STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 2, 2018
DOI: 10.24193/subbi.2018.2.05

INCREMENTAL DECOMPILATION OF LOOP-FREE BINARY CODE: ERLANG

GREGORY MORSE, DÁNIEL LUKÁCS, AND MELINDA TÓTH

Abstract. Decompiling byte code to a human readable format is an important research field. A proper decompiler can be used to recover lost source code, helps in different reverse engineering tasks and also enhances static analyzer tools by refining the calculated static semantic information. In an era with a lot of advancement in areas such as incremental algorithms and boolean satisfiability (SAT) solvers, the question of how to properly structure a decompilation tool to function in a completely incremental manner has remained an interesting problem.

This paper presents a concise algorithm and structuring design pattern for byte code which has a loop-free representation, as is seen in the Erlang language. The algorithms presented in this paper were implemented and verified during the decompilation of the Erlang/OTP library.

1. Introduction

Decompilation of compiled code is the process of transforming a compiled module typically in a machine readable byte code format, back into a human readable source code format. A decompiler, is a tool which automates this process. The practical nature and visible result of a decompiler is an important tool which is useful for source code recovery, reverse engineering of hostile code, or for compiler validation. Decompilers have almost exclusively relied upon an approach which treats the binary as a flat and static block of instruction

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68W01, 68N20.

1998 *CR Categories and Descriptors.* code [**Computing Methodologies - SYMBOLIC AND ALGEBRAIC MANIPULATION**]: Algorithms – *Nonalgebraic algorithms*; code [**Software - PROGRAMMING LANGUAGES**]: Processors – *Incremental compilers*.

Key words and phrases. incremental decompilation, Erlang, dominator tree, post-dominator tree, code duplication.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

data, and proceeds with various stages also called passes over the byte-code and iteratively through various graph structures until it achieves source code. This approach is well understood, yet it is not a general method, and it does not always produce usable or valid results.

The theoretical limit of decompilation lies at the general computability problem of decidability, famously highlighted by the halting problem. It is proven that there exist cases where it is undecidable whether or not a program continues executing or stops, and this can be generalized to any decision pathway. The first case needing this level of generality is unreachable code. The other is that of self-modifying code, where the binary code is able to modify itself.

The motivation for a study in incremental decompilation theory becomes clear: not only for generality of the decompilation process, but the approach invariably could be used to make better, more flexible compilers. Compilers and decompilers involve the same theory as both do binary transformation of code structures, albeit the programming languages utilize context-free grammars and encourage block structures which is typically far less expressive than byte code which tends to allow any control flow graph (CFG). A CFG can contain sequences, selections and loops. The language of Erlang [1] has been chosen as it is an excellent case study for incremental decompilation for several reasons: no back edges inducing loops in the graph, impossibility of self-modifying code, and an easy to process byte-code which will be pre-structured as instructions by its own libraries.

As a main contribution, a framework for incremental decompilation is presented in this paper based on using a symbolically semantic equivalent representation and meticulous graph structuring. This includes algorithms contributed for scanning methods, processing at merge and exit nodes and variable emission.

Various concepts and tools maintain the incremental cascading of effects. An analysis of semantic equivalence of byte code in a meta-data enhanced abstract syntax tree (AST) representation for any language allows for a cross-language approach. The importance of variable emission scenarios, classifying side effects and nearly inexpressible byte code operations is demonstrated. The incremental maintenance of dominator trees, reachability, and common ancestors are discussed as with minimal processing at merge nodes.

Data interfaces encapsulate the graph structure containing basic blocks, and another is used for the enhanced AST. Algorithms are considered for overall decompilation, handling edges not expressible in the target language, merge nodes with their optimized processing and minimal variable emission. Two scanning algorithms for overall decompilation are studied. The nuances highlight the technical challenge of achieving a consistent incremental algorithm.

Since code copying is a technique which has exponential growth consequences in complexity, simplifications for boolean short circuits are considered. The clean up of the AST is itself a crucial element of the decompiler for a readable and usable decompiled output.

2. Background

"Erlang is a programming language originally developed at the Ericsson Computer Science Laboratory. OTP (Open Telecom Platform) is a collection of middleware and libraries in Erlang." [2].

An abstract syntax tree (AST) is a tree containing information directly corresponding to the grammar of the language. It can be pretty printed to Erlang source code, or compiled to BEAM, or even emulated by the Erlang eval library. In fact the Erlang shell uses this eval emulator to execute commands through evaluating the AST directly without BEAM conversion, albeit with a performance penalty and possibility to execute code which fails the compilers more strict validation on things such as variable bindings in block structures.

A decompiler based on a graph rewriting technique was written which shows that multiple valid approaches to Erlang decompilation are certainly possible [3]. Both follow further a seminal work for decompiler graph structuring [4].

2.1. **BEAM code.**

The BEAM code, provides a set of opcodes, which operates on a state containing the current instruction location, 1024 registers **{**x, 0**}** through **{**x, 1023**}** and a stack starting at $\emptyset$ which when initialized has a head always at **{**y, 0**}** and can be of any size limited by the memory of the system or any emulator configured limit, and 16 special floating point registers **{**fr, 0**}** through **{**fr, 15**}**. The current line number is specified in an instruction and is part of the state, although it is only accessible through stack traces which should only be accessed when errors occur per documentation recommendation. The BEAM code is contained in a file with the .beam extension and directly representable by a large tuple containing the whole module, some attribute information, its functions and their BEAM opcodes.

The state of the system is accessible thanks to external code libraries, so potentially unknowable values could find their way into these registers when interacting with the greater system state.

A few special opcodes are used in the emulator itself where it modifies the original BEAM code but these do not concern the decompiler directly.

The correctness of BEAM code has several levels: syntactic valid when compiling it, correctness done by the compiler's validator to prevent situations that would crash the emulator, and finally that which runs on the emulator without crashing. For the sake of generality, it is best to consider the latter correctness as the decompiler could encounter code which is custom crafted with a modified Erlang source code which disables the validator. The gold standard of correctness is very hard to achieve, namely compilability to identical binary code. Though typically its need is rare, any timing, line number or other minute side effects require it.

2.2. **Incremental Graph Maintenance.** There is inefficiency of constantly recomputing the dominator information which is utilized constantly as soon as any decidable merging happens in the control flow. It is assumed the reader understands the shorthand graph notations for edges and dominators.

Maintaining a graph structure upon edge additions is termed as an incremental algorithm, while further including edge deletions which typically is more complex, is termed as a fully online algorithm.

Algorithms for maintaining a dominator tree incrementally on edge addition, as well as a fully on-line algorithm which also adds edge deletion are known. For this purpose, although G. Ramalingam of IBM Research Laboratories provided the next major break-through [5], the preferred algorithm is the Sreedhar, Ghao, Lee algorithm which uses a data structure called a DJ-graph which is a dominator tree with join edge information about the connectivity of the graph due to the fact that dominator information alone is not enough to easily determine the scope of how much of the tree is effected by addition or deletion. Join edges are all edges which are not between ancestors and descendants in the dominator tree. By introducing the concept of an iterated dominance frontier (IDF) [6], a relatively simple and elegant algorithm emerges to incrementally maintain the DJ graph [7].

Reachability of a given graph node to the return node is also important information to decide how to process when an exception disconnects the control flow for a node or a subset of nodes to the return node. Determining this information cumulatively can be performed with a simple depth or breadth first walk from the return node up the tree to the root where the visited nodes are the set. Incrementally maintaining this information is trivial for edge addition, as adding edges does not reduce reachability and nodes always start as reaching the return node to later not reaching it. For generality, only when a predecessor not reaching is added to a successor which is reaching, then the whole reverse subgraph of the predecessor is added to the set. For edge deletion, the successor if reachable which implies the predecessor is reachable can check if any of its remaining successors is still in the reaching set, otherwise remove itself and continue the process recursively up the graph. The two processes are mirrored as can be seen formally (where REVREACH(X) is the subgraph reachable from the reverse graph rooted at X):

$$\text{AddEdgeReachSet}(U, V, R_s) : \begin{cases} U \notin R_s, V \in R_s & R_s \cup \text{REVREACH}(U) \\ \text{otherwise} & R_s \end{cases}$$

$$\text{NREVREACH}(U, R_s) = [U] \cup \forall Y \in \text{PREDS}(U), \text{NREVREACH}(Y), \nexists X \in \text{SUCCS}(Y), X \in R_s$$

$$\text{RemoveEdgeReachSet}(U, V, R_s) : \begin{cases} V \in R_s, \nexists X \in \text{SUCCS}(U), X \in R_s & R_s \setminus \text{NREVREACH}(U) \\ \text{otherwise} & R_s \end{cases}.$$

It should be noted that actual implementations would likely use breadth first search (BFS) methodology for efficient computation. In fact, this reachability question is more formally known as a transitive closure, and maintaining transitive closure for a directed or undirected graph is a problem which has been studied. Solutions exist such as based on maintaining the order of a linked list, as this problem efficiency-wise is a data structure problem [8].

The depth first search (DFS) tree is a tool used mostly for efficient computation of dominators in this context. Its incremental computation is still an open problem for directed graphs due to the fact that one edge addition or deletion can cause very far reaching changes, along with the fact that multiple valid traversals are possible. In the acylic case however, there are algorithms known [9].

3. Main Contribution: Semantic Equivalence

Various data-flow oriented semantic equivalence of various BEAM opcodes and their corresponding Erlang code equivalent are analyzed. This is limited by control flow structures which are not single instructions but various sequences tied together in certain ways and requires graph analysis. Of importance to the data flow analysis is the concept of purity. Purity of a function is an attribute that the code it contains does not change the state of the system in anyway except that which is returned to the caller.

Three valid approaches to consistent side effect handling are apparent based on the purity analysis and detection. The first is to always emit variables for every state change, but this makes the code unreadable requiring later clean up based on single usage or dead variables, and is expensive as tracing original values for decidability in these variables requires traversing dominators. The second approach is to emit variables for any state change except pure built-in functions (BIFs). This is straightforward and readily implementable. It is the approach which is taken for simplicity. The last approach and more elegant is to emit variables for any state change except functions detected as pure by tools such as PURITY [10]. To not stick to one of these conservative approaches would ultimately to be emitting code which is in fact not reflective of the original code. The discussion does not end with side effects as any values represented as the result of AST emission also must be treated as having a side-effect or the state itself would need to contain AST entries. To avoid this, some situations require variable emissions for the entire block structure of the lambda function creator. The captured variables of the fun expressions also must have variables emitted, not because of side effects, but because injecting side-effect free representational expressions into the captured variables is certainly different from the original code, where a type of fence exists between the captures and the lambda function since only variables can be captured, and compiler be optimization beyond this is not done. Tables of all instructions with side effects and variable emission fences should be generated and codified.

The whole block of the function has a return value as well which is represented by a single value which must be denoted as a state item. In Erlang, it is always {x, 0}. In practice, this could be a group of values but ultimately most languages allow its expression as a singular value so some type of language grouping element would need to be used to bundle them regardless, such as tuples or lists. The generality could thus be extended.

3.1. **Byte code and Metadata Enhanced Abstract Syntax Tree.** The classical view of byte code running on a system is that of a Fetch, Decode and Execute loop, as per the way the central processing unit (CPU) itself works. The Fetch and Decode step can be considered as one combined unit when not considering timing issues. For a decompiler, this view is changed to a Decide how to Fetch and Decode, and Symbolically Execute loop. The state itself is symbolic of the actual state and does not contain literal values. However the fetch and decode operation when generalized must decide as specifically as it can, the set of bits resultant from the current symbolic state.

To maintain the data flow aspects of decompilation in the AST while it structures itself based on control flow, entries are utilized with a special meta-data key. These contain the values of the x, y and fr registers representing the current state. These are guaranteed at the beginning of every block, and where there are sequences in a block emitted due to side effects including function calls or variable assignment, an updated meta-data entry appears after it.

A table and then code for the state should be compiled for the target language which for Erlang includes the line number and registers as discussed.

3.2. **Scanning and Overall Algorithm.** The sequential scan for a decompiler is not only straightforward to implement, and seems to be a natural choice for scan order, it leads to a number of consequences when dealing with incremental algorithms and decidability aspects. In fact for decidability, it is not sufficient, and cannot be considered as an appropriate algorithm at all, since the decidability algorithm could effectively decide that it needs to know more about another pathway before it can make a decision. This is thereby a dependency and so a different scan ordering addressing these dependencies in decidability must be looked at. However the incremental theory of both approaches will be developed hereby, as some very interesting details are gleaned from the differences and answering decidability questions is not always a requirement.

The merge nodes are the motivating factor for decisions, and processing of these nodes which is discussed shortly, can be done best when all recursive predecessors which reach the merge point are already scanned, as at this point its post-dominating status is stable. Otherwise if it does not post-dominate, it may later, or if it does post-dominate, it may post-dominate more nodes later, in both cases based on exit scenarios. So the best place to scan at any moment, is any non-merge node, or the un-scanned merge node which is not reached by any other un-scanned merge node and it is processable meaning a colliding edge is confirmed not to reach it. Instead of doing a series of negative reachability checks, a BFS ordering of the graph can provide the topmost node which would reach all the others, so the first un-scanned and processable merge node in the BFS is the best candidate to scan. This greatly simplifies the exit scenarios and even better reduces exponential code copying which occurs as a result of delayed decisions.

The decompiler should maintain the status of each node in a simple structure called the ProcessingState which can have values of: Unprocessed, Processable, Processed, Colliding, which progresses as it is added and then becomes processable then processed, and thereafter possibly marked as a colliding for optimal processing of the node its edge collides, and then it is moved back to Processed. The BFS ordering scan deals with using the processing state to chose the next processable node and is guaranteed to mark all as processing if on jumps it looks ahead to the next node and marks it. The overall algorithm of the incremental decompiler decidably fetches and decodes, and symbolically executes in a loop while structuring based on conditionals, merge points, side effects, or semantic equivalents (see it with example in Appendices B and C).

3.3. **Return, Exit Nodes and Conditionals.** Due to the difficulty of maintaining the post-dominator tree as it could have multiple roots, an exit node is introduced which any node added to the graph maintains connectivity to at all times including the entry node. And also due to the nature of functional languages returning a value upon exit, another placeholder node called a return node representing the emission of a return value is also added. The return node will be permanently connected to the exit node, and all nodes who are being processed or not yet processed will maintaining a successor of the return node. Any exceptions or errors, will cause a node to redirect from the return node to the exit node.

However, the exit node functions differently as it is completely symbolic and no merge occurs. Therefore it does not make sense for it to post-dominate any nodes in the graph, beyond those in the subset of nodes which do not reach the return node. Therefore the reverse graph needs to be maintained slightly differently than the graph, and a technique for doing this, is that any node which is a predecessor of the exit node has its successor drawn as the successors to its set of predecessors which are part of the return node reaching set R_s. These nodes are chosen by the nearest predecessors reaching the return node or formally as PredSetReach(U, R_s).

The obvious caveat, is that in certain cases, the return node may not be needed, if all code paths go to the exit node. In this case, the return node could either be deleted or more conveniently made to be the sole successor of the exit node. Figure 1 indicates the presumed structuring first, and then one of them becomes the final structure.

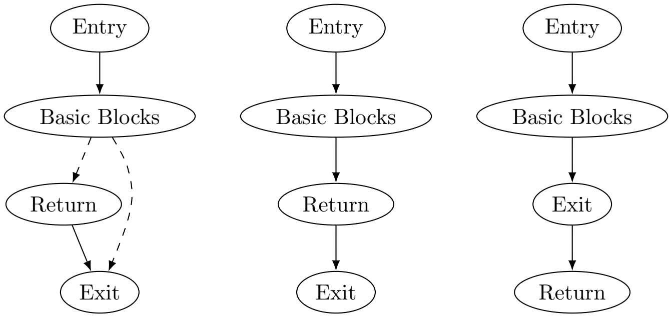


FIGURE 1. Three control flows: the latter two only for final CFG with no exceptions/exits or no normal returns

For the dominator tree, the prior strategy introduces a problem. The exit node should not dominate any node except the return node, as this is a special allowed control flow transfer which can exit the function regardless of where it is in the AST. To deal with this, when computing the dominator tree, all nodes succeeded by the exit node, have all of their predecessors' successors, replaced with the exit node successor. In effect, this makes the node have no effect on the dominator tree, as if it were deleted.

The exception/exit incidence including the relevant opcodes, their context, semantic equivalence and whether they are singular pathways which effect R_s should be compiled in a table and then coded.

No effect on post-dominator of nearest return reaching node, as the decision node itself is the nearest return reaching node, since by definition its continuation path is unexplored and thus still reaching the return node, and previously the node itself was unexplored thus reaching the return node.

An important routine of the decompiler is conditional structuring which revolves around laying out the edges in the graph for conditionals and exits which comprise the control flow instructions as well as any block structure instructions which require further analysis due to the fact that they are not single instructions but sequences thereof, and adding various AST emissions as well as meta-data embedded within.

Another table should be compiled of all the control flow instructions which structure each opcode as series of AST modifications and graph changes, while keeping track of the next node for sequential scanning over binary conditionals. Figure 3 demonstrates the equivalent non-block re-structuring of multi-selection conditionals to binary conditionals. Since variable assignment is allowed consistently within conditionals, using binary conditionals, and variable assignment, avoids what is termed a block structure, a language structure requiring a single return value and consistent entry and exit from the structure. The return values can simply be unused as hence ignored by avoiding block structuring. Block structuring has a solution albeit complex and not discussed. The cost is that exponential code duplication can occur.

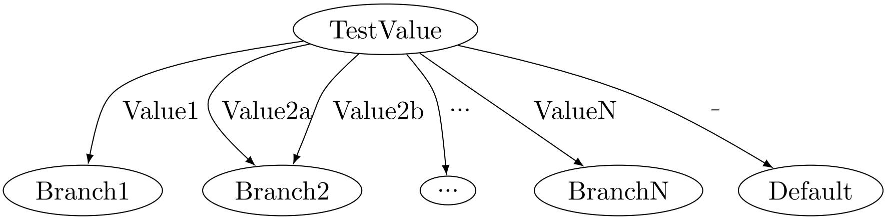


Figure 2. BEAM style select branching structure

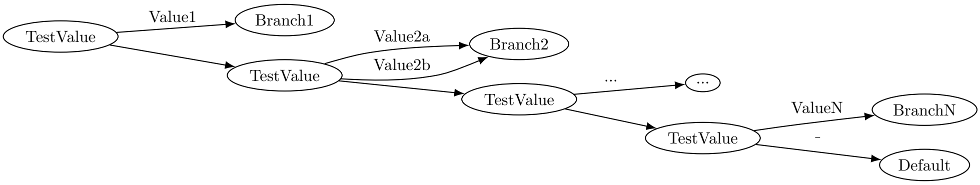


Figure 3. Select semantic equivalent for nested, non-block style structuring

3.4. **Merge Nodes, Cross Edges and AST Mapping.** Anytime two or more edges are incumbent on a node, a merge occurs where the data flows through different paths in the graph must be coalesced. The single static assignment (SSA) form has been used to represent this using a ϕ-function to represent state values at these collision places.

An optimization can occur at this stage when the merge node is reached by an implicit edge referred to hereby as a colliding edge and this edge is also classified as a cross edge. This is an optimization over merely using the equivalent jump control flow semantic equivalent.

The incremental theory develops by first realizing that the importance of the merge nodes is underscored by the fact that they post-dominate other nodes. Variable assignments at appropriate places can be done in a consistent way such that these merge nodes are not important unless they are also post-dominators. Two events can occur which cause a merge node to become a post-dominator: 1) Most common is that a label is reached where a prior basic block is implicitly added as a colliding edge, or alternatively it could be solely a merge from prior basic block jump targets. 2) An exception or exit occurs. This potentially causes a chain of potential post-dominator merge nodes which must be checked. However these do not merge in the same sense as having a colliding edge. Reprocessing of already processed post-dominators is possible, since an already processed post-dominator can become a post-dominator of an additional node when its exit is realized. In this case, no variable assignments can occur but cross edge processing must proceed. A node which is processed and all that it reaches is immediately post-dominated by a node which is also processed are not further processable and do not risk an exit occurring on them, a stable and decided set of the nodes.

Timing	**Node State**	**Action**
On Reach	Not post-dominating any node	Do nothing
On Reach	Post dominated ≥ 1 nodes	Process fully
Exit	Unprocessed	Do nothing
Exit	Processed and not post-dominating any node	Process fully
Exit	Processed and post-dominating ≥ 1 nodes $\exists N \in$ REVREACH(Node), get_processed(PIDOM(N)) $\neq$ Processed	Process w/o variable assignment

Table 1. Post Dominating Node Incidence Classifier for Sequential Scanning

Table 1 and a HandleExit function is thereby a consequence of a sequential scan through the code. This is a convenience and shortcut taking approach for Erlang BEAM code since it can be safely assumed if emitted from the compiler to be all reachable and since it is not able to self-reference and hence self-modify itself, a sequential scan only need know that a given label is reachable from the EntryLabel. An alternative and improved incremental approach could keep a queue of un-scanned locations, and not scan them until they can be processed and hence post-dominate some nodes, while at the same time all nodes reaching it area also processed. In this case, the whole table is unnecessary as there is only a single case that processes fully when post-dominating according to these conditions. This is a more general and more ideal way of decompiling, but in some contexts, Table 1 can also be a relatively easy to implement and workable methodology.

These are filtered so only the already visited ones are considered. Finally the merge node processing occurs for all of them, as it would normally. Block structures which have only exit paths should also be identified and processed at this point for maximizing incremental effect unless all of their processing would be done at once at the end. This discussion cannot continue without introducing the effect of cross edges as they are the most important aspect towards the resolution before variable assignment occurs.

The DFS-based cross edges and inexpressible forward edges – from the perspective of the AST which reflects valid edges – must be processed to determine where copying of code can resolve these situations. In general, the AST entries are allowed edges in a language when moving: forward to any ancestor, up to a any descendant, to an immediate sibling, or to an exit/exception. Other edges are considered to be a cross edge, and is represented hereby as EdgeClassifier.

A DFS is needed which given that the AST is an ordered tree, is unique and a sorting of the paths of the nodes. Edges which are cross edges will be represented by variable CrossEdges $\leftarrow$ E \ EdgeClassifier(E).

The cross edges must be classified based on their significance for processing based on the current node. Therefore the CrossEdges are further mapped by a function to the nearest post-dominator or nearest common ancestor (NCA) which is the longest common suffix (LCS) between dominator paths, except that the node in consideration has included itself in its dominator sequence:

$$\text{NCA(PathX, PathY)} = \text{hd(LCS(PathX, PathY))}$$

$$\text{NearPDom(P, S, Node)} = \text{NCA}(\begin{cases}\text{DOM}(P) & P = \text{Node}\\ \text{SDOM}(P) & \text{otherwise}\end{cases}, \text{DOM(S)})$$

$$\text{NearCrossPDom(Node)} \leftarrow \forall(P, S) \in \text{CrossEdges, NearPDom(P, S, Node)}$$

The cross edges are then filtered so that CopyEdges $\leftarrow \forall(P, S) \in$ CrossEdges, Node = NearPDom(P, S, Node). These are further sorted based on the DFS, where the greater successors or in case of equality, greater predecessors are processed first, hence a bottom up strategy, for convenience and consistency which allows certain data structure optimizations. These values are incrementally computable and as for the NCA, it could be recalculated based on the set of changed nodes in incremental dominator recomputation.

The merge structuring algorithm is divided into three stages where when a colliding edge is a cross edge, a special merge node is added as a place holder for code copying and variable assignments, followed by cross edge code duplication, and then variable assignment. The graph copying going on here removes cross edges, but otherwise has no effect on the post-dominator tree of the original nodes, so intermediate re-computation is not necessary.

Shortcuts are possible here where recognizing the nested, geometrically opposite diamond like shape of andalso as well as orelse is the most prevalent and most useful one as this causes an exponential blow up in the graph and therefore also the AST which makes processing slower even if it can be cleaned up at the end. Otherwise the only illegal edges generally seen are the result of compiler optimizations which reduced copied code, and hence copying the code again becomes necessary. The basic structures are easy patterns to find and can be recursively applied ideally in a bottom to top order.

A bijective mapping between the state nodes in the generated AST and the control flow graph is needed which also must be incrementally maintained. A simple method is to use an indexed path down the tree. The DFS of the tree then is nothing more than a sort operation over of these labels based on their indexed paths. This is incrementally maintained via a two-way indexing scheme for binary search and insertion in sorted order.

The data structure to encapsulate the AST would include not only the actual AST, but the mapping and a sorted two-way indexing for efficiency in lookups and ranking when doing operations on the graph where a DFS ordering based on the AST is necessary. The path of the nodes in the tree would be encapsulated and efficiently implemented in some format. The enhanced AST structure should provide operations for inserting nodes, child nodes, getting or setting the meta-data, getting the DFS of the ordered tree, and copying from a node into another node part of the AST with respect to the NCA.

As for the graph, adding edges, removing edges, getting a BFS, the reverse graph DFS ordering, and the same copy graph operation as for the AST is needed. Symbols for the entry, return, exit nodes and the next node and current node should be a part of this. The graph should be able to answer the various mathematically represented queries for edges, (post-)dominators, reachability, CrossEdges, NearCrossPDom(Node) (see Appendix A).

3.5. **Variable Assignment.** Variable assignment takes all the merge nodes that are post-dominators, after cross edges have been processed, and assigned variables to the minimal consistent set of nodes necessary based on the dominator tree. The merge node itself has its dominator used as a reference point, as variables would not need to be assigned beyond the dominator. But all of its predecessors, and their recursive dominators up to this top dominator need to be considered. If changes or the lack of changes are consistent between all children of a dominator, then the dominator itself can be chosen instead of the child nodes, which is a key reduction in avoiding excessive variable emission. The naive strategy would be to assign variables to all predecessors if there is a change in any one of them.

A very useful optimization is that it is possible to only perform variable assignment on merge nodes that are post-dominators, after cross edges have been processed, but then the algorithm needs to be adjusted to go through the predecessors of these nodes that do not post-dominate.

First the set of changed nodes is determined by recursively going through the dominator tree, and then one more traversal determines the nodes that did not change minimally with respect to the ones that did, so a variable emission always occurs to ensure that a variable is present at the merge node in all pathways. The state information should be maintained therefore by having symbolic current state information for every node as it would be expensive and undesirable to recompute something so easily maintained incrementally.

Algorithm 1 finds the minimal set of changed nodes with regards to the predecessors and their dominators, while Algorithm 2 does this for the nodes which did not change with respect to the nodes which did change in a similar way with this additional exclusion and not needing to query for state changes. Finally Algorithm 3 gives routines used in the merge algorithm for emitting a single value on return, or going through all state item values otherwise. A summary example is provided in Figure 4.

Algorithm 1 Find Minimal Set of Changed Nodes

```
Require: Candidates is a queue (not a set)
procedure FindChangedNodes(Candidates, Top, StateItem)
    Changed ← $\emptyset$
    while Candidates $\neq \emptyset$ do
        (C, Candidates) ← (head of Candidates, pop Candidates)
        if C = Top then                                        ▷ No Processing
        else if StateChange(IDOM(C), C, StateItem) then
            Changed ← $\{C\} \cup$ Changed
            Candidates ← Candidates $\setminus$IDOM($C$)
        else
            Candidates ← add IDOM(C) to end of Candidates
        end if
    end while
    return Changed
end procedure
```

- EmitVariable(X, Y, S) is a routine which returns a pair (Assignment, Variable) by assigning a variable for the state item S of node set X based on its current state, and then emitting the symbolic representation of the assignment of that variable in the state storage for node Y, the post-dominator, along with symbolic representation of the variable without the assignment.
- StateChange(X, Y, S) is a routine which determines if the stored state has a symbolic difference between nodes X and Y for state item S.

Algorithm 2 Find Minimal Set of Unchanged Nodes Relative to a Set of Changed Nodes

```
procedure FindNotChangedNodes(Changed, Candidates, Top, DominatorNodes)
    NotChanged ← $\emptyset$
    while Candidates $\neq \emptyset$ do
        NotChanged ← NotChanged $\cup \forall C \in$ Candidates, $C =$ Top $\vee$ IDOM(C) $\in$ DominatorNodes $\wedge$ IDOM(C) $\notin$ Changed
        Candidates ← $\{IDOM(C) | C \in$ Candidates, $C \neq$ Top, IDOM(C) $\notin$ DominatorNodes$\}$
    end while
    return NotChanged
end procedure
```

Algorithm 3 Variable Assignment Routines

```
procedure AssignVariable(X, StateItem)
    Nodes ← FindChangedNodes(PREDS(X), IDOM(X) StateItem)
    EmitVariable($\forall N \in$ Nodes $\cup$ FindNotChangedNodes(Nodes, PREDS(X) $\setminus$ Nodes, IDOM(X), $\bigcup\{DOM(C) | C \in$ Nodes$\}$), X, StateItem)
end procedure
procedure AssignVariables(X)
    $\forall$ StateItem $\in$ State, AssignVariable(X, StateItem)
end procedure
```

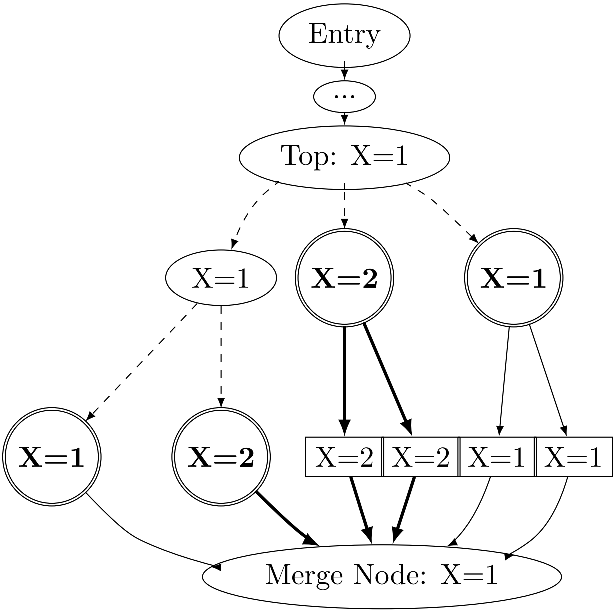


FIGURE 4. Example of the variable assignment algorithm

3.6. **Graph Correction, Clean up and Correctness.** The graph corrections required for a correct AST, which involve removing the meta-data annotations, and the list to tuple transformations of catch blocks and some function calls for receive which kept a list then tuple nesting structure must be fixed first and manually so a valid AST results. Next comes the various graph corrections required for compilable code, which are the empty values in case structure pathways where a single path went through due to an error in the other path. Any amount of sibling code which comes after can be considered to go into this area. This could be improved but more useful is a totally separate sequence for optimization. Based on this, what emerges is a dependency order for a minimal ambiguity in the cleanup actions.

Line numbers would be very difficult to match, and a novel approach would be needed for true generality.

A proof of correctness of the algorithm has a basis in that variable assignments are processed on merge nodes using a classical dominator-based approach only a single time when its decided the dominator tree for a merge node cannot further change. For the cross edges, they are handled only when they post-dominate some nodes or have an increase therein. The two processing orders will guarantee all predecessor edges recursively are known, so no cross edges will remain when the remaining set to process is empty.

4. Conclusion and Future Work

In this paper we presented a methodology to demonstrate that not only is incremental decompilation possible and feasible, but it can be practically implemented with good results. The technical considerations and details laid forth provide a framework for correct CFG structuring via binary conditionals, and can be extended to block structures like **catch** or **receive**, along with setting forth a code clean-up framework. Due to the close relationship between refactoring of code, and program transformation, this project is to become of the RefactorErl toolkit released by ELTE [11].

We have successfully evaluated and validated our methodology on the source of the Erlang/OTP libraries and the compiler test suite [12]. The details of the evaluation was presented in [13].

In the future, a study should deal with complicated incremental structuring induced by introduction of loops. They have their own theory and ambiguity in determining nesting, along with interference in conditionals studied here where multi-entry/exit loops are concerned. Decidability issues through the use of tools such as boolean satisfiability (SAT) solvers could be incorporated. As for Erlang, the possibility of writing obfuscators based on identified decompilation weaknesses is also an open challenge.

References

[1] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013. ISBN 978-1-93778-553-6.

[2] Ericsson AB. Erlang Programming Language. `http://www.erlang.org`, 2018. [Accessed: 2018.03.14].

[3] Dániel Lukács and Melinda Tóth. Structuring Erlang BEAM Control Flow. In *Proc. of the 16th ACM SIGPLAN International Workshop on Erlang*, Erlang 2017, pages 31–42, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5179-9.

[4] Cristina Cifuentes. *Structuring decompiled graphs*, pages 91–105. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-49939-8.

[5] G. Ramalingam and Thomas Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 287–296, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.

[6] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

[7] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Incremental computation of dominator trees. *ACM Trans. Program. Lang. Syst.*, 19(2):239–252, March 1997. ISSN 0164-0925.

[8] Paul F. Dietz. Maintaining order in a linked list. In *Proc. of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 122–127, New York, NY, USA, 1982. ACM. ISBN 0-89791-070-2.

[9] Paolo G. Franciosa, Giorgio Gambosi, and Umberto Nanni. The incremental maintenance of a depth-first-search tree in directed acyclic graphs. *Information Processing Letters*, 61(2):113 – 120, 1997. ISSN 0020-0190.

[10] Mihalis Pitidis and Konstantinos Sagonas. Purity in Erlang. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages*, pages 137–152, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24276-2.

[11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl - Source Code Analysis and Refactoring in Erlang. In *Proc. of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.

[12] Ericsson AB. Erlang/OTP (source code). `https://github.com/erlang/otp`, 2018. [Accessed: 2018.03.14].

[13] Gregory Morse. Towards a General Theory of Incremental Decompilation. TDK Thesis, Budapest, Hungary, May 2018.

Department of Programming Languages and Compilers, Faculty of Informatics, ELTE, Eötvös Loránd University, 1/C Pázmány Péter sétány, Budapest, 1117, Hungary

Email address: `morse@inf.elte.hu dlukacs@caesar.elte.hu toth_m@inf.elte.hu`

Appendix Appendix A Merge and Exit Incidence Algorithms

Algorithm 4 Algorithms to Process Merge Nodes

```
procedure HandleMerge(Node, IsExit)
Phase 1 - Add place holder node
    if Node = ReturnNode or IsExit then
        InsertNode ← ExitNode, SearchNode ← Node
    else if (CurrentNode, Node) ∈ NearestCrossPdom then
        InsertNode ← NextNode(), SearchNode ← InsertNode,
        insert_ast_node (Node, get_ast_node(Node), InsertNode)
        if CurrentNode ∈ PREDS(Node) then
            add_edge(CurrentNode, InsertNode), remove_edge(CurrentNode, Node)
        end if
        add_edge(Node, InsertNode), add_edge(InsertNode, ReturnNode)
        remove_edge(Node, ReturnNode)
    else
        InsertNode ← NextNode(), SearchNode ← Node,
    end if
Phase 2 - Resolve cross edges via code duplication
    CrossPairs ← $\forall (X, Y)$ ∈ CrossEdges, X ≠ SearchNode ∧ (NearestCrossPDom(X, Y) = Node
        ∨ NearestCrossPDom(X, Y) = InsertNode)
    while CrossPairs ≠ ∅ do
        (From, To) ← $\arg\max_{(X,Y)\in\text{CrossPairs}}$ ( get_ast_dfs (Y),  get_ast_dfs (X))
        CrossPairs ← CrossPairs \ {(From, To)}, NodeSet ← REACH(From) \ REACH(Node)
        copy_ast_node(X, Y, NodeSet), copy_graph_nodes(X, Y, NodeSet)
    end while
Phase 3 - Variable assignment
    AfterNode ← $\begin{cases} \text{Node} & \text{Node} = \text{ReturnNode} \lor \text{IsExit} \\ \text{InsertNode} & \text{otherwise} \end{cases}$
    if Node = ReturnNode then
        insert_ast_node (Node, AssignVariable(AfterNode, ReturnRegister))
    else if ¬ IsExit then
        set_ast_node( $\begin{cases} \text{Node} & \text{IsExit} \\ \text{InsertNode} & \text{otherwise} \end{cases}$ , AssignVariables(AfterNode))
    end if
    return AfterNode
end procedure
```

- insert_ast_node (Node, EmitValue, NewValue, NewNode) where EmitValue and NewNode are optional, must insert at the next available AST path after Node, EmitValue if present, and NewValue always, and if NewValue was a meta-data, then NewNode specifies the graph node path into which the meta-data path will be stored for later lookup or removal.
- insert_ast_node_child (Node, Kind, NewValue, NewNode) is identifical to the previous one except Kind gives an additional path information to be traversed based on what type of child is being added such as a conditional or block structure.
- get_ast_node(Node) fetches the node data specified.
- set_ast_node(Node, Data) replaces the data at the node specified.

Algorithm 5 Algorithms to Process Exit Nodes

```
procedure HandleExit
    NewPDoms ← PotentialNewPDoms
    while N doewPDoms $\neq \emptyset$
        $C \leftarrow \arg\min_{X \in \text{NewPDoms}}$ get_rev_dfs (X)
        HandleMerge(C, true), NewPDoms ← NewPDoms $\setminus C$
    end while
end procedure
```

- get_ast_dfs (X) returns the simple tree walk ordering of node X to allow a non-conflicting order when copying. In fact this is an approximation ordering that will not always work, without continuing to recompute nearest cross edge post dominators until a fixed point is reached. Technically there is a subgraph induced by FromPath which is copied to ToPath, and any of these subgraphs which contains a ToPath creates a dependency. The optimal scenario is therefore to avoid iterative calculation by adding all the cross edge pairs to a list in dependency order by not adding them until their dependencies are first added which also induces a proper partial ordering. It is a very important point as the algorithm simplifies and hides this fact.
- copy_ast_node(FromNode, ToNode, NodeSet) surgically copies all the contiguous set of the AST tree entries at the same level as FromPath and including it, which contains NodeSet, into ToPath and which must be inserted as new mapped entries for all the metadata that was copied into ToNode to maintain the integrity and consistency of the AST structure.

As for the graph, the following corresponding operations must be implemented and maintained:

- add_edge(X, Y) adds the edge from node X to node Y.
- remove_edge(X, Y) removes the edge between node X and node Y.
- get_bfs (X) gets a comparable breath first search ordering of node X for the scanning algorithm.
- get_rev_dfs (X) gets a comparable depth first search ordering of node X in the reverse graph rooted at ReturnNode.
- copy_graph_nodes(FromNode, ToNode, NodeSet) copies all the nodes in subgraph NodeSet replacing all edges between nodes in the set with the new nodes, and maintaining all edges which were to outside the subgraph except any nodes preceded by FromNode which are changed to ToNode. This is the corresponding operation to copy_ast_node which deals with the AST on copy.
- EntryNode symbolizes the entry node, ReturnNode symbolizes the return node, ExitNode symbolizes the exit node, and NextNode() symbolizes the next node which is to be newly added to the graph, and CurrentNode is the current node being processed or considered, while CurInst is the current instruction.
- PREDS, SUCCS, DOM, SDOM, PDOM, PSDOM, IDOM, PIDOM, REACH, REVREACH, CrossEdges, NearestCrossPdom(Node).
  PotentialNewPDoms ← $\forall C \in (\bigcup \forall X \in$ PredSetReach(Node, R_s), SPDOM(X)) \ [ReturnNode, ExitNode], get_processed(C)=Processed

Appendix Appendix B Overall and Scanning Algorithms

Algorithm 6 Sequential and Breadth First Oriented Scanning

```
procedure ContinueScan
    if Jumped and IsBFSScan then
        set_processed(GetNextLabel(), Processable)
        Candidates $\leftarrow \forall C \in$ PREDS(ReturnNode), get_processed(C) $=$ Processable
        if Candidates $= \emptyset$ then
            return ($\emptyset$, ReturnNode)
        else
            NextNode $\leftarrow \arg\min_{X \in \text{Candidates}}$ get_bfs(X)
            CollideNode $\leftarrow \forall C \in$ PREDS(NextNode), get_processed(C) $=$ Colliding
            return (GetInstructionAt(NextNode, $\begin{cases} \text{hd(CollideNode)} & \text{CollideNode} \neq \emptyset \\ \text{NextNode} & \text{otherwise} \end{cases}$)
        end if
    else
        return (GetNextInstruction(), CurrentNode)
    end if
end procedure
```

- get_processed(Node) returns either Unprocessed, Processable, Processed or Colliding.
- set_processed(Node, State) sets Node's processing status to State.
- SemanticEquivalence(C) provides the symbolic net effect on the state of instruction C.
- IsBranching(C) indicates if the C instruction is a branching or exit/exception instruction.
- IsLabel(C) indicates if the C instruction is a label and hence merging point.
- NodeFromLabel(C) returns an already mapped node for the label C, whether pre-existing or requiring a new graph node assignment.
- HasSideEffect(C) classifies if any side effect or other condition arises requires variable assignment.
- UpdateState(Data, StateDifference) updates the state in Data based on the difference.
- Output(C) gets the state.
- GetNextInstruction() gets the next instruction after instruction CurInst unless it is empty and then the entry instruction.
- GetNextLabel() scans forward after a jump for the next label to mark a node as not collided and hence processable.
- GetInstructionAt(Label) gets the instruction at location specified by Label.
- IsBFSScan represents the option for a breath first scan versus sequential.

Algorithm 7 Overall Decompile to AST

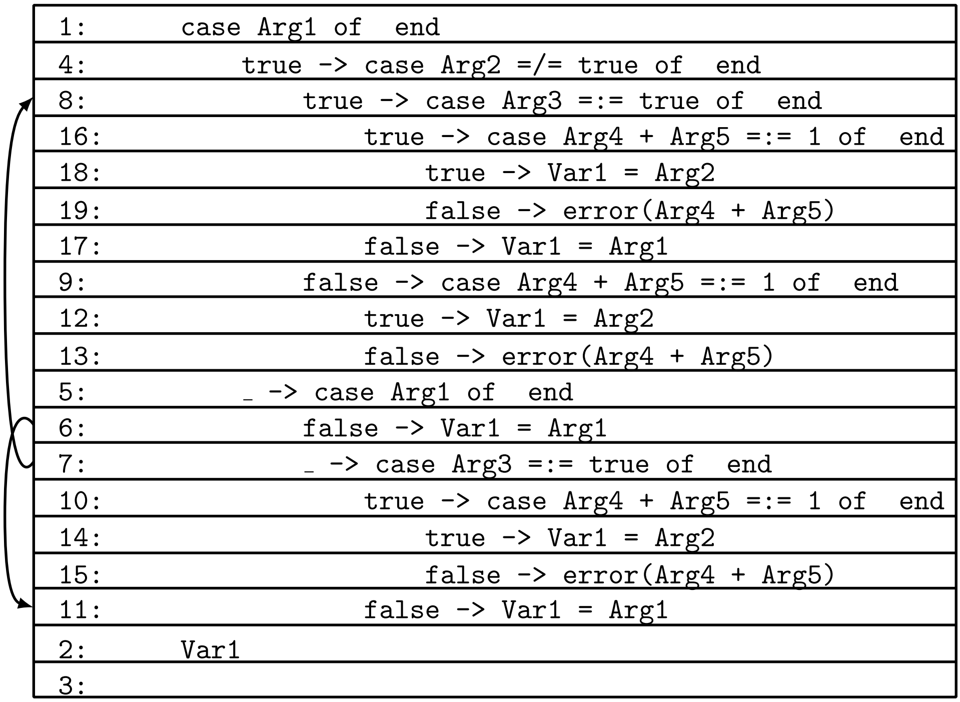
```
procedure DecompileToAST
    AST $\leftarrow$ [EntryMetadata], Graph $\leftarrow$ [EntryNode, ReturnNode, ExitNode]
    CurInst $\leftarrow \emptyset$, CurrentNode $\leftarrow$ EntryPoint, Jumped $\leftarrow$ false
    while (CurInst, CurrentNode) = ContinueScan(Jumped), CurInst $\neq \emptyset$ do
        if IsBranching(CurInst) then
            Jumped $\leftarrow$ DoBranchingStructuring()
        else if IsLabel(CurInst) then
            NewNode $\leftarrow$ NodeFromLabel(CurInst)
            if IsBFSScan $\wedge$ get_processed(CurrentNode) = Processed then
                set_processed(CurrentNode, Colliding)
            else
                add_edge(CurrentNode, NewNode), remove_edge (CurrentNode, ReturnNode), set_processed(CurrentNode, Processed), set_processed(NewNode, Processed), HandleMerge(NewNode, false)
            end if
        else if HasSideEffect(CurInst) then
            (Emit, NewVariable) $\leftarrow$ AssignVariable(SemanticEquivalent(CurInst))
            insert_ast_node(CurrentNode, Emit, UpdateState(get_ast_node(CurrentNode), NewVariable), CurrentNode)
        else
            set_ast_node(CurrentNode, UpdateState(get_ast_node (CurrentNode), SemanticEquivalent(CurInst)))
        end if
    end while
    if ExitNode $\notin$ PREDS(ReturnNode) then
        HandleMerge(ReturnNode)
    end if
    return Changed
end procedure
```

Appendix Appendix C Example

An overall example of a program in Figure 5 will highlight several of the key ideas with a special view of the final AST with super-imposed cross edge arrows in Figure 6 corresponding to the graph view of the prior step in Figure 7.

```
incstruct(A, B, C, D, E) ->
  if not A -> A;
  A andalso B orelse C ->
    if D + E =:= 1 -> B; true -> error(D + E) end;
  true -> A end.
```

Figure 5. Example code highlighting cross edge identification, merge node, code copying and variable assignment



```
1:       case Arg1 of  end
4:         true -> case Arg2 =/= true of  end
8:           true -> case Arg3 =:= true of  end
16:            true -> case Arg4 + Arg5 =:= 1 of  end
18:              true -> Var1 = Arg2
19:              false -> error(Arg4 + Arg5)
17:            false -> Var1 = Arg1
9:           false -> case Arg4 + Arg5 =:= 1 of  end
12:            true -> Var1 = Arg2
13:            false -> error(Arg4 + Arg5)
5:         _ -> case Arg1 of  end
6:           false -> Var1 = Arg1
7:           _ -> case Arg3 =:= true of  end
10:            true -> case Arg4 + Arg5 =:= 1 of  end
14:              true -> Var1 = Arg2
15:              false -> error(Arg4 + Arg5)
11:            false -> Var1 = Arg1
2:       Var1
3:
```

Figure 6. Example program after final structuring step at return node with arrows showing where copying occurred

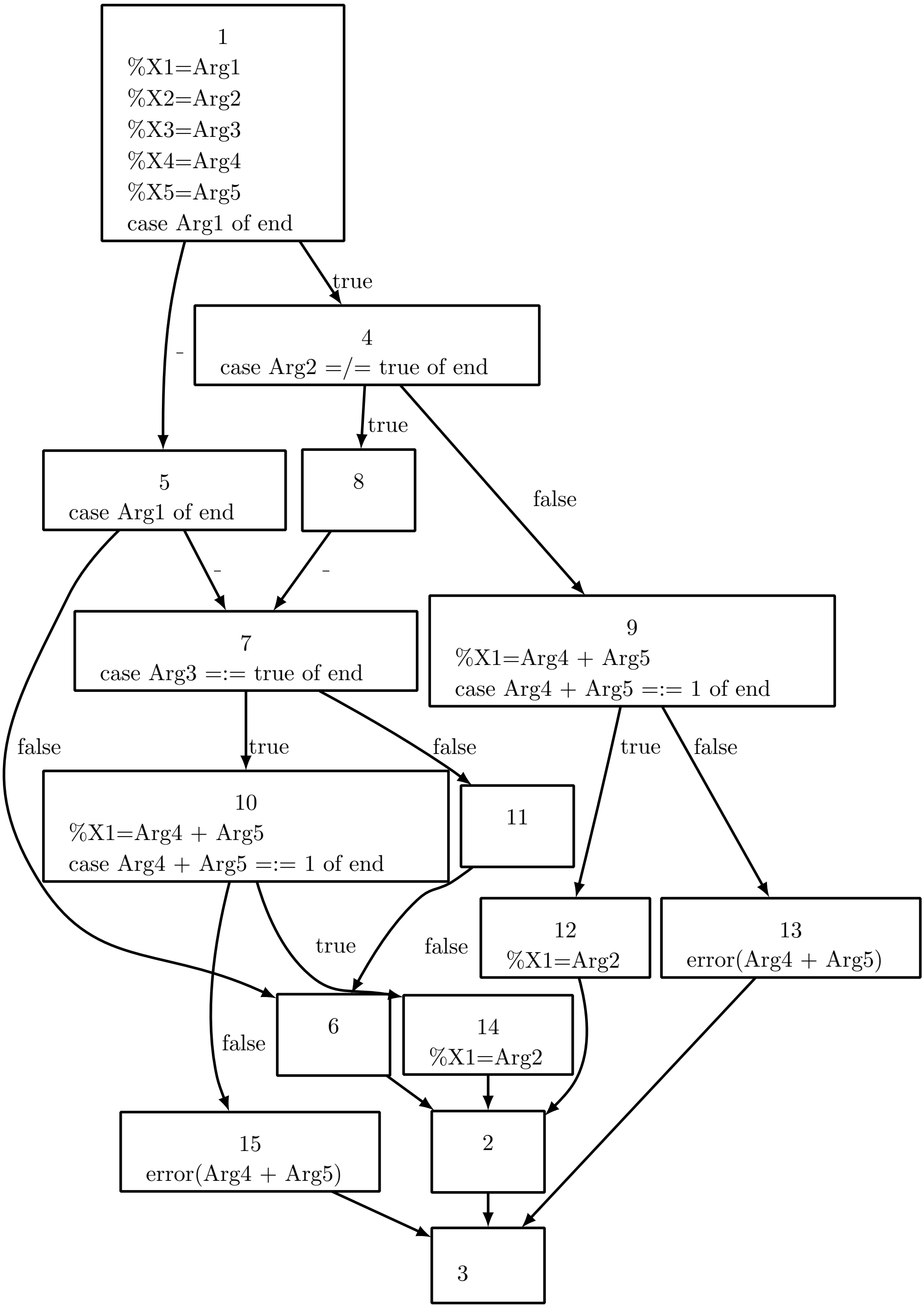


FIGURE 7. Graph derived from example program in final structuring step at return node