Function

When calling a function, first we need to check the number of actual and formal parameters, which shall be equal. Unless the function is elliptic, in which case the actual parameter list can be longer.

Each regular actual parameter is transformed to the type of the formal parameter, and each extra actual parameter is subjected to the type promotion described in Section **Fel! Hittar inte referenskälla.**. Unless the return type is void, we place the result value in a temporary symbol.

Each actual parameter is checked. For a regular parameter (paramType is not null), a function is converted to a pointer to a function, otherwise implicit type conversion occurs. For an extra parameter, a function, array, or string is converted to a pointer, otherwise argument promotion occurs.

For an extra parameter: a char is converted to an integer and a float is converted to a double.

Index

The index operator takes a pointer or array and an integral. We have three cases:

The index is constant, in which case we treat the operator as if it was the arrow operator.

The index is not constant and the array or pointer type size is one, in which case we generate that add the index to the pointer or array.

The index is not constant and the array or pointer type size is greater than, in which case we also need to generate code that multiply the index with the type size.

Dot

The dot operator generates different result is the operand has and address symbol. If it has, it is the result of the arrow, dereference, or index operator. In that case, it inherits the address symbol and increase the address offset with the member offset.

If the operand does not have an address symbol, it is a regular variable. Then we just create a temporary symbol with the same storage and increase the offset with the member offset.

Increment

The increment and decrement operators come in two forms: prefix and postfix. The difference is that in the prefix case the resulting value is value after the operator has been applied, while in the postfix case the resulting value is the original value before the operator has been applied. However, the side effects of the operators are the same: in the increment case the value is added by one, and in the decrement case the value is subtracted by one.

Default

The default statement have to comply with the following demands:

Main.m\_defaultStack must not be empty. If it is empty, the default statements misses a surrounding switch statements.

Since Main.m\_defaultStack is not empty, there is at least one surrounding switch statement. We call pop to see if the top value is minus one. If it is not, there has already been a default statement in the closest surrounding switch statement. If it is minus one, there has not been a earlier default statement and we push the line number at the beginning of the statement following the default statement. Since the line numbers are numbered from zero, it cannot be minus one.

While

The while header prepares the while statement by pushing break and continue stacks with integers sets. The continue set will eventually be backpatched to the beginning of the while statement and break set will be backpatched to the statement following the while statement.

The true set of the while expression is backpatched to the beginning of the statement (or the first statement of a block statement) inside the while loop while the false set is backpatched to the statement following the while statement, similar to the break set. In order to make sure that the while statement is followed by jump statement, the jump\_marker rule adds a jump statement at the end of the statement surrounded by the while statement, which is backpatched to the beginning of the while expression, similar to the continue set.

Header

The open statements include the compound statements if, if-else, switch, case, while, for, and label statements. The switch, while, and for statements need to do some preparation before the parsing, which means that they need one header rule each. Each rule calls its corresponding method in the Generate class that does the actual work of type checking and middle code generation.

Switch

The GenerateSwitchStatement method generates middle code jump instructions for each of the case statements, which are stored in the Main.CaseMapStack stack. Each entry in the stack holds an integer value (each case expression value must be possible to evaluate in compile-time) and a line number. If the original switch expression equals the constant case expression, we jump to the start line of the statement following the case expression.

If there is a default statements, we finally jump to its start line if the switch statement does not match any of the case statements. Note that the default statement does not have to be placed after the case statements, even though I can see no good reason not to do so.

Do

The do statement is a weaker version of the while statement. The difference is that the while expression may be false from the beginning resulting in zero iterations while the do expression is located at the end, resulting in at least one iteration. However, the do statement may in some cases present a simpler and more intuitive solution.

Like the while and for cases, the do statement is prepared by pushing the continue and break stacks. The true set of the do expression is backpatched to the beginning of the statement surrounded by the do statement, the false set is backpatched to the statement followed the do statement and finally the continue and break stacks are popped.

For

The for statement holds three optional expression: the initialization expression, the test expression, and the increment expression. The true and false sets of the initialization and increment expressions are all backpatched to beginning of the test expression. Like the while case, in order to make sure that the for statement is followed by jump statement the jump\_marker rule add a jump statement at the end of the statement surrounded by the for statement, which is also backpatched to the beginning of the test expression. There is also a jump line inserted after the test optional expression, which is backpatched to the beginning of the for statements to make sure that the for loop works properly even if the test expression has been omitted. An omitted test expression is equivalent to an infinitive loop.

Finally, like the while case, the continue set is backpatched to the beginning of the test expression and the break set is backpatched to the beginning of the statement following the while statement.

The for statement is more complicated than the do and while statements. It holds three optional expression: the initialization expression, the test expression, and the increment expression. The true and false sets of the initialization and increment expressions are all backpatched to beginning of the test expression. An omitted test expression is equivalent to an infinite loop.

Label goto

The label statement is quite simple, we just add the name of the label together with the line number of the beginning of the statement following the label to Main.LabelMap and check that the label has not been added already. The labels are used as targets of the goto statements. But, as we all know, goto has no place in well-structured programs. Labels and goto are included in C of historical reasons only. More recent languages have omitted goto.

Case

We call peek on the stack to obtain the map holding the case expression value of the closest surrounding switch statement, add the case value with the line number of the case statement, and check that there is no earlier cases expression with the same value (put returns the old value if the key is already present in the map, null if is not). Technically, it is possible to store null values in the map, in which case put would return null even when the key is present in the map. However, in this book, we never store null values in any map.

Default

Technically, the default statement does not have to be placed after the last case statements. However, we cannot have one default statement in a switch statement.

The GenerateReturnStatement method is surprisly complicated, which due to the fact the return statement shall be interpreted as an exit statement in case of the main function.

If the surrounding function returns a pointer to a function and the return expression is a or a pointer to a function, we check the they are equal; that is, that they return equal types and have equal parameter lists.

If the surrounding function returns a pointer and the expression type is array, we compare the pointer type with array type. If the return expression is a string we check that the surrounding function’s return type is a pointer to a (signed or unsigned) character. Otherwise, we just cast the return expression to the surrounding function’s return type.

If the surrounding function is the main function, we cast the return expression to a signed short integer, since we already checked that main does not return void, it has to return an integral type. If the surrounding function is not main, we generate middle code instructions for the return value and function return. Note that there are two different instructions, where the first one sets the return value and the second one performs the return jump.

On the other hand, if there is no return expression we check that the surrounding function returns void. If it is the main function, we add the exit middle code instruction with argument zero. If it is not, we just add the return middle code instruction without precede it with the setting of a return value.

We need to regard whether there is an expression and whether the return statement is located inside the main function.

public static Statement ReturnStatement(Expression expression) {

List<MiddleCode> codeList;

If the expression is not null, we need the check that the function does not return void. If it does, we report an error.

if (expression != null) {

Assert.Error(!SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),

Message.Non\_\_void\_return\_from\_void\_function);

We cast the return expression to the return type of the function.

expression = TypeCast.ImplicitCast(expression,

SymbolTable.CurrentFunction.Type.ReturnType);

codeList = expression.LongList;

If the function is the main function, we shall not return a value. Instead, we shall exit the program execution and return an integer value to the enclosing system.

If the function is not the main function, we return the value of the expression.

AddMiddleCode(codeList, MiddleOperator.Return,

null, expression.Symbol);

if (SymbolTable.CurrentFunction.UniqueName.Equals

(AssemblyCodeGenerator.MainName)) {

AddMiddleCode(codeList, MiddleOperator.Exit);

}

}

If the expression is null, we check that the function returns void. If it does not, we report an error.

else {

Assert.Error(SymbolTable.CurrentFunction.Type.ReturnType.IsVoid(),

Message.Void\_returned\_from\_non\_\_void\_function);

codeList = new List<MiddleCode>();

If the function is the main function, we exit the execution of the program without a return value.

If the function is not the main function, return the function without a return value.

AddMiddleCode(codeList, MiddleOperator.Return);

if (SymbolTable.CurrentFunction.UniqueName.Equals

(AssemblyCodeGenerator.MainName)) {

AddMiddleCode(codeList, MiddleOperator.Exit);

}

}

return (new Statement(codeList));

}

public static Expression LogicalOrExpression(Expression leftExpression,

Expression rightExpression) {

We check if the expression is constant. If it is constant, we return the constant expression.

Expression constantExpression =

ConstantExpression.Logical(MiddleOperator.LogicalOr,

leftExpression, rightExpression);

if (constantExpression != null) {

return constantExpression;

}

We type cast both the expressions to the logical type.

leftExpression = TypeCast.ToLogical(leftExpression);

rightExpression = TypeCast.ToLogical(rightExpression);

For the resulting expression to be true, it is enough that one of the left or right expression is true. Therefore, the true-set of the resulting expression is the union of the true-set of the left and right expression. If the left expression is evaluated to true, the right expression (including its side effects) shall not be evaluated. The false-sets, on the other hand, are different. If the left expression is evaluated to false, we need to evaluate the right expression. Therefore, we backpatch the false-set of the left expression to the beginning of the right expression code. The false-set of the resulting expression is the false-set of the right expression.

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>();

trueSet.UnionWith(leftExpression.Symbol.TrueSet);

trueSet.UnionWith(rightExpression.Symbol.TrueSet);

Backpatch(leftExpression.Symbol.FalseSet, rightExpression.LongList);

Symbol symbol = new Symbol(trueSet, rightExpression.Symbol.FalseSet);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

return (new Expression(symbol, shortList, longList));

}

There is a set of optimizations available:

We clear goto-next-statements: jumps to the next line.

We also modify goto-next-double-statements: an conditional jump instruction that jumps two steps ahead and is followed by an unconditional jump instruction.

We trace goto-chains: jump instructions that jump to other jump instructions.

We clear unreachable code: code that is not reachable from the first function instruction.

We remove empty code: code that has been cleared by the optimization above.

If the deep search does not found such a solution, we have to do something about the graph. In this book we remove one edge and try the deep search again. In this way we continue to remove edges until we have found a solution. The benefit of removing edges is that the likeliness of founding a solution increases for each removed edge. The drawback is that each removed edge means that two tracks have to share we one register and that we have to add store and load instruction for the code to work. When we need to remove an edge we sort the edges by counting the number of load and store instructions necessary to add as compensation for the removal of the edge. There is no guaranty that this method generates the optimal solution but since shortage of registers only occur on rare occasions it shall be good enough.

In this way, every call to DeepFirstSearch, including the first call in RegisterAllocator.

If a track holds the pointer property, it can be allocated to a reduced set or registers only.

The Generate method is called after the tracks have been assigned registers. We iterate through the entries and assign them the given register. Note that the entries may have different sizes, why we have to convert the register to the size of each entry.

The only other cases when a temporary symbol is assigned is in case of a dereferred, index, or arrow expression. However, in those cases, the address symbol is not null.

The multiplication, division, and modulo operations demands that the left operands must be stored in a specific register before the operation. In this way, the operation itself takes only the right operands.

static char\* swShortDayList[] = {"Son", "Man", "Tis", "Ons",

"Tor", "Fre", "Lor"};

static char\* swLongDayList[] = {"Sondag", "Mandag", "Tisdag", "Onsdag",

"Torsdag", "Fredag", "Lordag"};

static char\* swShortMonthList[] = {"Jan", "Feb", "Mar", "Apr", "Maj", "Jun",

"Jul", "Aug", "Sep", "Okt", "Nov", "Dec"};

static char\* swLongMonthList[] = {"Januari", "Februari", "Mars", "April",

"Maj", "Juni", "Juli", "Augusit",

"September", "Oktober", "November", "December"};

static char\* swMessageList[] = {"inga fel", "felaktigt functionsnummer",

"hittar ej filen", "hittar ej sokvagen",

"inget handtag tillgangligt", "atkomst nekad",

"utanfor doman", "utanfor range",

"felaktig multibyte-sekvens",

"fel vid oppning", "fel vid flushing",

"fel vid stangning", "fel oppningslage",

"fel vid skrivning", "fel vid lasning",

"fel vid sokning", "fel vid telling",

"fel vid borttagning av fil",

"fel vid namnbyte av fil"};

static struct lconv sw\_EN\_utf8 = {1, 2, swShortDayList, swLongDayList,

enShortMonthList, swLongMonthList,

"abcdefghijklmnopqrstuvwxyz",

"ABCDEFGHIJKLMNOPQRSTUVWXYZ",

swMessageList};

static struct \_s {

char\* name;

struct lconv\* localePtr;

} sArray[] = {{"", &sw\_EN\_utf8}, {"C", &en\_US\_utf8}, {"US", &en\_US\_utf8}, {"SE", &sw\_EN\_utf8}};

static void memswp(void\* value1, void\* value2, int valueSize) {

char\* charValue1 = (char\*) value1;

char\* charValue2 = (char\*) value2;

int index;

for (index = 0; index < valueSize; ++index) {

char tempValue = charValue1[index];

charValue1[index] = charValue2[index];

charValue2[index] = tempValue;

}

}

* + 1. Slash Codes

The slashToChar method inspect the character succeeding the slash.

If the characters following the slash are a lowercase ‘x’ or an uppercase ‘X’ and one or two hexadecimal digits, or three octal digits XXX the following two characters are inspected, and their value calculated and the two hexadecimal digits are replaced three octal digits. However, if the two following characters are not hexadecimal digits, an error message occurs.

If the three, two, or one characters following the slash are octal digits, the value of the digits is calculated and the slash sequence is replaced by the character with the ASCII value of the octal digits. If the value of the digits exceeds 255, we report an error.

If the character following the slash is a lowercase ‘x’ or an uppercase ‘X’ and two or one hexadecimal digits, the value of the digits is calculated, and the slash sequence is replaced by the character with the ASCII value of the hexadecimal digits. A lowercase ‘x’ or an uppercase ‘X’ not followed by at least one hexadecimal digit results in an error being reported.

If none of the cases above applies, we report an error.

When the character or string has been translated from slash codes to regular characters, charToOctal is called to translate them to octal slash codes.

The octalToChar method is called by the scanner to tralform the octal shalch codes into regular characters.

If we encounter lines ending with two backslashes, we continue to append the lines to lineBuffer. Since an empty line was added at the end of the line list in generateLineList above, there is no risk that that the last line ends with a backslash.

When traversing the lines, we check whether the line starts with a sharp (‘#’). If it does, we consider the word following the sharp sign (#). If it is a preprocessor directive, we call the corresponding method. If the line does not start with a sharp and we are in a visible part of source code, we call searchForMacros, which expands macros. If we are not in a visible part of the source code, the line is replaced by an empty line.

while (index < trimList.Count) {

if (trimList[index].StartsWith("#")) {

StringBuilder buffer = new StringBuilder();

for (; (index < trimList.Count) && trimList[index].EndsWith("\\");

++index) {

buffer.Append(trimList[index].

Substring(0, trimList[index].Length - 1) + "\n");

}

if (index < trimList.Count) {

buffer.Append(trimList[index++] + "\n");

}

resultList.Add(buffer.ToString());

}

else {

StringBuilder buffer = new StringBuilder();

for (; (index < trimList.Count) && !trimList[index].StartsWith("#");

++index) {

buffer.Append(trimList[index] + "\n");

}

resultList.Add(buffer.ToString());

}

}

Morover, we also want to merge all other lines between two preprocessor directives into one line.

int index = 0;

List<string> resultList = new List<string>();

while (index < trimList.Count) {

If the line starts with a sharp (‘#’), we iterate through the trim list and add the lines to a buffer as long as they end with a slash (‘/’). We add a newline at the end of each line to keep the line count.

if (trimList[index].StartsWith("#")) {

StringBuilder buffer = new StringBuilder();

for (; (index < trimList.Count) && trimList[index].EndsWith("\\");

++index) {

buffer.Append(trimList[index].

Substring(0, trimList[index].Length - 1) + "\n");

}

In C, it is possible to end the last line in the source file with a slash. In that is not the case, we add the next line after the last line ended with a slash to the buffer.

if (index < trimList.Count) {

buffer.Append(trimList[index++] + "\n");

}

resultList.Add(buffer.ToString());

}

If the line does not start with a sharp, we instead iterate through the trim list and add the lines to a buffer as long as they do not start with a sharp. Also in this case, we add a newline at the end of each line to keep the line count.

else {

StringBuilder buffer = new StringBuilder();

for (; (index < trimList.Count) && !trimList[index].StartsWith("#");

++index) {

buffer.Append(trimList[index] + "\n");

}

resultList.Add(buffer.ToString());

}

}

return resultList;

}

private void AddNewlinesToBuffer(List<Token> tokenList) {

foreach (Token token in tokenList) {

m\_outputBuffer.Append(token.ToNewlineString());

CCompiler\_Main.Scanner.Line += token.GetNewlineCount();

}

}

The trimLeft and trimRight methods simple removes all white-spaces to the left or right of the string.

The countChar method simple counts the number of occurrences of the character.

. When the first character that is not a letter, digit, or an underline is reached, we check whether the line so far is a proper identifier. Then we have three cases:

1. We have reached the end of the line, in which case we have a macro with a name, but without parameters or body.

2. We have reached a space, in which case we have a macro with a name and a body, but no parameters.

3. We have reached a left parenthesis; in which case we have a list of parameters and we call DoParameterDefine.

4. If none of the above applies, we generate an error message.

The doParameterDefine method look into macros with parameters in two steps. First we extract the parameters by calling scanParameters. Then we go through the parameter list and check that the parameters are identifiers and that no parameters is repeated.

Then we extract the identifiers from the body and replace each occurrence of a parameters with the text “$parameter\_index$”.

Finally, we look for the merge operator ##. When we find one, we remove the operator and trim the text to its left and right.

It is allowed to redefine a macro if it is has the same parameters list and macro.

An identifier is a text starting with a letter or an underline and continuing with letters, digits, or underlines. If the given name is not an identifier, an error occurs.

The scanParameters extracts the actual parameters of a macro call. The parameters are separated by commas and the list is terminated by a right parenthesis. It becomes a bit complicated since the parameters themselves can hold parenthesis and commas. The field paranCount counts the level of parentheses nesting and only takes parameter into consideration when the nesting level is zero.

The lookupMacro method looks up a given macro. If it stored in MacroMap, the number of parameters is checked and then the parameter locations in the macro body are replaced by the actual parameters.

If the macro name is not stored in MacroMap, it may be one of the predefined special macros. The \_\_STDC\_\_ macro is replaced by the one integer value since the compiler of this book supports standard C.

The \_\_FILE\_\_ macro is replaced by the current path (Main.Path), the \_\_LINE\_\_ macro is replaced by the current line number (Main.Line).

The \_\_DATE\_\_ macro is replaced by the current date on the format “Jan 1, 1970” and the \_\_TIME\_\_ macro is replaced by the current time on the format “01:02:03”.

The addition and subtraction operators are a bit complicated, since it is possible to add and subtract pointers in C.

MiddleCodeGenerator.cs

public static Expression AdditionExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

Similar to the other cases above, the return the possible constant expression.

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

In the addition case, we also check if the expression is a static expression, in which case we return the expression.

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

if (staticExpression != null) {

return staticExpression;

}

The final short list is simple the short lists of the left and right expression.

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

If at least on the expressions has pointer or array type, we call the PointerArithmetic methods that handles pointer arithmetic.

if (leftType.IsPointerOrArray()) {

return PointerArithmetics(middleOp, leftExpression, rightExpression);

}

In the right expression has pointer type, the operator must be addition, not subtraction. Note that we swap the expressions in the call to PointerArithmetic, so that the pointer is the first expression. XXX

else if (rightType.IsPointerOrArray()) {

Assert.Error(middleOp == MiddleOperator.BinaryAdd, middleOp,

Message.Invalid\_types\_in\_subtraction\_expression);

return PointerArithmetics(middleOp, rightExpression, leftExpression);

}

If none of the left or right expressions are pointers, we perform an arithmetic operation. First, we make sure that both expressions have arithmetic types (integral or floating).

else {

Assert.Error(leftExpression.Symbol.Type.IsArithmetic(),

leftExpression, Message.Non\_\_arithmetic\_expression);

Assert.Error(rightExpression.Symbol.Type.IsArithmetic(),

rightExpression, Message.Non\_\_arithmetic\_expression);

The we find the maximal type and cast both expressions to the type. The resulting expression does also have the maximal type.

Type maxType = TypeCast.MaxType(leftType, rightType);

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

Symbol resultSymbol = new Symbol(maxType);

The final long list is the long lists of the two expressions, and the operation (addition or subtraction).

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

}

The PointerArithmetic methods performs pointer arithmetic. We have two cases: addition of a pointer and an integral value, and subtraction of two pointer values.

private static Expression PointerArithmetic(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.LongList);

shortList.AddRange(rightExpression.LongList);

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

If the left type is pointer and the right type is integral, we add an integer values to and pointer. What makes this different compared to regular addition is that we need take the size of the pointer type into consideration.

if (leftType.IsPointerOrArray() && rightType.IsIntegral()) {

We make sure that the left pointer type is not void, since pointer arithmetic is not allowed on void pointers.

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

We type cast the integral right expression to the left pointer type for the arithmetic to be performed on values of the same size.

rightExpression = TypeCast.ImplicitCast(rightExpression, leftType);

The final long list is the long lists of the left and right expression, and the pointer arithmetic.

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

The result symbol has the same type as the pointer expression.

Symbol resultSymbol =

new Symbol(new Type(leftType.PointerOrArrayType));

The pointer arithmetic is performed in two steps. First, we multiply the integral value with the pointer size, and then we add the product to the

int pointerSize = leftType.PointerOrArrayType.Size();

Symbol multSymbol = new Symbol(Type.PointerTypeX),

sizeSymbol = new Symbol(Type.PointerTypeX,

(BigInteger) pointerSize);

We multiply the integral value with the size of the pointer type and add the product to the pointer value. The multSymbol value be the result of two constant values, in case of a constant integral value, or the integral value times one, in case of a pointer size of one, which is inefficient. However, the Middle Code Optimizer in Chapter 11 will take care of those cases.

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply, multSymbol,

rightExpression.Symbol, sizeSymbol);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, multSymbol);

return (new Expression(resultSymbol, shortList, longList));

}

If it is not the case of a pointer and an integral value, then there must be two pointer values. We make sure that none of them pointes at void. Moreover, we also make sure that their pointer types have the same size.

else {

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

Assert.Error(!rightType.PointerOrArrayType.IsVoid(),

rightExpression, Message.Pointer\_to\_void);

Assert.Error(leftType.PointerOrArrayType.Size() ==

rightType.PointerOrArrayType.Size(),

leftType + " and " + rightType,

Message.Invalid\_expression);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol = new Symbol(Type.VoidPointerType);

int pointerSize = rightType.PointerOrArrayType.Size();

Symbol additionSymbol = new Symbol(Type.IntegerPointerType),

sizeSymbol = new Symbol(Type.IntegerPointerType,

(BigInteger) pointerSize);

We first subtract the pointer values, and then divide the difference by the size of the pointer types. There may be a division by one, in case the pointer types have size one, which is inefficient. However, the Middle Code Optimizer in Chapter 11 will take care of the case.

AddMiddleCode(longList, middleOp, additionSymbol,

leftExpression.Symbol, rightExpression.Symbol);

AddMiddleCode(longList, MiddleOperator.UnsignedDivide, resultSymbol,

additionSymbol, sizeSymbol);

Expression resultExpression = new Expression(resultSymbol,

shortList, longList);

Finally, we type cast the result of the division into signed integer.

return TypeCast.ImplicitCast(resultExpression,

Type.SignedIntegerType);

}

}

public static Expression AdditionExpressionX(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

if (middleOp == MiddleOperator.BinaryAdd) {

Assert.Error((leftType.IsArithmetic() && rightType.IsArithmetic()) ||

(leftType.IsPointerOrArray() && rightType.IsIntegral()) ||

(leftType.IsIntegral() && rightType.IsPointerOrArray()),

leftExpression, Message.Non\_\_arithmetic\_expression);

}

else {

Assert.Error((leftType.IsArithmetic() && rightType.IsArithmetic()) ||

(leftType.IsPointerOrArray() && rightType.IsIntegral()) ||

(leftType.IsPointerOrArray() && rightType.IsPointerOrArray() &&

(leftType.PointerOrArrayType.Size() == rightType.PointerOrArrayType.Size())),

leftExpression, Message.Non\_\_arithmetic\_expression);

}

/\*if (leftType.IsPointerOrArray() && rightType.IsPointerOrArray()) {

Assert.Error(((middleOp == MiddleOperator.BinaryAdd) &&

(leftType.PointerOrArrayType.Size() == 1) &&

(rightType.PointerOrArrayType.Size() == 1)) ||

((middleOp == MiddleOperator.BinarySubtract) &&

(leftType.PointerOrArrayType.Size() == rightType.PointerOrArrayType.Size())),

leftExpression, Message.Non\_\_arithmetic\_expression);

}

else {

Assert.Error((leftType.IsArithmetic() && rightType.IsArithmetic()) ||

(leftType.IsPointerOrArray() && rightType.IsIntegral()) ||

(leftType.IsIntegral() && rightType.IsPointerOrArray()),

leftExpression, Message.Non\_\_arithmetic\_expression);

}\*/

if (leftType.IsPointerOrArray()) {

Assert.Error(!leftType.PointerOrArrayType.IsVoid() &&

!leftType.PointerOrArrayType.IsFunction(),

leftExpression, Message.Non\_\_arithmetic\_expression);

}

if (rightType.IsPointerOrArray()) {

Assert.Error(!rightType.PointerOrArrayType.IsVoid() &&

!rightType.PointerOrArrayType.IsFunction(),

rightExpression, Message.Non\_\_arithmetic\_expression);

}

/\* Assert.Error(leftExpression.Symbol.Type.IsArithmetic(),

leftExpression, Message.Non\_\_arithmetic\_expression);

Assert.Error(rightExpression.Symbol.Type.IsArithmetic(),

rightExpression, Message.Non\_\_arithmetic\_expression);\*/

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

if (staticExpression != null) {

return staticExpression;

}

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

if (leftType.IsPointerOrArray() && rightType.IsIntegral()) {

int size = leftType.PointerOrArrayType.Size();

Symbol sizeSymbol = new Symbol(rightType, new BigInteger(size));

Expression sizeExpression = new Expression(sizeSymbol);

rightExpression = MultiplyExpression(MiddleOperator.UnsignedMultiply, rightExpression, sizeExpression);

}

else if (leftType.IsIntegral() && rightType.IsPointerOrArray()) {

int size = rightType.PointerOrArrayType.Size();

Symbol sizeSymbol = new Symbol(leftType, new BigInteger(size));

Expression sizeExpression = new Expression(sizeSymbol);

leftExpression = MultiplyExpression(MiddleOperator.UnsignedMultiply, leftExpression, sizeExpression);

}

/\*if (leftType.IsPointerOrArray()) {

return PointerArithmetic(middleOp, leftExpression, rightExpression);

}

else if (rightType.IsPointerOrArray()) {

Assert.Error(middleOp == MiddleOperator.BinaryAdd, middleOp,

Message.Invalid\_types\_in\_subtraction\_expression);

return PointerArithmetic(middleOp, rightExpression, leftExpression);

}

else {\*/

/\*Assert.Error(leftExpression.Symbol.Type.IsArithmetic(),

leftExpression, Message.Non\_\_arithmetic\_expression);

Assert.Error(rightExpression.Symbol.Type.IsArithmetic(),

rightExpression, Message.Non\_\_arithmetic\_expression);\*/

Type maxType = TypeCast.MaxType(leftType, rightType);

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

Symbol resultSymbol = new Symbol(maxType);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

Expression resultExpression = new Expression(resultSymbol, shortList, longList);

//return (new Expression(resultSymbol, shortList, longList));

//}

if ((middleOp == MiddleOperator.BinarySubtract) &&

leftType.IsPointerOrArray() && rightType.IsPointerOrArray()) {

if (leftType.PointerOrArrayType.Size() > 1) {

int size = leftType.PointerOrArrayType.Size();

Symbol sizeSymbol = new Symbol(leftType, new BigInteger(size));

Expression sizeExpression = new Expression(sizeSymbol);

resultExpression = MultiplyExpression(MiddleOperator.UnsignedDivide, resultExpression, sizeExpression);

}

resultExpression = TypeCast.ImplicitCast(resultExpression, Type.SignedIntegerType);

}

return resultExpression;

}

/\*private static Expression PointerArithmetic(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.LongList);

shortList.AddRange(rightExpression.LongList);

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

if (leftType.IsPointerOrArray() && rightType.IsIntegral()) {

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

rightExpression = TypeCast.ImplicitCast(rightExpression, leftType);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol =

new Symbol(new Type(leftType.PointerOrArrayType));

int pointerSize = leftType.PointerOrArrayType.Size();

Symbol multSymbol = new Symbol(Type.IntegerPointerType),

sizeSymbol = new Symbol(Type.IntegerPointerType,

(BigInteger) pointerSize);

//AddStaticSymbol(sizeSymbol);

//StaticSymbol sizeStaticSymbol = new StaticSymbol(sizeSymbol.UniqueName);

//SymbolTable.StaticSet.Add(sizeStaticSymbol);

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply, multSymbol,

rightExpression.Symbol, sizeSymbol);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, multSymbol);

return (new Expression(resultSymbol, shortList, longList));

}

else {

Assert.Error(!leftType.PointerOrArrayType.IsVoid(),

leftExpression, Message.Pointer\_to\_void);

Assert.Error(!rightType.PointerOrArrayType.IsVoid(),

rightExpression, Message.Pointer\_to\_void);

Assert.Error(leftType.PointerOrArrayType.Size() ==

rightType.PointerOrArrayType.Size(),

leftType + " and " + rightType,

Message.Invalid\_expression);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol = new Symbol(Type.VoidPointerType);

int pointerSize = rightType.PointerOrArrayType.Size();

Symbol additionSymbol = new Symbol(Type.IntegerPointerType),

sizeSymbol = new Symbol(Type.IntegerPointerType,

(BigInteger)pointerSize);

//StaticSymbol sizeStaticSymbol = new StaticSymbol(sizeSymbol.UniqueName);

//SymbolTable.StaticSet.Add(sizeStaticSymbol);

AddMiddleCode(longList, middleOp, additionSymbol,

leftExpression.Symbol, rightExpression.Symbol);

AddMiddleCode(longList, MiddleOperator.UnsignedDivide, resultSymbol,

additionSymbol, sizeSymbol);

Expression resultExpression = new Expression(resultSymbol,

shortList, longList);

return TypeCast.ImplicitCast(resultExpression,

Type.SignedIntegerType);

}

}\*/

/\* public static Expression SubtractionExpression(Expression leftExpression,

Expression rightExpression) {

Expression constantExpression =

ConstantExpression.Arithmetic(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

if (constantExpression != null) {

return constantExpression;

}

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

if (staticExpression != null) {

return staticExpression;

}

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

if (leftType.IsPointerOrArray()) {

return PointerArithmentics(MiddleOperator.BinarySubtract,

leftExpression, rightExpression);

}

else {

Assert.Error(leftType.IsArithmetic(), leftExpression,

Message.Invalid\_expression);

Assert.Error(rightType.IsArithmetic(), rightExpression,

Message.Invalid\_expression);

Type maxType = TypeCast.MaxType(leftType, rightType);

leftExpression = TypeCast.ImplicitCast(leftExpression, maxType);

rightExpression = TypeCast.ImplicitCast(rightExpression, maxType);

Symbol resultSymbol = new Symbol(maxType);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

AddMiddleCode(longList, MiddleOperator.BinarySubtract, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

}\*/

/\*if (integerExpression.Symbol.Value is BigInteger) {

int integerValue = (int)

((BigInteger) integerExpression.Symbol.Value);

int sizeValue = integerValue \* pointerType.PointerOrArrayType.Size();

Symbol sizeSymbol = new Symbol(Type.PointerTypeX,

(BigInteger) sizeValue);

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol,

pointerExpression.Symbol, sizeSymbol);

}

else if (pointerType.PointerOrArrayType.Size() == 1) {

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol,

pointerExpression.Symbol, integerExpression.Symbol);

}

else\*/

/\*if (leftType.IsPointerOrArray()) {

rightExpression = TypeCast.ImplicitCast(rightExpression, leftType);

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol = new Symbol(new Type(leftType.PointerOrArrayType));

if (rightExpression.Symbol.Value is BigInteger) {

int rightValue = (int) ((BigInteger) rightExpression.Symbol.Value);

Symbol sizeSymbol = new Symbol(Type.PointerTypeX, (BigInteger) (rightValue \* leftType.PointerOrArrayType.Size()));

sizeSymbol.StaticSymbol = ConstantExpression.Value(sizeSymbol.UniqueName, sizeSymbol.Type, sizeSymbol.Value);

SymbolTable.StaticSet.Add(sizeSymbol.StaticSymbol);

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, leftExpression.Symbol, sizeSymbol);

}

else if (leftType.PointerOrArrayType.Size() > 1) {

Symbol multSymbol = new Symbol(Type.PointerTypeX),

sizeSymbol = new Symbol(Type.PointerTypeX, (BigInteger) leftType.PointerOrArrayType.Size());

sizeSymbol.StaticSymbol = ConstantExpression.Value(sizeSymbol.UniqueName, sizeSymbol.Type, sizeSymbol.Value);

SymbolTable.StaticSet.Add(sizeSymbol.StaticSymbol);

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply, multSymbol, rightExpression.Symbol, sizeSymbol);

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, leftExpression.Symbol, multSymbol);

}

else {

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, leftExpression.Symbol, rightExpression.Symbol);

}

return (new Expression(resultSymbol, shortList, longList));

}

else if (rightType.IsPointerOrArray()) {

leftExpression = TypeCast.ImplicitCast(leftExpression, rightType);

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

Symbol resultSymbol;

if (leftExpression.Symbol.Value is BigInteger) {

int leftValue = (int) ((BigInteger) leftExpression.Symbol.Value);

Symbol sizeSymbol = new Symbol(Type.PointerTypeX, (BigInteger) (leftValue \* rightType.PointerOrArrayType.Size()));

sizeSymbol.StaticSymbol = ConstantExpression.Value(sizeSymbol.UniqueName, sizeSymbol.Type, sizeSymbol.Value);

SymbolTable.StaticSet.Add(sizeSymbol.StaticSymbol);

resultSymbol = new Symbol(new Type(rightType.PointerOrArrayType));

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, sizeSymbol, rightExpression.Symbol);

}

else if (rightType.PointerOrArrayType.Size() > 1) {

Symbol multSymbol = new Symbol(Type.PointerTypeX),

sizeSymbol = new Symbol(Type.PointerTypeX, (BigInteger) rightType.PointerOrArrayType.Size());

sizeSymbol.StaticSymbol = ConstantExpression.Value(sizeSymbol.UniqueName, sizeSymbol.Type, sizeSymbol.Value);

SymbolTable.StaticSet.Add(sizeSymbol.StaticSymbol);

AddMiddleCode(longList, MiddleOperator.UnsignedMultiply, multSymbol, leftExpression.Symbol, sizeSymbol);

resultSymbol = new Symbol(new Type(rightType.PointerOrArrayType));

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, multSymbol, rightExpression.Symbol);

}

else {

resultSymbol = new Symbol(new Type(rightType.PointerOrArrayType));

AddMiddleCode(longList, MiddleOperator.BinaryAdd, resultSymbol, leftExpression.Symbol, rightExpression.Symbol);

}

return (new Expression(resultSymbol, shortList, longList));

}\*/

/\*if (rightExpression.Symbol.Value is BigInteger) {

int rightValue = (int) ((BigInteger) rightExpression.Symbol.Value);

int sizeValue = rightValue \* leftType.PointerOrArrayType.Size();

Symbol sizeSymbol = new Symbol(Type.PointerTypeX, (BigInteger) sizeValue);

AddMiddleCode(longList, MiddleOperator.BinarySubtract, resultSymbol,

leftExpression.Symbol, sizeSymbol);

}

else if (leftType.PointerOrArrayType.Size() == 1) {

AddMiddleCode(longList, MiddleOperator.BinarySubtract, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

}

else\*/

public static Expression IndexExpression(Expression leftExpression,

Expression rightExpression) {

Type leftType = leftExpression.Symbol.Type,

rightType = rightExpression.Symbol.Type;

Assert.Error((leftType.IsPointerArrayOrString() && rightType.IsIntegral()) ||

(leftType.IsIntegral() && rightType.IsPointerArrayOrString()),

null, Message.Invalid\_type\_in\_index\_expression);

Expression staticExpression =

StaticExpression.Binary(MiddleOperator.Index, leftExpression,

rightExpression);

if (staticExpression != null) {

return staticExpression;

}

Expression arrayExpression, indexExpression;

if (leftExpression.Symbol.Type.IsPointerOrArray()) {

arrayExpression = leftExpression;

indexExpression = rightExpression;

}

else {

indexExpression = leftExpression;

arrayExpression = rightExpression;

}

Type arrayType = arrayExpression.Symbol.Type,

indexType = indexExpression.Symbol.Type;

if (indexExpression.Symbol.Value is BigInteger) {

int indexValue = (int) ((BigInteger) indexExpression.Symbol.Value),

indexSize = arrayType.PointerOrArrayType.Size();

Symbol resultSymbol = new Symbol(arrayType.PointerOrArrayType);

return Dereference(arrayExpression, resultSymbol,

indexValue \* indexSize);

}

else {

int size = arrayType.PointerOrArrayType.Size();

if (size > 1) {

Symbol sizeSymbol = new Symbol(indexType, new BigInteger(size));

Expression sizeExpression = new Expression(sizeSymbol);

indexExpression = MultiplyExpression(MiddleOperator.UnsignedMultiply,

indexExpression, sizeExpression);

}

indexExpression =

TypeCast.ImplicitCast(indexExpression, arrayType);

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(arrayExpression.ShortList);

shortList.AddRange(indexExpression.ShortList);

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(arrayExpression.LongList);

longList.AddRange(indexExpression.LongList);

Symbol addSymbol = new Symbol(arrayType);

AddMiddleCode(longList, MiddleOperator.BinaryAdd,

addSymbol, arrayExpression.Symbol, indexExpression.Symbol);

Expression addExpression =

new Expression(addSymbol, shortList, longList);

return Dereference(addExpression, resultSymbol, 0);

}

}

Addition and subtraction, increment, and decrement operations perform the same task with signed and unsigned values, while the multiplication, division, and modulo operations need to take into consideration whether the values are signed or unsigned.

The GenerateFunctionCode is called after the middle code of the function has been generated. Its task is to generate the assembly and object code of the function. If the function returns void and is the main function, the Exit middle code is added to middle code list. If it returns void and is not the main, the Return middle code instruction is added.

Then the middle code is being optimized and a list of base blocks is generated.

The object code is actually generated twice, since we want to know which symbols are not used and can be removed from the symbol table.

If the function is the main function we have to modify the entry point of the function.

Finally, the function is added to the linker map and the symbol table is popped.

The generateAssemblyText method generates the assembly code for the function.

When a function containing goto statements has been parsed, the addresses need to be set with the help of the label map.

A theoretical explanation of the open-closed statement solution is beyond the scope of this book, but I recommend the Dragon Book by Aho et al. for a closer look.

To keep track of the label statements we have the m\_labelMap map with the label name as key and the first middle code instruction of the following statement as value.

An expression statement is an optional expression followed by a semicolon. The expression is evaluated and its true and false-sets are backpatched to the beginning of the next statement. Note that we do not use the result of the expression, the only interested part is its potential side effects. If it has no side effects, the statements will be removed by the middle code optimizer in Chapter 0.

A statement can be an optional sequence of statements enclosed in brackets. The sequence is parsed with a new symbol table, since two symbols with the same name can be defined in different blocks.

The address operator takes the address of its operand and the return type is a pointer to the operand type, unless it is an array, in which case the result is a pointer to the array type. The operand cannot have register storage.

A symbol is addressable if it does not have register storage and is not a bitfield in a struct or a union.

When generating middle code, there are numerous occasions that we generate forward jump instructions without knowing which instruction to jump to. In those cases, we store the instructions in sets and later go back and fill in the jump address. The process is called backpatching.

The static Backpatch methods backpatches a single address or a set of addresses. The first two methods call the last two methods with the current size of the middle code list as target address. In this way, the target address becomes the next middle code instruction to become generated. The backpatching is performed by setting the third address of the middle code, since the third address is always the jump address in conditional and unconditional jump instructions.

When declared, a function may be extern or static. If it is static, it does not have external linkage. So after the external linkage has been set above, we do not need to storage of a function declarator. Therefore, we always set extern storage for a function. If this function declaration should in fact be a function definition, with a function body, then its storage will be set to static by the FunctionHeader method in the GenerateMiddleCode class.

if (type.IsFunction()) {

storage = Storage.Extern;

}

If the storage specifier is omitted, a variable becomes static in global scope and auto in a function while a function becomes extern, unless it has a body in which case is becomes static. The register storage has been included for the sake of completeness. The only difference between an auto or register symbol is that the address operator cannot be applied to a register symbol.

C has a rather large set of types. The simple types are made up of the integral types signed and unsigned char, short int, int, and long int as well as the floating types float, double, and long double. The compound types are pointers, arrays, structs, unions, and functions. Values of enumeration types (enum) are stored as signed integer. Moreover, a type can also be constant or volatile. Internally, we introduce the logical and string types, even though there are no such types in C. We also have the void type, which technically is not a type, but rather mark the absence of a type.

The Sort enumeration holds the simple and compound types of C. Note that String and Logical are present, even though they are not types in C. However, temporary vale may hold these types.

The MiddleCodeOptimization class takes care of the optimization.

The swap map is used in OptimizeRelation below. The reason we define it at this point in the code is that we cannot have more than one static area in the same class.

public static Expression Relation(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

If at least one of the expressions is not constant, we just return null.

if (!IsConstant(leftExpression) || !IsConstant(rightExpression)) {

return null;

}

If at least one of the expressions have floating type, we call the RelationFloating method. If not at least one of the expressions have floating type, both expressions must have integral type and we call RelationIntegral instead.

else if (leftExpression.Symbol.Type.IsFloating() ||

rightExpression.Symbol.Type.IsFloating()) {

return RelationFloating(middleOp, leftExpression, rightExpression);

}

else {

return RelationIntegral(middleOp, leftExpression, rightExpression);

}

}

The RelationIntegral method evaluates the constant value of a relation expression. In order for this method to be called, both operands must be constant.

private static Expression RelationIntegral(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

First, we need to cast the expression from a possible logical type into the integral type: signed integer.

leftExpression = TypeCast.LogicalToIntegral(leftExpression);

rightExpression = TypeCast.LogicalToIntegral(rightExpression);

This method is called only if both expressions have values. Therefore, we do not have to check that the values are not null.

BigInteger leftValue = (BigInteger) leftExpression.Symbol.Value,

rightValue = (BigInteger) rightExpression.Symbol.Value;

Since the expression has a constant value, we only add one unconditional jump instruction.

List<MiddleCode> longList = new List<MiddleCode>();

MiddleCode jumpCode = new MiddleCode(MiddleOperator.Jump);

longList.Add(jumpCode);

ISet<MiddleCode> jumpSet = new HashSet<MiddleCode>();

jumpSet.Add(jumpCode);

Then we need to evaluate the value of the expression. Since the values are object of the BigInteger class we can use the regular relation operators.

bool resultValue = false;

switch (middleOp) {

case MiddleOperator.Equal:

resultValue = (leftValue == rightValue);

break;

case MiddleOperator.NotEqual:

resultValue = (leftValue != rightValue);

break;

case MiddleOperator.LessThan:

resultValue = (leftValue < rightValue);

break;

case MiddleOperator.LessThanEqual:

resultValue = (leftValue <= rightValue);

break;

case MiddleOperator.GreaterThan:

resultValue = (leftValue > rightValue);

break;

case MiddleOperator.GreaterThanEqual:

resultValue = (leftValue >= rightValue);

break;

}

Finally, we define the logical symbol with the jump set, depending on the value of the expression. If the value is true the jump set becomes the true-set of the symbol, and if the value is false the jump set becomes the false-set of the symbol.

Symbol resultSymbol = resultValue ? (new Symbol(jumpSet, null))

: (new Symbol(null, jumpSet));

The resulting expression has no short list, and its long list is made up be one instruction only: the unconditional jump.

return (new Expression(resultSymbol, null, longList));

}

The RelationFloating method makes sure the expression holds integral type. If it holds logical type, it is cast from logical type to signed integer type.

private static Expression RelationFloating(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

First, we need to cast the potential logical or integral expressions to floating type (double).

leftExpression = TypeCast.LogicalToFloating(leftExpression);

rightExpression = TypeCast.LogicalToFloating(leftExpression);

Then we cast left value and right from BigInteger to decimal, if necessary.

decimal leftValue;

if (leftExpression.Symbol.Value is BigInteger) {

leftValue = (decimal) ((BigInteger) leftExpression.Symbol.Value);

}

else {

leftValue = (decimal) leftExpression.Symbol.Value;

}

decimal rightValue;

if (rightExpression.Symbol.Value is BigInteger) {

rightValue = (decimal) ((BigInteger) rightExpression.Symbol.Value);

}

else {

rightValue = (decimal) rightExpression.Symbol.Value;

}

We add an unconditional jump instruction to the long list and jump set. The jump set will take the place of true-set of false-set, depending on whether the value is true.

List<MiddleCode> longList = new List<MiddleCode>();

MiddleCode jumpCode = new MiddleCode(MiddleOperator.Jump);

longList.Add(jumpCode);

ISet<MiddleCode> jumpSet = new HashSet<MiddleCode>();

jumpSet.Add(jumpCode);

Since the values are object of the decimal class, we can use the regular relation operators to evaluate the value.

bool resultValue = false;

switch (middleOp) {

case MiddleOperator.Equal:

resultValue = (leftValue == rightValue);

break;

case MiddleOperator.NotEqual:

resultValue = (leftValue != rightValue);

break;

case MiddleOperator.LessThan:

resultValue = (leftValue < rightValue);

break;

case MiddleOperator.LessThanEqual:

resultValue = (leftValue <= rightValue);

break;

case MiddleOperator.GreaterThan:

resultValue = (leftValue > rightValue);

break;

case MiddleOperator.GreaterThanEqual:

resultValue = (leftValue >= rightValue);

break;

}

The resulting symbol has a logical value. The jump set becomes its true-set or false-set, depending on whether the result value is true.

Symbol resultSymbol = resultValue ? (new Symbol(jumpSet, null))

: (new Symbol(null, jumpSet));

return (new Expression(resultSymbol, null, longList));

}

When we have evaluated the value, we must decide its type, which we do by calling the MaxType method in the TypeCast class.

Type maxType = TypeCast.MaxType(leftExpression.Symbol.Type,

rightExpression.Symbol.Type);

The final symbol is a symbol with the maximum type and resulting value.

Symbol resultSymbol = new Symbol(maxType, resultValue);

In the floating evaluation case, we need to push the resulting value to the floating value stack.

List<MiddleCode> longList = new List<MiddleCode>();

MiddleCodeGenerator.AddMiddleCode(longList, MiddleOperator.PushFloat,

resultSymbol);

The final expression has no short list, and the long list hold the stack pushing instruction only.

return (new Expression(resultSymbol, null, longList));

}

If the source type is integral, array, pointer, or floating, and the target type is logical, we start by constructing a jump set holding one unconditional jump instruction.

If the value of the source symbol is zero, we create a target symbol with jump set as its false-set. If it not zero, we create the target symbol with the jump set as its true-set.

The long list holds the unconditional jump instruction of the jump set.

else if (sourceType.IsArithmeticPointerArrayStringOrFunction() &&

targetType.IsLogical()) {

bool isTrue = !sourceValue.Equals(BigInteger.Zero) &&

!sourceValue.Equals(decimal.Zero);

MiddleCode gotoCode = new MiddleCode(MiddleOperator.Jump);

longList.Add(gotoCode);

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),

falseSet = new HashSet<MiddleCode>();

if (isTrue) {

trueSet.Add(gotoCode);

}

else {

falseSet.Add(gotoCode);

}

targetSymbol = new Symbol(trueSet, falseSet);

}

else {

targetSymbol = new Symbol(targetType, sourceValue);

}

if (targetType.IsFloating()) {

longList.Add(new MiddleCode(MiddleOperator.PushFloat, targetSymbol));

}

return (new Expression(targetSymbol, null, longList));

}

Finally, we have reached the point where both the source type and target type are integral, array, pointer, or floating. The only thing that remains to be done is to make sure the final value is an object of the correct class: BigInteger or decimal.

If the source type is integral, array, or pointer, and the target type is floating, we need to type cast the value from BigInteger to decimal.

if (sourceType.IsIntegralPointerArrayStringOrFunction () &&

targetType.IsFloating()) {

targetSymbol =

new Symbol(targetType, (decimal) ((BigInteger) sourceValue));

}

In the same way, if the source type is floating and the target type is integral, array, or pointer, we need to type cast the value from decimal to BigInteger.

else if (sourceType.IsFloating() &&

targetType.IsIntegralPointerOrArray()) {

targetSymbol =

new Symbol(targetType, (BigInteger) ((decimal) sourceValue));

}

Finally, if the both the source and target types are integral, array, or pointer, or if they both are floating, we just create a new symbol with target type and source value.

else {

targetSymbol = new Symbol(targetType, sourceValue);

}

List<MiddleCode> longList = new List<MiddleCode>();

if (targetType.IsFloating()) {

MiddleCodeGenerator.

AddMiddleCode(longList, MiddleOperator.PushFloat, targetSymbol);

}

return (new Expression(targetSymbol, null, longList));

}

}

/\*case MiddleOperator.Dereference:

if (symbol.Value is StaticAddress) { // \*&a[i], \*&s.i

StaticAddress staticAddress = (StaticAddress) symbol.Value;

StaticValue staticValue =

new StaticValue(staticAddress.UniqueName, staticAddress.Offset);

return (new Symbol(new Type(symbol.Type), staticValue));

}

else if (symbol.IsExternOrStatic()) { // \*&i

StaticValue staticValue = new StaticValue(symbol.UniqueName, 0);

return (new Symbol(new Type(symbol.Type), staticValue));

}

break;

}\*/

The LinkerInfo class holds the information of a function definition or a static variable:

* m\_fullName: the full name of the function or static variable. The full name of a function or global variable with external linkage is just its regular name, while the full name of
* m\_entryPoint: the function call entry point, which may differ from the start address of the function. Actually, a function’s start address and its the entry point differs in one case only: when main is defined with the argc and argv parameters and is called recursively, which is allowed in the C standard. In that case, the first part of the main function holds code for initializing argc and argv, which shall not be executed in a recursive call.
* m\_accessMap: the map of all accesses of function or static variables.
* m\_callMap: the map of all function calls. The reason we distinguish between m\_accessMap and m\_callMap is that accesses are absolute while function calls are relative.
* m\_returnSet: the set of all return address assignments.
* m\_byteList: the actual code of the function or initialization data of the static variable. If the static variable is uninitialized, all data is zero.

To be more exact, its tasks is to

* To resolve access to static symbols
* To resolve function calls.

To resolve return addresses.

There are several maps:

* m\_globalMap: holds all functions and static variables of all object files.
* m\_globalList: holds all functions and static variables reachable from main, where main is always placed first.
* m\_addressMap holds the beginning of each function or static variables in the final code.
* m\_entryMap holds the entry point of each function. As mention in Section 13.2, a function’s start address and its the entry point differs only when main is defined with the argc and argv parameters and is called recursively.

If an expression is not constant, the next step is to decide whether it is static.

There are several cases to consider:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Initializer** | **Example** |
| Pointer | Address | static int array[3];  static int\* p = &a[2]; |
| Pointer to signed or unsigned char | String | static char \*p = “Hello”; |
| Pointer | List | int \*p = {1,2,3}; |
| Array | String | char s[] = “World”; |
| Array | List | Char st[] = {‘a’, ’b’, ‘c’}; |
| Struct | List |  |
| Union | List |  |
| Integral or Pointer |  |  |
| Floating |  |  |

If the variable is a pointer and the initializer is an address we add its offset to the block and store the name of the address in the access map, the address value will later be looked up and added by the linker.

If the variable type is a pointer to a (signed or unsigned) character and the initializator is a string, the address is stored in the access map and a zero address is stored in the block, it will later be properly looked up and set by the linker.

if (fromType.Equals(toType)) {

return fromExpression;

}

else if (fromType.IsFloating() && toType.IsFloating() &&

(fromType.Size() == toType.Size())) {

return fromExpression;

}

else if (fromType.IsIntegralPointerArrayStringOrFunction() &&

toType.IsIntegralPointerArrayStringOrFunction() &&

(fromType.SizeArray() == toType.SizeArray())) {

return fromExpression;

}

else {

return ExplicitCast(fromExpression, toType);

}

if (leftType.IsPointerOrArray() && rightType.IsPointerOrArray()) {

if (leftType.PointerOrArrayType.Size() > 1) {

int size = leftType.PointerOrArrayType.Size();

Symbol sizeSymbol =

new Symbol(resultExpression.Symbol.Type, new BigInteger(size));

Expression sizeExpression = new Expression(sizeSymbol);

resultExpression = MultiplyExpression(MiddleOperator.Divide,

resultExpression, sizeExpression, false);

}

return TypeCast.ImplicitCast(resultExpression, Type.SignedIntegerType);

}

if (fromType.Equals(toType) ||

(fromType.IsLogical() && toType.IsLogical()) ||

(fromType.IsPointerArrayStringOrFunction() &&

toType.IsPointerOrArray()) ||

(fromType.IsPointerArrayStringOrFunction() &&

toType.IsIntegral() && (fromType.Size() == toType.Size())) ||

(((fromType.IsFloating() && toType.IsFloating()) ||

(fromType.IsIntegralPointerOrFunction() &&

toType.IsIntegralPointerArrayOrFunction())) &&

(fromType.SizeArray() == toType.SizeArray()))) {

return fromExpression;

}\*/

public void AddTag(string name, Type newType) {

if (m\_tagMap.ContainsKey(name)) {

Type oldType = m\_tagMap[name];

Assert.Error(!oldType.IsEnumerator() &&

(oldType.Sort == newType.Sort), name,

Message.Name\_already\_defined);

If the member map of the old tag is null, we assign it the member map of the new tag.

if (oldType.MemberMap == null) {

oldType.MemberMap = newType.MemberMap;

oldType.MemberList = newType.MemberList;

}

If the neither the old nor the new member map is null, we report an error since it is not allowed to have two structs or unions with the same name tags with non-null member maps.

else {

Assert.Error(newType.MemberMap == null, name,

Message.Name\_already\_defined);

}

}

If there is no struct or union tag previous added to the tag map, we simply add the new tag to the map.

else {

m\_tagMap.Add(name, newType);

}

}

Before the parsing of the members we assign them a symbol table of their own.

SymbolTable.CurrentTable =

new SymbolTable(SymbolTable.CurrentTable, (Scope) $1);

}

LEFT\_BLOCK declaration\_list RIGHT\_BLOCK {

$$ = MiddleCodeGenerator.StructUnionSpecifier($2, $1);

After the parsing of the member list of the struct, we restore the original symbol table.

SymbolTable.CurrentTable = SymbolTable.CurrentTable.ParentTable;

}

In case of a struct of union without a declaration list, but with an obligatory name, we look up the name.

| struct\_or\_union NAME {

$$ = MiddleCodeGenerator.LookupStructUnionSpecifier($2, $1);

};

public static Type StructUnionSpecifier(string optionalName, Sort sort) {

if (optionalName != null) {

Type type = SymbolTable.CurrentTable.LookupTag(optionalName, sort);

type.MemberMap = SymbolTable.CurrentTable.EntryMap;

type.MemberList = SymbolTable.CurrentTable.EntryList;

return type;

}

If the optional name is null, we create and return a type with the entry map of the current symbol table.

else {

return (new Type(sort, SymbolTable.CurrentTable.EntryMap,

SymbolTable.CurrentTable.EntryList));

}

}

public static Type LookupStructUnion(string name, Sort sort) {

Type type = SymbolTable.CurrentTable.LookupTag(name, sort);

If the struct or union exists, we simply return its type.

if (type != null) {

return type;

}

If the struct or union does not exist, we create a new type and add it to the tag map. However, the type lacks a member map, which means that it is not yet possible to define variables of the type. It is only when the struct or union becomes properly defined with a member map that it will be possible to define variables.

else {

type = new Type(sort);

SymbolTable.CurrentTable.AddTag(name, type);

return type;

}

}

### Arrow Expression

The ArrowExpression method does also call Dereference after checking that the expression type and member name is valid.

### Index Expression

In an index expression one of the expressions shall be a pointer or an array, while the other expression is an integral value. For instance, if a is an array and i an integer value, both a[i] and i[a] are valid index expressions.

On of the types must be a non-void pointer, array, or string, and the other type must be integral.

Similar to the cases above we check for static expression. However, an index expression cannot be constant.

Note that either of the left and right expression may be the array or index expression. Therefore, we check which expression is a pointer or an array.

If the index value is constant, we can call Dereference with the index value multiplied with the size of the pointer or array type as offset.

If the index value is not constant, we begin by generating code for multiplying the index expression with the size of the pointer or array type.

r for the The increment and decrement operator apply not only to integral values, but also to floating values. The operation. We start by pushing the value one at the floating value stack, the value to be incremented or decremented has already been pushed at the stack by earlier operations. We perform the operation, which is addition or subtraction. We preform the operation on both the short list and list of the expression. The difference is that in the short list case we pop the value off the stack since we do not need it anymore. In the long list case we do nothing, we just let the value stay on the stack to be used by later operations.

### The Condition Expression

The condition operator is rather complicated.

public static Expression ConditionalExpression(Expression testExpression,

Expression trueExpression,

Expression falseExpression) {

We type cast the test expression to logical type.

testExpression = TypeCast.ToLogical(testExpression);

If the test expression is constant, we simple return the true expression if the test expression is true and the false expression if it is false.

if (ConstantExpression.IsConstant(testExpression)) {

return ConstantExpression.IsTrue(testExpression)

? falseExpression : trueExpression;

}

If both the true and false expressions hold logical types, we keep their types.

if (trueExpression.Symbol.Type.IsLogical() &&

falseExpression.Symbol.Type.IsLogical()) {

We start by backpatching the true-set and false-set of the test expression to the beginning of the true and false expressions’ code list.

Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);

Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);

The resulting true-set is the union of the true-sets of the true and false expression, and the resulting false-set is the union of the false-sets of the true and false expression.

ISet<MiddleCode> trueSet = new HashSet<MiddleCode>(),

falseSet = new HashSet<MiddleCode>();

trueSet.UnionWith(trueExpression.Symbol.TrueSet);

trueSet.UnionWith(falseExpression.Symbol.TrueSet);

falseSet.UnionWith(trueExpression.Symbol.FalseSet);

falseSet.UnionWith(falseExpression.Symbol.FalseSet);

List<MiddleCode> shortList = new List<MiddleCode>();

If the short lists of both the true and false expression is empty, it does not matter if the test expression is true or false, and we let the resulting short list be the short list of the true expression (which may be empty).

if (IsEmpty(trueExpression.ShortList) &&

IsEmpty(falseExpression.ShortList)) {

shortList.AddRange(testExpression.ShortList);

}

If the short list of the test expression is not empty, the situation becomes a bit more complicated. We add the long list, rather than the short list, of the test expression to the final short list since we need the value of the test expression in order to jump to the beginning of the short list of either the true of false expression.

else {

shortList.AddRange(testExpression.LongList);

shortList.AddRange(trueExpression.ShortList);

shortList.AddRange(falseExpression.ShortList);

}

We add the long list of the test, true, and false expression to the final long list.

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(testExpression.LongList);

longList.AddRange(trueExpression.LongList);

longList.AddRange(falseExpression.LongList);

Finally, we create a new symbol with logical type and the resulting true-sets and false-sets.

Symbol symbol = new Symbol(trueSet, falseSet);

return (new Expression(symbol, shortList, longList));

}

If at least one of the true or false expression does not hold logical type, we define maxType as their largest type, and type cast both expressions to that type.

else {

Type maxType = TypeCast.MaxType(trueExpression.Symbol.Type,

falseExpression.Symbol.Type);

trueExpression = TypeCast.ImplicitCast(trueExpression, maxType);

Backpatch(testExpression.Symbol.TrueSet, trueExpression.LongList);

We create a new temporary symbol to hold the result of the condition expression.

Symbol symbol = new Symbol(maxType);

In case of non-floating type, we add the assignment instruction. In case of a floating type, the value is already placed at the floating value stack and we do not need to do anything.

if (maxType.IsFloating()) {

AddMiddleCode(trueExpression.LongList,

MiddleOperator.DecreaseStack);

}

else { // XXX

if (trueExpression.Symbol.IsTemporary()) {

foreach (MiddleCode middleCode in trueExpression.LongList) {

if (middleCode[0] == trueExpression.Symbol) {

middleCode[0] = symbol;

}

}

}

else {

AddMiddleCode(trueExpression.LongList, MiddleOperator.Assign,

symbol, trueExpression.Symbol);

}

}

We add the jump to a target code to both the short and long list of the true expression. The target code will be added after the code of the false expression. Its purpose is to jump over the false expression code.

MiddleCode targetCode = new MiddleCode(MiddleOperator.Empty);

AddMiddleCode(trueExpression.ShortList,

MiddleOperator.Jump, targetCode);

AddMiddleCode(trueExpression.LongList,

MiddleOperator.Jump, targetCode);

Similar to the true expression, we type cast and false expression, and backpatch the true-set of the test expression to the beginning of the false expression code.

falseExpression = TypeCast.ImplicitCast(falseExpression, maxType);

Backpatch(testExpression.Symbol.FalseSet, falseExpression.LongList);

We also assign the value of the false expression to the temporary symbol if it does not hold floating type. If it holds floating type, the value is already placed at the top of the floating value stack, and we do nothing.

if (!maxType.IsFloating()) {

if (falseExpression.Symbol.IsTemporary()) {

foreach (MiddleCode middleCode in falseExpression.LongList) {

if (middleCode[0] == falseExpression.Symbol) {

middleCode[0] = symbol;

}

}

}

else {

AddMiddleCode(falseExpression.LongList, MiddleOperator.Assign,

symbol, falseExpression.Symbol);

}

}

If both the short lists of the true and false expressions are empty, we just add the short list of the test expression (which may be empty).

List<MiddleCode> shortList = new List<MiddleCode>();

if (IsEmpty(trueExpression.ShortList) &&

IsEmpty(falseExpression.ShortList)) {

shortList.AddRange(testExpression.ShortList);

}

If not both the short lists of the true and false expressions are empty, we add the long list of the test expression, rather than the short list, since we need to value of the test expression, as well as the short lists of the true and false expressions.

else {

shortList.AddRange(testExpression.LongList);

shortList.AddRange(trueExpression.ShortList);

shortList.AddRange(falseExpression.ShortList);

shortList.Add(targetCode);

}

Finally, add the long lists and return the expression.

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(testExpression.LongList);

longList.AddRange(trueExpression.LongList);

longList.AddRange(falseExpression.LongList);

longList.Add(targetCode);

return (new Expression(symbol, shortList, longList));

}

}

### Shift Expression

In C, there are both left shift and right shift.

public static Expression ShiftExpression(MiddleOperator middleOp,

Expression leftExpression,

Expression rightExpression) {

First, we check that the left expression holds integral or pointer type. An array, a string, or a function is cast to a pointer.

Assert.Error(leftExpression.Symbol.Type.

IsIntegralPointerArrayStringOrFunction(),

leftExpression, Message.Invalid\_type\_in\_shift\_expression);

If the expression can be evaluated to a constant expression, we return the constant expression.

Expression constantExpression =

ConstantExpression.Arithmetic(middleOp, leftExpression,

rightExpression);

if (constantExpression != null) {

return constantExpression;

}

The right expression is type cast to an unsigned character, which always have a size of one byte.

rightExpression =

TypeCast.ImplicitCast(rightExpression, Type.UnsignedCharType);

The final short list is simple the short lists of the left and right expression.

List<MiddleCode> shortList = new List<MiddleCode>();

shortList.AddRange(leftExpression.ShortList);

shortList.AddRange(rightExpression.ShortList);

The final short list is simple the short lists of the left and right expression.

List<MiddleCode> longList = new List<MiddleCode>();

longList.AddRange(leftExpression.LongList);

longList.AddRange(rightExpression.LongList);

The final long list is the short lists of the left and right expression, and the shift operation.

Symbol resultSymbol = new Symbol(leftExpression.Symbol.Type);

AddMiddleCode(longList, middleOp, resultSymbol,

leftExpression.Symbol, rightExpression.Symbol);

return (new Expression(resultSymbol, shortList, longList));

}

/\*if (leftExpression.Symbol.Type.IsSigned() &&

rightExpression.Symbol.Type.IsUnsigned()) {

rightExpression.Symbol.Type = leftExpression.Symbol.Type;

}

else if (leftExpression.Symbol.Type.IsUnsigned() &&

rightExpression.Symbol.Type.IsSigned()) {

leftExpression.Symbol.Type = rightExpression.Symbol.Type;

}\*/

If one of the expression types is signed and the other is unsigned, we make sure they are both signed.

if (leftExpression.Symbol.Type.IsSigned() &&

rightExpression.Symbol.Type.IsUnsigned()) {

rightExpression =

TypeCast.ImplicitCast(rightExpression, leftExpression.Symbol.Type);

}

else if (leftExpression.Symbol.Type.IsUnsigned() &&

rightExpression.Symbol.Type.IsSigned()) {

leftExpression =

TypeCast.ImplicitCast(leftExpression, rightExpression.Symbol.Type);

}

The registers are only used internally, in conjunction with system calls. On some occasion the interrupt call returns information stored in a register.

The closed statements do also include a set of internal statements; that is, statements not included in standard C, but necessary for the standard library functionality. The first statement is the load register statement, which stores a value in a register. The operands are the name of a register and an expression of integral or pointer type with same size as the register.

Every function definition, as well as static variables, are added to the global static set. They will eventually be translated into assembly code.

If the number of bits is less then the type size in bits, we need to define the bit mask that is used to delete the bits not covered by the mask.

Contrary to the previous cases, we do not add the symbol to the global static set since it cannot be static.

If the declarator is null (it has been omitted in the code) we just check that the number of bits is in range: at least one and at most the type size in bits.

When generating the final assembly code, we cannot have an integer value as the left expression in a relational expression. Therefore, we swap the operands if the left operand holds an integer value. The expression cannot hold two integer values, in that case the ConstantExpression class of Chapter 8 would have reduced the expression to its resulting value.

Moreover, if the left expression is an array, function, or string, we want to use its address rather than its value. Similar to the integer value case, we cannot use the address directly in the assembly code. Therefore, if the left expression is an array, function or string, and the right expression is not an array, function, or string, or an integer value, we also swap the expressions.