LLVM Project

Part 2

Overview

This report serves as a description of the second part of our LLVM project, which entails the design and implementation of a dataflow analysis framework, which we make use of in order to implement certain analyses and optimizations. We begin by implementing an intra-procedural dataflow analysis framework which takes a lattice, flow functions and a representation of the program to be analyzed as input and yields as output an element of the lattice for the beginning of each basic block and each instruction of the program. We subsequently run the following analyses on the program: constant propagation, available expressions, range analysis and intra-procedural pointer analysis.

To demonstrate the flow of our analysis, we separate control flow graphs by function, performing an intraprocedural analysis. Within each function we travel through the control flow graph by the order of blocks provided by LLVM. We observe that for each block, all predecessors are already traversed, which is useful to deal with dataflow merges. However, this is not the case if there is a loop, which implies back edges. To implement loops where the control flow revisits blocks, we utilize recursion in order to handle cases where there are loops nested within loops. This looping behavior will terminate and resume normal control flow once there is a fixed dataflow fact.

Our analysis prints out the following recorded data for a current block:

- Predecessor block references and corresponding incoming data flow facts at the start of a block
- Instructions with incoming and outgoing dataflow facts (useful for verifying dataflow functions and future program transformations). For conditional branch instructions for relevant analyses, it will output true and false versions of outgoing dataflow facts
- Successor block references. When we traverse those blocks, those blocks will grab the current block's last instruction outgoing (true/false if conditional branch) dataflow facts

Our analysis also prints revisited blocks in case of loops when recomputing new iterations of dataflow facts, in order to verify.

In order to facilitate our implementation, we make use of some of LLVM's built-in passes such as *instnamer* and *mem2reg*. LLVM's *instnamer* helps name temporary variables that would not be named in the bitcode (they are shown as *%i* where *i* is a number, typically the destination of load instructions). LLVM's *mem2reg* promotes memory references to be register references when appropriate. We utilize instnamer in order to store and propagate facts when possible in our analyses. mem2reg helps ease the use of our subexpression analysis.

Our analyses has the following constraints:

- It can only handle integer values

If any of these constraints are broken, there will be undocumented behavior.

Architecture

Since the goal of the project is to implement several different types of analyses, we wanted to build a generic structure to the code so that it would be easy to build upon and extend as new analyses are created. However, the design of a generic platform was not clear from the beginning.

The three main pieces of our software are the lattice, the analysis, and the block traversal logic. Between the analyses, the only part that differs significantly from one another is the analysis. The lattice and the block traversal logic are more or less similar.

A lattice object signifies whether it is top or bottom, and a container object for storing dataflow facts for use if it is neither top or bottom. The lattice object differs between the different analyses by the container object that is used at the lattice node. So we decided to template this class so any client could create a lattice with any type of container object that they choose. Most of the times, the container object would be a C++ map. Other times, it was a custom container object. By templating the lattice object, it allowed a greater flexibility and nimble prototyping during development.

After the lattice was templated, the block traversal logic was next. The logic was similar between the analyses and only differed by the current analysis and the type of lattice being used. It is with these two objects, that the template for our block traversal logic was created.

By having the flexibility of templates, it allowed us to ensure that the common logic is sound between the different types of analyses. Early on in the project, we started with clones of objects that became specialized toward each analysis. However, soon this specialized code started to drift and code management soon became a nightmare.

The following sections will describe each analysis in greater detail.

Section 1 (Constant Propagation)

We note that a constant propagation analysis is a must analysis. As such, we define the lattice with the given language:

$$
\begin{aligned}
&Given: \\
&X, Y, Z: \ Variable\ names \\
&Vars \ : Set\ of\ all\ possible\ variable\ names \\
&C \quad : Integer\ constant\ literals \\
&op \quad : \ + \mid - \mid * \mid / \mid \% \mid \& \mid \mid \mid \ \wedge
\end{aligned}
$$

$$D = 2^A \ where \ A = \{X \rightarrow C \mid X \in Vars\}$$

$$\perp = A\ (Full\ Set)$$

$\top = \varnothing$

$\sqcup = \cap$

$\sqcap = \cup$

$\sqsubseteq = \supseteq$

In our implementation, we handle and define flow functions as such:

$$F_{(X := C)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow C\}$$

$$F_{(X := Y)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow N \mid (Y \rightarrow N) \in in\}$$

$$F_{(X := Y\ op\ Z)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow N \mid (Y \rightarrow N_1) \in in \wedge (Z \rightarrow N_2) \in in \wedge N = N_1\ op\ N_2\}$$

$$F_{(if\ (X == C))}(in) = \begin{cases} in - \{X \rightarrow *\} \cup \{X \rightarrow C\} & true \\ in & false \end{cases}$$

$$F_{(if\ (X\ != C))}(in) = \begin{cases} in & true \\ in - \{X \rightarrow *\} \cup \{X \rightarrow C\} & false \end{cases}$$

$$F_{(Merge)}(in_0, in_1) = in_o \cap in_1$$

$$F_{(All\ others)}(in) = in$$

Passes used before analysis: *instnamer*

For each point in the lattice, we maintain a *map<string, ConstantInt*>*. The key is the name of the register, with the *ConstantInt** containing the value of the register at a specific point of time. As a must analysis, we can initialize block edges as bottom, or the full set.

Our analysis notes when an assignment to a register involves a constant or expression. In terms of a constant, the register will get updated. In terms of an expression, if it's a direct assignment that resolves to a constant, the register gets updated accordingly. If it's a binary expression with both operands resolving to a constant, we perform the operation and update the register with the result.

Another important form of instruction to make note of is a conditional branch. We can obtain new facts if the condition tests for equality (== or !=). Depending on the result, dataflow facts may or may not change. It is important to note the successor blocks, as their incoming edge facts will come from the true or false branch facts.

When we first started implementing this analysis, we noted some problems. There were temporary registers used, which were typically the destination of loads and used for future instructions. We

noted that such registers did not actually have a name and the name displayed in the readable bitcode was unretrievable. After some investigation LLVM's instnamer pass helped resolve this problem. In retrospect, this may or may not have been a great approach. Which it helped facilitate using a *map<string, ConstantInt *>*, it may be simpler to use LLVM classes to compare keys, since they are all pointers, and a pointer equality check could have been simple.

Section 2 (Available Expressions)

We note that available expression analysis is a must analysis. As such, we define the lattice with the given language:

$$
\begin{aligned}
&Given: \\
&X, Y, Z: \ Variable\ names \\
&Vars \ : Set\ of\ all\ possible\ variable\ names \\
&C \ : Integer\ constant\ literals \\
&op \ : \ + \mid - \mid * \mid / \mid \% \mid \& \mid \mid \mid \wedge
\end{aligned}
$$

$$D = 2^A \ where\ A = \{X \rightarrow Y \mid X, Y \in Vars\} \cup \{X \rightarrow Y\ op\ Z \mid X, Y, Z \in Vars\}$$

$$\bot = A\ (Full\ Set)$$

$$\top = \oslash$$

$$\sqcup = \bigcap$$

$$\sqcap = \bigcup$$

$$\sqsubseteq = \supseteq$$

In our implementation, we handle and define flow functions as such:

$$F_{(X := Y\ op\ C)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow Y\ op\ C \mid (A \rightarrow Y\ op\ C) \notin in \wedge A \in Vars\} \cup \{X \rightarrow A \mid (A \rightarrow Y\ op\ C) \in in\}$$

$$F_{(X := Y\ op\ Z)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow Y\ op\ Z \mid (A \rightarrow Y\ op\ Z) \notin in \wedge A \in Vars\} \cup \{X \rightarrow A \mid (A \rightarrow Y\ op\ Z) \in in\}$$

$$F_{(Merge)}(in_0, in_1) = in_o \cap in_1$$

$$F_{(All\ others)}(in) = in$$

Passes used before analysis: *instnamer, mem2reg*

For each point in the lattice, we maintain a custom object named *ExpressionContainer*. There were a few different considerations for what type of object should be used for storage in the lattice. It was probably the trickiest part of this analysis. At first blush, a map from string to expression seemed logical. However, having the ability to look up via expression would be needed.

It's for these reasons, the *ExpressionContainer* was created. A *map<string,Expression*>* is the internal structure used for storing the expressions. However, methods exist on the object to lookup by expression and return all variables mapped to that expression. A custom *Expression* class and comparator were implemented to handle this logic. As a must analysis, we can initialize block edges as bottom, or the full set.

Our analysis notes when an assignment to a register involves a binary operation. Since the underlying mechanism in our *ExpressionContainer* is a *map<string,Expression>*, constants and variables are handled similarly. The *Expression* class has two private members: *Value** operand1 and *Value** operand2. So the operand can either be a constant or variable. The important part is that the expression is saved in the map. The code handles most of the common binary operations. However, if it comes across one that it cannot handle, nothing gets propagated and the value is removed from the map.

When the analysis reaches a branch, the data flow facts are copied from incoming to outgoing. A merge is an intersection of the two edges.

Early on in processing our benchmarks, we noticed that there were many different types of intermediate representations for variables before a binary operation is performed. For instance, consider the simple code snippet with its LLVM translation:

i = 5; j = 5 + i;	%i = alloca i32, align 4 %j = alloca i32, align 4 store i32 5, i32* %i, align 4 %0 = load i32* %i, align 4 %add = add nsw i32 5, %0 store i32 %add, i32* %j, align 4

The pass *mem2reg* will eliminate a lot of these extraneous stores and loads. By performing this pass, the binary operations are condensed to a form that make it easily comparable to each other. It essentially does constant propagation and constant folding and would be analogous to running constant propogation before common subexpression elimination.

Section 3 (Range Analysis)

We note that a constant propagation analysis is a may analysis. As such, we define the lattice with the given language:

$Given$:

X, Y, Z : $Variable\ names$

$Vars$: $Set\ of\ all\ possible\ variable\ names$

C : $Integer\ constant\ literals$

op : $+ \mid - \mid * \mid / \mid \%$

$min(S)$: $Returns\ minimum\ value\ in\ set\ S$

$max(S)$: $Returns\ maximum\ value\ in\ set\ S$

$MININT$: $Minimum\ signed\ integer\ value$

$MAXINT$: $Maximum\ signed\ integer\ value$

$D = 2^A\ where\ A = \{X \rightarrow (C_1, C_2) \mid X \in Vars \wedge C_1 \leq C_2\}$

$\bot = \oslash$

$\top = A\ (Full\ Set)$

$\sqcup = \cup\ (in\ terms\ of\ number\ ranges)$

$\sqcap = \cap\ (in\ terms\ of\ number\ ranges)$

$\sqsubseteq = \subseteq$

For union and intersection of number ranges:

$(C_1, C_2) \cup (C_3, C_4) = \{(min(C_1, C_3), max(C_2, C_4))\}$

$(C_1, C_2) \cap (C_3, C_4) = \{(max(C_1, C_3), min(C_2, C_4)) \mid max(C_1, C_3) \leq min(C_2, C_4)\}$

*Note that a non-existent range is considered an empty range. An empty range $\cap$ with any other range results in an empty range. An empty range $\cup$ with any other range results in that other range. To showcase the behavior:

In our implementation, we handle and define flow functions as such:

$F_{(int\ X)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow (MININT, MAXINT)\}$

$F_{(X := C)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow (C, C)\}$

$F_{(X := Y)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow (L, U) \mid (Y \rightarrow (L, U)) \in in\}$

$F_{(X := C_1\ op\ C_2)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow (C_1\ op\ C_2, C_1\ op\ C_2)\}$

$$F_{(X := Y\ op\ C)}(in) = in - \{X \to *\} \cup \{X \to (min(B),\ max(B)) \mid B = \{N\ op\ C \mid N \in \{L_Y, U_Y\} \wedge Y \to (L_Y, U_Y) \in in\}\}$$

$$F_{(X := Y\ op\ Z)}(in) = in - \{X \to *\} \cup \{X \to (min(B),\ max(B)) \mid B = \{N_1\ op\ N_2 \mid N_1 \in \{L_Y, U_Y\} \wedge Y \to (L_Y, U_Y) \in in \wedge N_2 \in \{L_Z, U_Z\} \wedge Z \to (L_Z, U_Z) \in in\}\}$$

$$F_{(if\ (X == C))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (C,C)\} & true \\ in & false \end{array} \right\}$$

$$F_{(if\ (X != C))}(in) = \left\{ \begin{array}{ll} in & true \\ in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (C,C)\} & false \end{array} \right\}$$

$$F_{(if\ (X < C))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (MININT, C-1)\} & true \\ true\ branch\ of\ F_{(if(X \geq C))}(in) & false \end{array} \right\}$$

$$F_{(if\ (X > C))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (C+1, MAXINT)\} & true \\ true\ branch\ of\ F_{(if(X \leq C))}(in) & false \end{array} \right\}$$

$$F_{(if\ (X \leq C))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (MININT, C)\} & true \\ true\ branch\ of\ F_{(if(X > C))}(in) & false \end{array} \right\}$$

$$F_{(if\ (X \geq C))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge R_2 = (C, MAXINT)\} & true \\ true\ branch\ of\ F_{(if(X < C))}(in) & false \end{array} \right\}$$

$$F_{(if\ (X == Y))}(in) = \left\{ \begin{array}{ll} in - \{X \to *\} - \{Y \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge (Y \to R_2) \in in\} \cup \{Y \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge (Y \to R_2) \in in\} & true \\ in & false \end{array} \right\}$$

$$F_{(if\ (X != Y))}(in) = \left\{ \begin{array}{ll} in & true \\ in - \{X \to *\} - \{Y \to *\} \cup \{X \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge (Y \to R_2) \in in\} \cup \{Y \to R_1 \cap R_2 \mid (X \to R_1) \in in \wedge (Y \to R_2) \in in\} & false \end{array} \right\}$$

$$F_{(if\ (X < Y))}(in) =$$

$$true : \{in - \{X \to *\} - \{Y \to *\} \cup \{X \to R \cap (MININT, Y_U - 1) \mid (X \to R) \in in \wedge (Y \to (Y_L, Y_U)) \in in\} \cup \{Y \to R \cap (X_L + 1, MAXINT) \mid (Y \to R) \in in \wedge (X \to (X_L, X_U)) \in in\}$$

$$false : \{true\ branch\ of\ F_{(if\ (X \geq Y))}\}$$

$$F_{(if\ (X > Y))}(in) =$$

$$true : \{in - \{X \to *\} - \{Y \to *\} \cup \{X \to R \cap (Y_L + 1, MAXINT) \mid (X \to R) \in in \wedge (Y \to (Y_L, Y_U)) \in in\} \cup \{Y \to R \cap (MININT, X_U - 1) \mid (Y \to R) \in in \wedge (X \to (X_L, X_U)) \in in\}$$

$$false : \{true\ branch\ of\ F_{(if\ (X \leq Y))}\}$$

$$F_{(if\,(X \le Y))}(in) \;=$$
$$true\;:\{in - \{X \to *\} - \{Y \to *\} \cup \{X \to R \cap (MININT, Y_U) \mid (X \to R) \in in \wedge (Y \to (Y_L, Y_U)) \in in\}$$
$$\cup \{Y \to R \cap (X_L, MAXINT) \mid (Y \to R) \in in \wedge (X \to (X_L, X_U)) \in in\}$$
$$false:\ \{true\ branch\ of\ F_{(if\,(X > Y))}\}$$

$$F_{(if\,(X \ge Y))}(in) \;=$$
$$true\;:\{in - \{X \to *\} - \{Y \to *\} \cup \{X \to R \cap (Y_L, MAXINT) \mid (X \to R) \in in \wedge (Y \to (Y_L, Y_U)) \in in\}$$
$$\cup \{Y \to R \cap (MININT, X_U) \mid (Y \to R) \in in \wedge (X \to (X_L, X_U)) \in in\}$$
$$false:\ \{true\ branch\ of\ F_{(if\,(X < Y))}\}$$

$$F_{(Merge)}(in_0,\ in_1) \;=\; in_o \cup in_1$$

$$F_{(All\ others)}(in) \;=\; in$$

Passes used before analysis: *instnamer*

For each point in the lattice, we maintain a *map<string, vector<int> >*. The key is the name of the register, with the vector of ints containing two values, the minimum and the maximum of the range (both inclusive). As a may analysis, we can initialize block edges as bottom, or the empty set.

When we first started implementing this analysis, we noted some problems. There were temporary registers used, which were typically the destination of loads and used for future instructions. We noted that such registers did not actually have a name and the name displayed in the readable bitcode was unretrievable. After some investigation LLVM's instnamer pass helped resolve this problem. In retrospect, this may or may not have been a great approach. Which it helped facilitate using a *map<string, vector<int> >*, it may be simpler to use LLVM classes as the key and comparator, since they are all pointers, and a pointer equality check could have been simple. Also,the value, which is a vector, could probably be replaced with an LLVM class, such as ConstantRange. Alas, when we started implementation, we did not discover this class until late in the project process.

Implementing dataflow functions for range analysis is much more complex than the other analyses. For example, for binary operations we must find the minimum and maximum computed value of multiple combinations of expressions while considering whether or not it is commutative. In addition, there lies tremendous difficulty when formulating dataflow functions for range analysis, especially in terms of relational operators. Also, we must propagate a variable's range values to each dataflow function, which is shown above. Plus, if both are variables, both their ranges need to change also! For example, if $X >= Y$, then $(X_L, X_U) \cap (Y_L, MAXINT)$, and $(Y_L, Y_U) \cap (MININT, X_L)$, and that's just

the true branch. Despite this layer of complexity, propagation and bidirectional relational dataflow are supported and implemented.

Among us, there was a degree of confusion when computing our dataflow functions in branch statements involving an unknown range (also known as an empty range) from a variable, and a known range from a variable or constant. The result of the intersection is the empty range, due to the way how intersection of ranges work. For example, if $F_{(if\,(X>Y))}(in)$ where $X \in in \wedge Y \in in$, the result of the function would ultimately return $in - \{Y \rightarrow *\}$. It seems correct in that such the branch flow would provide more precise facts, heading down the lattice.

Section 4 (Intra-procedural Pointer Analysis)

We note that there is relevance in a may and must pointer analysis, and implement both. We define the lattice with the given language.

May Pointer Analysis

$Given:$

$X, Y, Z: \ Variable\ names$

$Vars \ : Set\ of\ all\ possible\ variable\ names$

$D = 2^A \ where \ A = \{X \rightarrow Y \mid X, Y \in Vars\}$

$\bot = \oslash$

$\top = A\ (Full\ Set)$

$\sqcup = \cup$

$\sqcap = \cap$

$\sqsubseteq = \subseteq$

$F_{(X := \&Y)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow Y\}$

$F_{(X := Y)}(in) = in - \{X \rightarrow *\} \cup \{X \rightarrow A \mid (Y \rightarrow A) \in in\}$

$F_{(Merge)}(in_0, in_1) = in_o \cup in_1$

Must Pointer Analysis

$Given:$

$X, Y, Z: \ Variable\ names$

$Vars \ : Set\ of\ all\ possible\ variable\ names$

$D = 2^A \; where \; A \; = \; \{X \rightarrow Y \mid X, Y \; \in \; Vars\}$

$\bot \; = \; Full \; Set$

$\top \; = \; \oslash$

$\sqcup = \; \bigcap$

$\sqcap = \; \bigcup$

$\sqsubseteq = \; \supseteq$

$F_{(X := \&Y)}(in) \; = \; in \; - \{X \; \rightarrow \; *\} \; \cup \; \{X \; \rightarrow Y\}$

$F_{(X := Y)}(in) \; = \; in \; - \{X \; \rightarrow \; *\} \; \cup \; \{X \; \rightarrow A \mid (Y \rightarrow A) \; \in \; in\}$

$F_{(Merge)}(in_0, \; in_1) \; = \; in_o \cap in_1$

Passes used before analysis: *instnamer*

For each point in the lattice, we maintain a *map<string,set<Value*,valueComp>>* where the string is the variable name, and the set is the set of possible values that the variable may point to or must point to. Obviously in the must analysis, the set has a length of 1.

The main instructions that were focused on were both load and store. Utilizing the LLVM method *isPointerTy(),* we could glean information as to whether the destination register is a pointer or not. If it is, its value was added to the map.

The only difference that we say between may and must is when it came to merging. As variables were loaded and stored, the map was updated accordingly. When branches came, the map was copied from incoming to outcoming edge. However, when it came to merging, only pointer mappings that had the same type of information between the different edges were propagated to the outgoing edge. For may analysis, the edges were unioned.

With these analyses, one can combine it with previous analyses to improve the overall range and precision. For example, with may and must pointer facts at each points, one can evaluate extra dataflow function for analyses like constant propagation such as:

$F_{(X := *Y)}(in) \; = \; in \; - \; \{X \; \rightarrow \; *\} \; \cup \; \{X \; \rightarrow N \mid \forall Z \; \in \; may - point(Y), \; (Z \; \rightarrow \; N) \; \in \; in\}$

$F_{(*X := Y)}(in) \; = in \; - \; \{Z \; \rightarrow \; * \mid Z \; \in \; may - point(X)\} \; \cup \; \{Z \; \rightarrow \; N \mid (Y \; \rightarrow \; N) \; \in \; in\} \; \wedge \; \{Z \; \rightarrow \; N) \; \in \; in\}$
$\cup \{Z \rightarrow N \mid Z \in must - point - to(X) \wedge Y \rightarrow N \in in\}$

However, we did not implement combinations of passes in our project. For future work, this would be simple to implement given the way we've designed our architecture. However, due to constraints in time, we did not implement this.

Benchmarks

*Note, the benchmark scripts might be dos line ending files. You need to re-save the file using Unix line endings or you can use the dos2unix command line utility to do it for you.

branchloop: Aimed towards testing block traversal and loop behavior. Also contains many equality conditions on branches.

range: Aimed towards testing relational conditions such as $\leq, \geq, <, >$

pointer: Test dataflow functions for pointers, should provide different results for may and must analyses.

expression: Aimed toward the different types of binary operations and ensuring that the correct dataflow facts are propagated for available expression analysis.

Conclusions

One of the more challenging aspects of the project was to get started. The first phase of the project was smaller in scale and was a bit more manageable. This time around, a lot of the infrastructure needed to be built from the ground up. So one of the first questions that we wrestled with from the beginning was how to start it? How do you build the infrastructure so many different types of analyses could be run?

Another challenging piece of the project was LLVM itself. LLVM is a large toolchain with an impressive class library. Jumping in it and trying to learn how to parse the information in code was quite a challenge. No one in our group was familiar with LLVM so we had to learn and understand the type library and what methods needed to be used to get the information we would need. I suspect that as we would get more familiar with LLVM we could leverage more of its power. We wouldn't be surprised if we duplicated work or represented something that maybe has a cleaner representation in LLVM.

It was quite exciting to see our analyses start working. After implementing both the analyses and the lattices, the practicality of the theory began to take shape in our work which is rewarding.

All in all, we can see that this platform could be extended for other analyses. Not only that, but transformations of the code itself would be an interesting next step to take. It would also be interesting to combine the analyses to glean even more information from the code.