (e_1\ i)\ else\ (e_2\ i))$
$SIMPL\ \lambda x.e = \lambda x.\ SIMPL\ e$
$SIMPL\ e_1\ e_2 = (SIMPL\ e_1)\ (SIMPL\ e_2)$
$SIMPL\ < e_1,\ ...\ e_n > = < (SIMPL\ e_1),\ ...\ (SIMPL\ e_n) >$
$SIMPL\ if\ c\ then\ e_1\ else\ e_2 = if\ (SIMPL\ c)\ then\ (SIMPL\ e_1)\ else\ (SIMPL\ e_2)$
$SIMPL\ e = e$

Here is an example of how one may use some of the above rules to translate the imperative function *union*:

```
1) TS {begin [FOREACH,x,IN,b,DO,[INSERT,x,INTO,union]] end} [] [a,b,union:set(alpha)]
  2) TS {[FOREACH,x,IN,b,DO,[INSERT,x,INTO,union]]} [] [a,b,union:set(alpha)]
    3) MATCH [FOREACH,x,IN,b,DO,[INSERT,x,INTO,union]]
             [FOREACH,x1:alpha@local,IN,s1:set(alpha)@component,
                   DO,f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [] [a,b,union:set(alpha)]
      MATCH [x,IN,b,DO,[INSERT,x,INTO,union]]
             [x1:alpha@local,IN,s1:set(alpha)@component,
               DO,f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [] [a,b,union:set(alpha)]
      MATCH [IN,b,DO,[INSERT,x,INTO,union]]
             [IN,s1:set(alpha)@component,DO,
               f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [x1=x] [a,b,union:set(alpha)]
      MATCH [b,DO,[INSERT,x,INTO,union]]
             [s1:set(alpha)@component,DO,
               f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [x1=x] [a,b,union:set(alpha)]
      MATCH [DO,[INSERT,x,INTO,union]]
             [DO,f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [s1=b,x1=x] [a,b,union:set(alpha)]
      MATCH [[INSERT,x,INTO,union]]
             [f1:function(state with [ext1:alpha])->state@stmt(x1)]
             [s1=b,x1=x] [a,b,union:set(alpha)]
      4) TS {[INSERT,x,INTO,union]} [s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
        5) MATCH [INSERT,x,INTO,union]
                 [INSERT,a2:alpha@value,INTO,s2:set(alpha)@component]
                 [s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
          MATCH [x,INTO,union] [a2:alpha@value,INTO,s2:set(alpha)@component]
                 [s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
          MATCH [INTO,union] [INTO,s2:set(alpha)@component]
                 [a2=x,s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
          MATCH [union] [s2:set(alpha)@component]
                 [a2=x,s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
          MATCH [] [] [s2=union,a2=x,s1=b,x1=x] [a,b,union:set(alpha);x:alpha]
        5--> [s2=union,a2=x,s1=b,x1=x]
        5) BODY {update $ by [s2 := insert(a2,s2)]} [a,b,union:set(alpha);x:alpha]
                [s2=union,a2=x,s1=b,x1=x]
        5--> l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)>
      4--> l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)>
      MATCH [] [] [f1=l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)>,s1=b,x1=x]
             [a,b,union:set(alpha)]
    3--> [f1=l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)>,s1=b,x1=x]
    3) BODY foreach(s1,f1,$) [a,b,union:set(alpha)]
            [f1=l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)>,s1=b,x1=x]
    3--> l x. (foreach (x 2) l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)> x)
  2--> l x. (foreach (x 2) l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)> x)
1--> l x. (foreach (x 2) l x. <(x 1), (x 2), (insert (x 4) (x 3)), (x 4)> x)
```

Therefore, the statement in the body of *union* is translated into:

$\lambda x.(foreach\ (x\ 2)\ \lambda x. <(x\ 1),\ (x\ 2),\ (insert\ (x\ 4)\ (x\ 3)),\ (x\ 4)>\ x)$

The initial vector is $< a,\ b,\ a >$. So the body of the function *union* is:

$(\lambda x.(foreach\ (x\ 2)\ \lambda x. <(x\ 1),\ (x\ 2),\ (insert\ (x\ 4)\ (x\ 3)),\ (x\ 4)>\ x)\ < a,\ b,\ a >)\ 3$

which can be simplified into:

$(foreach\ b\ \lambda x. <(x\ 1),\ (x\ 2),\ (insert\ (x\ 4)\ (x\ 3)),\ (x\ 4)> < a,\ b,\ a >)\ 3$