Introduction to Compilers

Exam task set 2

Troels Thomsen - qvw203

17. Januar 2014

Department of Computer Science
University of Copenhagen

Indhold

1 Grammar transformation for LL(1) **3**
1.1 (a) . . . 3
1.2 (b) . . . 3
1.3 (c) . . . 4
1.4 (d) . . . 5

2 Extend the equality operation in Paladim to work on arrays. **6**
2.1 Type-checker changes. . . . 6
2.2 Code generation. . . . 8
2.3 Testing. . . . 11

3 A Flat-Reduce High-Order Function in Paladim. **12**
3.1 Type-checker changes. . . . 12
3.2 Code generation. . . . 13
3.3 Testing. . . . 15

4 A switch statement for Paladim. **16**
4.0.1 Lexer, grammar and abstract syntax tree. . . . 16
4.0.2 Type checking. . . . 18
4.0.3 Code generation. . . . 19
4.1 Testing. . . . 20

A Code changes **21**
A.1 Array equal implementation. . . . 21
A.1.1 Type.sml . . . 21
A.1.2 Compiler.sml . . . 22
A.2 Reduce function implementation . . . 25
A.2.1 Type.sml . . . 25
A.2.2 Compiler.sml . . . 26
A.3 Switch statement implementation. . . . 28
A.3.1 Lexer.lex . . . 28
A.3.2 Parser.grm . . . 28
A.3.3 AbSyn.sml . . . 28
A.3.4 TpAbSyn.sml . . . 29

A.3.5 Type.sml 29
A.3.6 Compiler.sml 30
A.3.7 TpInterpret.sml 32
A.4 Tests 33
A.4.1 Equal 33
A.4.2 Reduce 34
A.4.3 Switch 35

1 Grammar transformation for LL(1)

1.1 (a)

The following grammar G_{Tup}:

$$G_{Tup} : S \to id \tag{1}$$
$$S \to (\, T\,) \tag{2}$$
$$T \to S\,,\, T \tag{3}$$
$$T \to S \tag{4}$$

Is not compatible with a LL(1) parser, since both productions of T have not only overlapping, but identical First sets. $First(T) = \{id, (\}$ and $First(T) = \{id, (\}$. We can use the left-factorisation approach, to remove this problem and make the grammar compatible with LL(1).

We perform the left-factorisation by rewriting the overlapping productions into a single production, which contains the common prefix of the overlapping productions. We add a new nonterminal for the different suffixes, which do not overlap. This gives us G'$_{Tup}$.

$$G'_{Tup} : S \to id \tag{5}$$
$$S \to (\, T\,) \tag{6}$$
$$T \to S\, A \tag{7}$$
$$A \to ,\, T \tag{8}$$
$$A \to \tag{9}$$

G'$_{Tup}$ do not have a First/First conflict.

1.2 (b)

Our first sets can be found quite easily.
$First(S) = \{id, (\}$
$First(T) = \{id, (\}$
$First(A) = \{, \}$

1.3 (c)

When adding $S' \to S\,\$$ to our G'$_{Tup}$ we get the following:

$$G'_{Tup} : S' \to S\,\$ \tag{10}$$
$$S \to id \tag{11}$$
$$S \to (\,T\,) \tag{12}$$
$$T \to S\,A \tag{13}$$
$$A \to ,\,T \tag{14}$$
$$A \to \tag{15}$$

We can now calculate Follow sets for the non-terminals.

$S' \to S\,\$$	$\{\$\} \subseteq Follow(S')$	$\{\$\}$
$S \to id$	$\{,\} \cup Follow(T) \subseteq Follow(S)$	$\{id, (, ,\}$
$S \to (\,T\,)$	$\{,\} \cup Follow(T) \subseteq Follow(S)$	$\{id, (, ,\}$
$T \to S\,A$	$\{)\} \subseteq Follow(T)$	$\{)\}$
$A \to ,\,T$	$\emptyset \subseteq Follow(A)$	$\emptyset$
$A \to$	$\emptyset \subseteq Follow(A)$	$\emptyset$

We can conclude the follow sets as follows:

$Follow(S') = \{\$\}$
$Follow(S) = \{id, (, ,\}$
$Follow(T) = \{)\}$
$Follow(A) = \emptyset$

1.4 (d)

We can now calculate our look-ahead sets for all our productions of G'$_{Tup}$.

La($S \to id$)	$First(id) \subseteq La(S)$	$\{id\}$
La($S \to (\ T\ )$)	$First(() \cup First(T) \cup First()) \subseteq La(S)$	$\{(, id,)\}$
La($T \to S\ A$)	$First(S) \cup First(A) \subseteq La(T)$	$\{id, (\}$
La($A \to ,\ T$)	$First(,) \cup First(T) \subseteq La(A)$	$\{id, (, ,\}$
La($A \to$)	$\emptyset \subseteq La(A)$	$\{\emptyset\}$

We still have the problem that the grammar is ambiguous, and as such cannot be LL(1). I have tried to find a way to make it unambiguous, but i cannot seem to find a way, without altering the syntax which the grammar represents.

2 Extend the equality operation in Paladim to work on arrays.

Extending the equal operation to work with arrays, is a quite useful feature for potential Paladim programmers. It removes the burden from the programmer of implementing such a comparison themselves in their Paladim code. However in order for this to make sense, the built-in equal operation between arrays must be more efficient than a compiled version a similar comparison written in Paladim.

I find this difficult to assess, since i have not tried comparing the compiled code generated from similar Paladim code, with the Mips code generated by the following builtin extension. I would assume however, that the builtin functionality is going to be slightly more efficient, and since we can also have better type-checking on the built-in functionality, it should be preferred.

2.1 Type-checker changes.

In order to ensure that we correctly type-check the equal operation on arrays, we need to check the dimensions an array type of both arrays, and take into account the fact that either may be an array-literal. We do this in Type.sml in the following snippet:

Listing 1: Type.sml changes for array equality.

```
| typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
    let val e1_try = SOME ( typeCheckExp (vtab, e1, UnknownType) )  ↩
       handle _ => NONE
        val e2_try = SOME ( typeCheckExp (vtab, e2, UnknownType) )  ↩
           handle _ => NONE
        val (e1_new, e2_new) =
           case (e1_try, e2_try) of
               (SOME a, SOME b) => (a, b)
             | (SOME a, NONE  ) => (a, typeCheckExp (vtab, e2,  ↩
               KnownType(typeOfExp a)))
             | (NONE  , SOME b) => (typeCheckExp (vtab, e1, KnownType( ↩
               typeOfExp b)), b)
             | (NONE  , NONE  ) => raise Error("in type check equal,  ↩
               neither operand" ^ "type-checks (possibly polymorphism) ↩
               ," ^ " at ", pos)
```

```
    val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)

    val _ = typesEqual(tp1, tp2) orelse raise Error("in type check
       equal, argument types do not match, "^ pp_type tp1^" isn't "
       ^pp_type tp2^" at ", pos)

    (* Find the rank of an Array var *)
    val r1 = case tp1 of Array (rank, _) => rank
                                          | _ => ~1
    val r2 = case tp2 of Array (rank, _) => rank
                                          | _ => ~1

    (* Handle comparison between Array and ArrLit. Checks if they
       have the same rank. *)
    val (rank1, rank2) = case (e1_new, e2_new) of
         (ArrLit(exps1, _, _), ArrLit(exps2, _, _)) =>
             (length(exps1), length(exps2))
       | (ArrLit(exps1, _, _), LValue(_)) =>
             (length(mkShape e1_new), r2)
       | (LValue(_), ArrLit(exps2, _, _)) =>
             (r1, length(mkShape e2_new))
       | _ => (r1, r2)

    val _ = case rank1 = rank2 of
              false => raise Error("in type check equal, array
                 dimensions do not match at ", pos)
            | _ => ()

in Equal(e1_new, e2_new, pos) end
```

2.2 Code generation.

Since we have already checked that the arrays have matching dimensions in the type-checker, we can go ahead and simply check whether or not we have array operators. If we do not, we simply use the existing functionality.

Listing 2: Compiler.sml check for arrays in equal operation.

```
...
val (rank1, arrType1) = case typeOfExp e1 of
    Array (rank, bType) => (rank, bType)
  | _ => (~1, Int)
val (rank2, arrType2) = case typeOfExp e2 of
    Array (rank, bType) => (rank, bType)
  | _ => (~1, Int)
...
(* handle arrays *)
if rank1 <> ~1 andalso rank2 <> ~1 then
... let ...
else
...
```

Afterwards we can handle the comparison between the two arrays. We do this by finding the length of the arrays, making sure the lengths are equal, and iterating over the array elements. This needs to be done in Mips code, since we cannot get this information in the compiling stage.

Listing 3: Compiler.sml calculate array lengths in equal operation.

```
...
let
    val temp = "temp_" ^ newName()
    val arrLen1 = "arr1_len" ^ newName()
    val arrLen2 = "arr2_len" ^ newName()

    fun arrayLength (0, _, _)  = []
      | arrayLength (r, a, arr) = [Mips.LW (temp, a, makeConst(r*4-4))] ↩
          @
                                  [Mips.MUL (arr, arr, temp)] @
                                  arrayLength(r - 1, a, arr)

    val increment = case arrType1 of Int => 4
                                   | Bool => 1
                                   | Char => 1
```

```
val loadType = case arrType1 of Int => Mips.LW
                              | Bool => Mips.LB
                              | Char => Mips.LB

val arr1_el = "arr_eq1_" ^ newName()
val arr2_el = "arr_eq2_" ^ newName()
val arrLabel = "_equal_array_" ^ newName()
val notEqualLabel = "_not_equal_" ^ newName()
val isEqualLabel = "_is_equal_" ^ newName()
val continueLabel = "_continue_program_" ^ newName()
val iterator = "iterator_" ^ newName()

val c = c1 @ c2 @
        [ Mips.LI (arrLen1, "1"),
          Mips.LI (arrLen2, "1") ] @
        (arrayLength (rank1, t1, arrLen1)) @
        (arrayLength (rank2, t2, arrLen2)) @
        [ Mips.BNE (arrLen1, arrLen2, notEqualLabel) ] @
...
```

In the last line the program jumps to *notEqualLabel*, if the lengths are unequal, instead of throwing an error.
If the length check passes, the rest of the code needs to be executed. The code which actually does the comparison. First we skip the dimensions and strides, in order to get to the address pointing to the actual array elements.

We loop through the array, and load in the elements into $arr1_el$ and $arr2_el$. We then increment the counter *iterator*, which holds our position in the arrays, and $t1$ and $t2$ which holds our array address references. The amount which we increment $t1$ and $t2$ is calculated cased on what type of arrays we are comparing, and stored in *increment*.

If $arr1_el$ is unequal to $arr2_el$, we jump to *notEqualLabel*. If they are equal, we continue to next iteration. If the loop reaches the last element without jumping to *notEqualLabel*, we jump to *isEqualLabel*.

Listing 4: Compiler.sml calculate whether arrays are equal in equal operation.

```
  ...
[ Mips.LW (t1, t1, makeConst((rank1*2-1)*4)),
  Mips.LW (t2, t2, makeConst((rank1*2-1)*4)) ] @
[ Mips.LI (iterator, "0"),
  Mips.LABEL arrLabel,
  loadType (arr1_el, t1, "0"),
  loadType (arr2_el, t2, "0"),
  Mips.ADDI (iterator, iterator, "1"),
  Mips.ADDI (t1, t1, makeConst(increment)),
  Mips.ADDI (t2, t2, makeConst(increment)),
  Mips.BNE (arr1_el, arr2_el, notEqualLabel),
  Mips.BNE (arrLen1, iterator, arrLabel),
  Mips.BEQ (arrLen1, iterator, isEqualLabel),
  Mips.LABEL notEqualLabel,
  Mips.LI (place, "0"),
  Mips.J continueLabel,
  Mips.LABEL isEqualLabel,
  Mips.LI (place, "1"),
  Mips.LABEL continueLabel ]
  ...
```

We then simply append all of the newly generated Mips code by returning c.

2.3 Testing.

In the test file *arrayEqual.pal*, which can be in appendix A.4.1, we test the following scenarios:

- Array equal to itself. This should be true.
- Array equal to array of same length and rank, but with reversed elements. This should be false.
- Array equal to array containing the same elements. This should be true.
- Array of multiple dimensions equal to array of same dimensions, with same elements. This should be true.
- Array of basic type char, equal to array of basic type char, with the same elements. This should be true.
- Array of basic type char, equal to array of basic type char, with different elements. This should be false.
- Array of basic type bool, equal to array of basic type bool, with the same elements. This should be true.
- Array of basic type bool, equal to array of basic type bool, with different elements. This should be false.
- Two literals of same type equal to each other. This is done to ensure the existing equal functionality was not broken by array equal implementation. This should be true.
- Array literal equal to array of same type, but with different length. This is done to ensure we can compare array literals with array variables. This should be false.

All of these tests pass with expected result.

3 A Flat-Reduce High-Order Function in Paladim.

In order to implement the higher-order function *reduce* in paladim, we need to add functionality to the type-checker and the code generator. Both already contains pattern matching for a *reduce* function, so we simply extend it.

I personally find this built-in function to be of much less use than the equal operation. Even though the function definetly have use cases, the Paladim language in its current state, have very few operations on its basic types. This significantly limits the usefulness in my oppinion.

3.1 Type-checker changes.

We type-check *reduce* by first checking that we have the correct arguments. Since *reduce* requires a function name followed by an array, we raise errors in all other cases. We then check the type of the array, and the return-type of the given function. If these do not match, we raise an error. Lastly we return a *Red* with our new type-checked arguments.

Listing 5: Extend Type.sml typeCheckExp to handle reduce.

```
...
| typeCheckExp ( vtab, AbSyn.FunApp ("reduce", args, pos), etp ) =
    (* result should use TpAbSyn.sml's Exp constructor: Red of FIdent * ↩
        Exp * Pos *)
    let
        val (funId, arrArg) = case (hd args, hd (tl args)) of
            (AbSyn.LValue(AbSyn.Var(funId), pos), arrArg) => (funId, ↩
               arrArg)
          | _ => raise Error("the second argument of reduce must be an ↩
            array.", pos)

        val (tps, rtp) = case SymTab.lookup funId (!functionTable) of
            SOME (tps, SOME rtp) => (tps, rtp)
          | _ => raise Error("the first argument of reduce must be a ↩
            function name", pos)

        val new_arrArg = typeCheckExp (vtab, arrArg, UnknownType)

        val _ = case typeOfExp new_arrArg of
            Array (_, bVal) =>
```

```
                if (BType(bVal) <> rtp) then
                    raise Error("function return type and the array  ↩
                       must be of the same type.", pos)
                else ()
          | _ => raise Error("the second argument of reduce must be an  ↩
              array.", pos)

        val fIdent = (funId, (tps, SOME rtp))
    in
        Red(fIdent, new_arrArg, pos)
    end
...
```

3.2 Code generation.

Firstly we compile our input array meta data, and store the reference to the meta data in *arr_id*. Since we want to iterate over the array, we set *increment* to 4 or 1 based on whether or not our array is of type integer, like we do in our array equality operation.

We then find the length of the array which we also need for iteraiton. If the length of the array is 1, we need to raise a runtime error. The input function takes two arguments, and as such an array of minimum length 2 is required.
To have a slightly more sensible error for this special case, we add a new runtime error *_IllegalReduceInputArrSizeError_*.

If no error is thrown, we move on to fetch the address of the actual array elements from the meta data. We store this address by overwriting *arr_id* which we no longer need. We can now load the first element from the array into *el*1. We then increment our *arr_id* and our *iterator* which counts the number of elements passed.

We then add the loop label *loopLabel*, and load the second array element into *el*2. Since we now have both elements, we can call $mkFunCallCode$ on our fIdent and our elements. We store the result in *temp*, and then move the result back into *el*1. This way *el*1 becomes the result of applying our input function to the first two elements.
We can now increment *arr_id* and *iterator*, and continue the loop. The next loop iteration will store the next array element into *el*2, and then do $mkFunCallCode$ on this element and the result from our previous $mkFunCallCode$, which is stored in *el*1.

Listing 6: Extend Compiler.sml to support reduce.

```
...
| compileExp( vtab, Red(fIdent, arr, pos), place ) =
    let
        val arr_id = "reduce_array_" ^ newName()
        val new_arr = compileExp(vtab, arr, arr_id)

        val (funId, returnType) =
            case fIdent of (funId, (argTypes, SOME returnType)) =>
                (funId, returnType)
              | _ => raise Error("Invalid reduce arguments", pos)

        val increment = case returnType of
            BType(Int) => "4"
          | _ => "1"

        val loadType = case returnType of
            BType(Int) => Mips.LW
          | _ => Mips.LB

        val rank = case typeOfExp arr of
            Array(rank, _) => rank
          | _ => raise Error("Input is not an array", pos)

        val temp = "temp_" ^ newName()
        val arrLen = "arr_len" ^ newName()

        fun arrayLength (0, _, arr)  = []
          | arrayLength (r, a, arr) = [Mips.LW (temp, a, makeConst(r ↩
            *4-4))] @
                                      [Mips.MUL (arr, arr, temp)] @
                                      arrayLength(r - 1, a, arr)

        val iterator = "iterator_" ^ newName()
        val loopLabel = "_reduce_loop_" ^ newName()
        val exitLabel = "_exit_reduce_" ^ newName()

        val el1 = "arr_el1" ^ newName()
        val el2 = "arr_el2" ^ newName()
        val code = (mkFunCallCode (fIdent, [el1, el2], vtab, temp)) @
                   [ Mips.MOVE (el1, temp) ]
```

```
    in
        new_arr @
        [ Mips.LI (arrLen, "1") ] @
        (arrayLength (rank, arr_id, arrLen)) @
        [ Mips.LI (temp, "1"),
          Mips.BEQ (arrLen, temp, "_IllegalReduceInputArrSizeError_"),
          Mips.LW (arr_id, arr_id, makeConst((rank*2-1)*4)),
        loadType (el1, arr_id, "0"),
          Mips.LI (iterator, "1") ] @
        [ Mips.ADDI (arr_id, arr_id, increment),
          Mips.LABEL loopLabel,
        loadType (el2, arr_id, "0") ] @
        code @
        [ Mips.ADDI (iterator, iterator, "1"),
          Mips.ADDI (arr_id, arr_id, increment),
          Mips.BNE (arrLen, iterator, loopLabel),
          Mips.MOVE (place, el1),
          Mips.LABEL exitLabel ]
    end
...
```

3.3 Testing.

In the test file *reduceTest.pal*, which can be in appendix A.4.2, we test the following scenarios:

- Adding together all the int elements of a multi-dimensional array, the expected output of $f(f(f(f(el1, el2), el3), el4)..., eln)$. In our case it should be 21 since we perform (((((1+2)+3)+4)+5)+6).
- Comparing boolean elements, to ensure we can sure any basic type.

Both of these tests performs as expected. Unfortunately i was unable to find a way to test char arrays, since Paladim has no default char operations.

4 A switch statement for Paladim.

The switch statement is a very common and widely used control-structure in many different programming languages. In enables for slightly more efficient execution of many simple literal comparisons. In my oppinion it also greatly improves code readability, over having a long if-then-else block.

4.0.1 Lexer, grammar and abstract syntax tree.

First we need to change the grammar and the lexer to allow for the new structure. We do this by adding the following productions.

Listing 7: Switch structure changes to Parser.grm

```
...
%token <(int*int)> SWITCH DEFAULT CASE
...

%type <AbSyn.Case>         Case
%type <AbSyn.Case list>    Cases
%type <AbSyn.BasicVal>     Lit
...
        | SWITCH Exp COLON Cases { AbSyn.Switch  ($2, $4, $3) }
;
Cases   : Case SEMICOL Cases     { $1 :: $3 }
        | DEFAULT COLON Blk      { [ AbSyn.DefaultCase ($3, $2) ] }
;
Case    : CASE Lit COLON Blk     { AbSyn.Case    ($2, $4, $3) }
;
Lit     : NUM                  { AbSyn.Num(#1 $1) }
        | TRUE                 { AbSyn.Log(true ) }
        | FALSE                { AbSyn.Log(false) }
        | CHARLIT              { AbSyn.Chr(#1 $1) }
;
```

And the following lexer tokens.

Listing 8: Switch structure changes to Lexer.lex

```
...
| "switch"       => Parser.SWITCH  pos
| "case"         => Parser.CASE    pos
| "default"      => Parser.DEFAULT pos
...
```

The same structure needs to be added to the abstract syntax tree, and the typed abstract syntax tree. Here we only show one figure, since the changes are identical.

Listing 9: Switch structure changes to AbSyn.sml and TpAbSyn.sml

```
...
| Switch of Exp * Case list * Pos

and Case = Case of BasicVal * StmtBlock * Pos
  | DefaultCase of StmtBlock * Pos
...
```

4.0.2 Type checking.

Next we need to add type checking for the switch statement. We need to recursively typecheck the cases. We do this by adding a new function *typeCheckCases* which returns a list of *TpAbSyn.Case* and a single *TpAbSyn.DefaultCase*, both with their statement block typechecked.

Listing 10: Switch statement type checking in Type.sml

```
...
(* Switch statement type checking *)
| typeCheckStmt ( vtab, AbSyn.Switch (exp, cases, pos) ) =
    let
        val new_exp = typeCheckExp (vtab, exp, UnknownType)
        val new_cases = typeCheckCases (vtab, cases)
    in
        Switch(new_exp, new_cases, pos)
    end
...
(* typeCheckCases - type checks a list of AbSyn.Case and returns a list ↩
    of TpAbSyn.Case *)
and typeCheckCases ( vtab, AbSyn.Case(basicVal, stmtBlock, pos)::cs ) =
      let val new_bval = case basicVal of AbSyn.Num x => Num x
                                        | AbSyn.Chr x => Chr x
                                        | AbSyn.Log x => Log x
      in
          Case(new_bval, typeCheckBlock (vtab, stmtBlock), pos):: ↩
             typeCheckCases(vtab, cs)
      end
  | typeCheckCases ( vtab, AbSyn.DefaultCase(stmtBlock, pos)::cs ) =
      DefaultCase(typeCheckBlock (vtab, stmtBlock), pos):: ↩
         typeCheckCases(vtab, cs)
  | typeCheckCases ( _, [] ) = []
...
```

4.0.3 Code generation.

Lastly we need to add Mips code in the compiler. We want to first add Mips code for the expression, which is the switch statement condition. Then a branch on equal code for each case, such that each case compares it's literal with condition of the statement. If these arguments are equal, we jump to the label associated with the case.

After these branch codes, we simply add the default label, and the code for the default statement block. This means that if none of the branch on equal codes were triggered, we automatically enter the default label. When the default code is executed, we jump to the exit label for the switch statement, called *_switch_end_*.

After the default label, we add labels and their corresponding statement blocks for each case. These are the labels we jump to in the branch on equal codes above. Each of these labels end by jumping to the exit label. After this label, the rest of the program continues.

Listing 11: Compiler code for generating the Mips instructions

```
...
| Switch(exp, cases, pos) =>
    let
        val switchCondition = "_switch_con_" ^ newName()
        val switchExit = "_switch_end_" ^ newName()
        val caseLabels = generateCaseLabels( cases )
        val caseLabelBlks = generateCaseLabelBlks( vtable, switchExit, ↩
            caseLabels, cases )

        val condition = compileExp( vtable, exp, switchCondition )
        val casesMips = compileCases( vtable, switchCondition, ↩
            caseLabels, cases )
        val defaultCaseMips = compileDefaultCase( vtable, switchExit, ↩
            cases )
    in
        condition @
        casesMips @
        defaultCaseMips @
        caseLabelBlks @
        [ Mips.LABEL switchExit ]
    end
...
```

4.1 Testing.

In the test file *switchTest.pal*, which can be in appendix A.4.3, we test the following scenarios:

- Switching on an input integer.
- Switching on a boolean literal.
- Switching on a boolean variable.
- Switching on a char literal.
- Switching on a char variable.

All of these scenarios enters the expected switch case.

A Code changes

A.1 Array equal implementation.

A.1.1 Type.sml

```
        ...
| typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
      let val e1_try = SOME ( typeCheckExp (vtab, e1, UnknownType) )  ↩
         handle _ => NONE
          val e2_try = SOME ( typeCheckExp (vtab, e2, UnknownType) )  ↩
             handle _ => NONE
          val (e1_new, e2_new) =
              case (e1_try, e2_try) of
              (SOME a, SOME b) => (a, b)
            | (SOME a, NONE  ) => (a, typeCheckExp (vtab, e2, KnownType ↩
               (typeOfExp a)))
            | (NONE  , SOME b) => (typeCheckExp (vtab, e1, KnownType( ↩
               typeOfExp b)), b)
            | (NONE  , NONE  ) => raise Error("in type check equal,  ↩
               neither operand" ^
                                              "type-checks (possibly  ↩
                                                 polymorphism)," ^
                                              " at ", pos)

          val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)

          val _ = typesEqual(tp1, tp2) orelse
                  raise Error("in type check equal, argument types do  ↩
                     not match, "^
                              pp_type tp1^" isn't "^pp_type tp2^" at ", ↩
                                  pos)

          (* Find the rank of an Array var *)
          val r1 = case tp1 of Array (rank, _) => rank
                                               | _ => ~1
          val r2 = case tp2 of Array (rank, _) => rank
                                               | _ => ~1

          (* Handle comparison between Array and ArrLit. Checks if they ↩
              have the same rank. *)
          val (rank1, rank2) = case (e1_new, e2_new) of
```

```
                (ArrLit(exps1, _, _), ArrLit(exps2, _, _)) =>
                    (length(exps1), length(exps2))
              | (ArrLit(exps1, _, _), LValue(_)) =>
                    (length(mkShape e1_new), r2)
              | (LValue(_), ArrLit(exps2, _, _)) =>
                    (r1, length(mkShape e2_new))
              | _ => (r1, r2)

        val _ = case rank1 = rank2 of
                    false => raise Error("in type check equal, array dimensions do not match at ", pos)
                  | _ => ()

    in Equal(e1_new, e2_new, pos) end
    ...
```

A.1.2 Compiler.sml

```
      ...
| compileExp( vtable, Equal(e1, e2, _), place ) =
      let
          val t1 = "eq1_" ^ newName()
          val c1 = compileExp(vtable, e1, t1)
          val t2 = "eq2_" ^ newName()
          val c2 = compileExp(vtable, e2, t2)
          val lEq = "_equal_" ^ newName()

          val p = c1 @ c2 @
                  [ Mips.LI (place,"1"), Mips.BEQ (t1, t2, lEq),
                    Mips.LI (place,"0"), Mips.LABEL lEq ]

          val (rank1, arrType1) = case typeOfExp e1 of Array (rank, bType) => (rank, bType)
                                                     | _ => (~1, Int)
          val (rank2, arrType2) = case typeOfExp e2 of Array (rank, bType) => (rank, bType)
                                                     | _ => (~1, Int)
      in
          (* handle arrays *)
          if rank1 <> ~1 andalso rank2 <> ~1 then
              let
                  val temp = "temp_" ^ newName()
```

```
val arrLen1 = "arr1_len" ^ newName()
val arrLen2 = "arr2_len" ^ newName()

fun arrayLength (0, _, _)  = []
  | arrayLength (r, a, arr) = [Mips.LW (temp, a,  ↩
     makeConst(r*4-4))] @
                              [Mips.MUL (arr, arr,  ↩
                                 temp)] @
                              arrayLength(r - 1, a, ↩
                                  arr)

val increment = case arrType1 of Int => 4
                               | Bool => 1
                               | Char => 1

val loadType = case arrType1 of Int => Mips.LW
                               | Bool => Mips.LB
                               | Char => Mips.LB

val arr1_el = "arr_eq1_" ^ newName()
val arr2_el = "arr_eq2_" ^ newName()
val arrLabel = "_equal_array_" ^ newName()
val notEqualLabel = "_not_equal_" ^ newName()
val isEqualLabel = "_is_equal_" ^ newName()
val continueLabel = "_continue_program_" ^ newName ↩
   ()
val iterator = "iterator_" ^ newName()

val c = c1 @ c2 @
        [ Mips.LI (arrLen1, "1"),
          Mips.LI (arrLen2, "1") ] @
        (arrayLength (rank1, t1, arrLen1)) @
        (arrayLength (rank2, t2, arrLen2)) @
        [ Mips.BNE (arrLen1, arrLen2, notEqualLabel ↩
           ) ] @
        [ Mips.LW (t1, t1, makeConst((rank1*2-1)*4) ↩
           ),
          Mips.LW (t2, t2, makeConst((rank1*2-1)*4) ↩
             ) ] @
        [ Mips.LI (iterator, "0"),
```

```
                    Mips.LABEL arrLabel,
                    loadType (arr1_el, t1, "0"),
                    loadType (arr2_el, t2, "0"),
                    Mips.ADDI (iterator, iterator, "1"),
                    Mips.ADDI (t1, t1, makeConst(increment)),
                    Mips.ADDI (t2, t2, makeConst(increment)),
                    Mips.BNE (arr1_el, arr2_el, notEqualLabel ↩
                        ),
                    Mips.BNE (arrLen1, iterator, arrLabel),
                    Mips.BEQ (arrLen1, iterator, isEqualLabel ↩
                        ),
                    Mips.LABEL notEqualLabel,
                    Mips.LI (place, "0"),
                    Mips.J continueLabel,
                    Mips.LABEL isEqualLabel,
                    Mips.LI (place, "1"),
                    Mips.LABEL continueLabel ]
        in
            c
        end
    else
        p
end
...
```

A.2 Reduce function implementation

A.2.1 Type.sml

```
...
| typeCheckExp ( vtab, AbSyn.FunApp ("reduce", args, pos), etp ) =
        (* result should use TpAbSyn.sml's Exp constructor: Red of
           FIdent * Exp * Pos *)
        let
            val (funId, arrArg) = case (hd args, hd (tl args)) of
                (AbSyn.LValue(AbSyn.Var(funId), pos), arrArg) => (funId
                    , arrArg)
              | _ => raise Error("the first argument of reduce must be
                 an array.", pos)

            val (tps, rtp) = case SymTab.lookup funId (!functionTable)
               of
                  SOME (tps, SOME rtp) => (tps, rtp)
                | _ => raise Error("the first argument of reduce must
                   be a function name", pos)

            val new_arrArg = typeCheckExp (vtab, arrArg, UnknownType)

            val _ = case typeOfExp new_arrArg of
                Array (_, bVal) =>
                    if (BType(bVal) <> rtp) then
                        raise Error("function return type and the array
                            must be of the same type.", pos)
                    else ()
              | _ => raise Error("the second argument of reduce must be
                  an array.", pos)

            val fIdent = (funId, (tps, SOME rtp))
        in
            Red(fIdent, new_arrArg, pos)
        end
...
```

A.2.2 Compiler.sml

```
...
  (*** EXAM: flat Reduce ***)
    | compileExp( vtab, Red(fIdent, arr, pos), place ) =
        let
            val arr_id = "reduce_array_" ^ newName()
            val new_arr = compileExp(vtab, arr, arr_id)

            val (funId, returnType) =
                case fIdent of (funId, (argTypes, SOME returnType)) =>
                    (funId, returnType)
                  | _ => raise Error("Invalid reduce arguments", pos)

            val increment = case returnType of
                BType(Int) => "4"
              | _ => "1"

            val loadType = case returnType of
                BType(Int) => Mips.LW
              | _ => Mips.LB

            val rank = case typeOfExp arr of
                Array(rank, _) => rank
              | _ => raise Error("Input is not an array", pos)

            val temp = "temp_" ^ newName()
            val arrLen = "arr_len" ^ newName()

            fun arrayLength (0, _, arr)  = []
              | arrayLength (r, a, arr) = [Mips.LW (temp, a, makeConst( ↩
                r*4-4))] @
                                          [Mips.MUL (arr, arr, temp)] @
                                          arrayLength(r - 1, a, arr)

            val iterator = "iterator_" ^ newName()
            val loopLabel = "_reduce_loop_" ^ newName()
            val exitLabel = "_exit_reduce_" ^ newName()

            val el1 = "arr_el1" ^ newName()
            val el2 = "arr_el2" ^ newName()
```

```
            val code = (mkFunCallCode (fIdent, [el1, el2], vtab, temp)) ↩
                @
                        [ Mips.MOVE (el1, temp) ]
        in
            new_arr @
            [ Mips.LI (arrLen, "1") ] @
            (arrayLength (rank, arr_id, arrLen)) @
            [ Mips.LI (temp, "1"),
              Mips.BEQ (arrLen, temp, "_IllegalReduceInputArrSizeError_ ↩
                 "),
              Mips.LW (arr_id, arr_id, makeConst((rank*2-1)*4)),
              loadType (el1, arr_id, "0"),
              Mips.LI (iterator, "1") ] @
            [ Mips.ADDI (arr_id, arr_id, increment),
              Mips.LABEL loopLabel,
              loadType (el2, arr_id, "0") ] @
              code @
            [ Mips.ADDI (iterator, iterator, "1"),
              Mips.ADDI (arr_id, arr_id, increment),
              Mips.BNE (arrLen, iterator, loopLabel),
              Mips.MOVE (place, el1),
              Mips.LABEL exitLabel ]
        end
...
        Mips.ALIGN "2",
        Mips.LABEL "_IllegalReduceInputArrSizeString_",
        Mips.ASCIIZ "Error: Array size less or equal to 1 at line ",
...
```

A.3 Switch statement implementation.

A.3.1 Lexer.lex

```
...
| "switch"        => Parser.SWITCH  pos
| "case"          => Parser.CASE    pos
| "default"       => Parser.DEFAULT pos
...
```

A.3.2 Parser.grm

```
...
%token <(int*int)> SWITCH DEFAULT CASE
...

%type <AbSyn.Case>         Case
%type <AbSyn.Case list>    Cases
%type <AbSyn.BasicVal>     Lit
...
 | SWITCH Exp COLON Cases { AbSyn.Switch  ($2, $4, $3) }
;
Cases    : Case SEMICOL Cases      { $1 :: $3 }
         | DEFAULT COLON Blk       { [ AbSyn.DefaultCase ($3, $2) ] }
;
Case     : CASE Lit COLON Blk      { AbSyn.Case    ($2, $4, $3) }
;
Lit      : NUM                  { AbSyn.Num(#1 $1) }
         | TRUE                 { AbSyn.Log(true ) }
         | FALSE                { AbSyn.Log(false) }
         | CHARLIT              { AbSyn.Chr(#1 $1) }
;
```

A.3.3 AbSyn.sml

```
    ...
           | Switch of Exp * Case list * Pos
    and Case = Case of BasicVal * StmtBlock * Pos
           | DefaultCase of StmtBlock * Pos
    ...
```

A.3.4 TpAbSyn.sml

```
...
        | Switch of Exp * Case list * Pos
and Case = Case of BasicVal * StmtBlock * Pos
        | DefaultCase of StmtBlock * Pos
...
```

A.3.5 Type.sml

```
...
(* Switch statement type checking *)
| typeCheckStmt ( vtab, AbSyn.Switch (exp, cases, pos) ) =
  let
      val new_exp = typeCheckExp (vtab, exp, UnknownType)
      val new_cases = typeCheckCases (vtab, cases)
  in
      Switch(new_exp, new_cases, pos)
  end
...

(* typeCheckCases - type checks a list of AbSyn.Case and returns a
   list of TpAbSyn.Case *)
and typeCheckCases ( vtab, AbSyn.Case(basicVal, stmtBlock, pos)::cs
    ) =
    let val new_bval = case basicVal of AbSyn.Num x => Num x
                                      | AbSyn.Chr x => Chr x
                                      | AbSyn.Log x => Log x
    in
        Case(new_bval, typeCheckBlock (vtab, stmtBlock), pos)::
           typeCheckCases(vtab, cs)
    end
| typeCheckCases ( vtab, AbSyn.DefaultCase(stmtBlock, pos)::cs ) =
    DefaultCase(typeCheckBlock (vtab, stmtBlock), pos)::
       typeCheckCases(vtab, cs)
| typeCheckCases ( _, [] ) = []
...
```

A.3.6 Compiler.sml

```
    ...
    | compileStmt(vtable, s, exitLabel) =
    ...
    | Switch(exp, cases, pos) =>
          let
              val switchCondition = "_switch_con_" ^ newName()
              val switchExit = "_switch_end_" ^ newName()
              val caseLabels = generateCaseLabels( cases )
              val caseLabelBlks = generateCaseLabelBlks( vtable, switchExit, caseLabels, cases )

              val condition = compileExp( vtable, exp, switchCondition )
              val casesMips = compileCases( vtable, switchCondition, caseLabels, cases )
              val defaultCaseMips = compileDefaultCase( vtable, switchExit, cases )
          in
              condition @
              casesMips @
              defaultCaseMips @
              caseLabelBlks @
              [ Mips.LABEL switchExit ]
          end
...
and compileDefaultCase ( vtab, switchExit, DefaultCase( blk, pos )::cs ) =
      [ Mips.LABEL ("_default_case_" ^ newName()) ] @
      ( compileStmts blk vtab ("_condition_end_" ^ newName()) ) @
      [ Mips.J switchExit ]
  | compileDefaultCase ( vtab, switchExit, Case(bVal, blk, pos)::cs ) =
      compileDefaultCase( vtab, switchExit, cs )
  | compileDefaultCase ( _, _, [] ) = []

(* generates the branch-jumps which we use to run the correct case. *)
and compileCases ( vtab, condition, l::ls, Case(bVal, blk, pos)::cs ) =
      let
```

```
            val bValReg = "_case_bval_" ^ newName()
        in
            bValToReg ( bValReg, bVal ) @
            [ Mips.BEQ ( bValReg, condition, l) ] @
              compileCases( vtab, condition, ls, cs)
        end
  | compileCases ( vtab, condition, ls, DefaultCase( blk, pos )::cs ) ↩
      =
      compileCases(vtab, condition, ls, cs)
  | compileCases ( _, _, [], []) = []
  | compileCases ( _, _, _, _) = raise Error("The compiler does not  ↩
     work, and there is nothing you can do about it.", (~1, ~1))

(* generate labels for all cases, except the default case *)
and generateCaseLabels ( Case( _, _, _ )::cs ) =
      ("_case_label_" ^ newName())::generateCaseLabels(cs)
  | generateCaseLabels ( DefaultCase( blk, pos )::cs ) =
      generateCaseLabels(cs)
  | generateCaseLabels ( [] ) = []

(* generate code for the blocks inside each case, except the default  ↩
   case *)
and generateCaseLabelBlks ( vtab, switchExit, l::ls, Case(bVal, blk,  ↩
   pos)::cs ) =
      [ Mips.LABEL l ] @
      ( compileStmts blk vtab ("_condition_end_" ^ newName()) ) @
      [ Mips.J switchExit ] @
      generateCaseLabelBlks( vtab, switchExit, ls, cs )
  | generateCaseLabelBlks ( vtab, switchExit, ls, DefaultCase( blk,  ↩
     pos )::cs ) =
      generateCaseLabelBlks( vtab, switchExit, ls, cs )
  | generateCaseLabelBlks ( _, _, [], [] ) = []
  | generateCaseLabelBlks ( _, _, _, _ ) = raise Error("The compiler  ↩
     does not work, and there is nothing you can do about it.", (~1,  ↩
     ~1))

(* convert basic value to something which can be stored in a register ↩
    *)
and bValToReg (reg, Num n) = [ Mips.LI (reg, makeConst n) ]
  | bValToReg (reg, Log b) =
      (case b of true => [ Mips.LI (reg, "1") ]
```

```
            | false => [ Mips.LI (reg, "0") ])
  | bValToReg (reg, Chr c) = [ Mips.LI (reg, makeConst(ord c)) ]
  ...
```

A.3.7 TpInterpret.sml

```
  ...
  | execStmt ( _, _, _) = raise Error("Unimplemented!", (~1, ~1))
  ...
  (*| evalExp ( ZipWith ((fid,signat), arrexp1, arrexp2, pos), vtab,
     ftab ) =
      raise Error("In TpInterpret, zipWith is Unimplemented!", pos)*)

  | evalExp _  = raise Error("Unimplemented!", (0,0))

  end
```

A.4 Tests

All test files can be found in the DATA/ folder, along with their .in and .out files.

A.4.1 Equal

The .out file for the following test is "1011100010"

Listing 12: arrayEqual.pal

```
program arrayEqual;

procedure main()
var
    arr1 : array of int;
    arr2 : array of int;
    arr3 : array of int;
    arr5 : array of int;
begin
    arr1 := new(2);
    arr2 := new(2);
    arr5 := new(2);

    arr1 := {0, 1};
    arr2 := {1, 0};
    arr3 := {1, 2, 3, 4};
    arr5 := {0, 1};

    write(arr1 = arr1); // true
    write(arr1 = arr2); // false
    write(arr1 = arr5); // true
    write({{1, 0}, {2, 3}} = {{1, 0}, {2, 3}}); // true
    write({'1', '2', '3'} = {'1', '2', '3'}); // true
    write({'1', '2', '3'} = {'1', '2', '4'}); // false
    write({true, false} = {true, true}); // false
    write({true, false} = {false, false}); // false
    write(1 = 1); // true
    write({1, 2, 3} = arr3); // false
end;
```

A.4.2 Reduce

The .out file for the following test is "211"

Listing 13: reduceTest.pal

```
program reduceTest;

function plus(x : int; y : int) : int
    return (x + y);

function andalso(x : bool; y : bool) : bool
        return (x = y);

procedure main()
var
    x : int;
    y : bool;
    a : array of array of int;
    b : array of array of bool;
begin
    a := {{1,2,3},{4,5,6}};
    b := {{true,true,true},{true,true,true}};

    x := reduce(plus, a);

    write(x);

    y := reduce(andalso, b);

    write(y);
end;
```

A.4.3 Switch

The .out file for the following test is

i is 1

true

true

i is 1

i is 0

Listing 14: switchTest.pal

```
program switchTest;

procedure main()
var
    i : int;
    x : char;
    y : bool;
begin
    i := read();
    switch i:
        case 0: write("is is 0!");
        case 1: write("i is 1");
        default: write("is neither 0 nor 1.");

    write("\n");

    switch true:
        case true: write("true");
        case false: write("false");
        default: write("NO");

    write("\n");

    y := true;

    switch y:
        case true: write("true");
        case false: write("false");
        default: write("NO");

    write("\n");
```

```
    switch '1':
        case '2': write("i is 2");
        case '0': write("i is 0");
        case '1': write("i is 1");
        default: write("NO");

    write("\n");

    x := '0';

    switch x:
        case '2': write("i is 2");
        case '0': write("i is 0");
        case '1': write("i is 1");
        default: write("NO");
end;
```