Semantic Reasoning about the Sea of Nodes

Delphine Demange
Univ Rennes / Inria / CNRS / IRISA
France

Yon Fernández de Retana
Univ Rennes / Inria / CNRS / IRISA
France

David Pichardie
Univ Rennes / Inria /CNRS / IRISA
France

Abstract

The *Sea of Nodes* intermediate representation was introduced by Cliff Click in the mid 90s as an enhanced Static Single Assignment (SSA) form. It improves on the initial SSA form by relaxing the total order on instructions in basic blocks into explicit data and control dependencies. This makes programs more flexible to optimize. This graph-based representation is now used in many industrial-strength compilers, such as HotSpot or Graal. While the SSA form is now well understood from a semantic perspective – even formally verified optimizing compilers use it in their middle-end – very few semantic studies have been conducted about the Sea of Nodes.

This paper presents a simple but rigorous formal semantics for a Sea of Nodes form. It comprises a denotational component to express data computation, and an operational component to express control flow. We then prove a fundamental, dominance-based semantic property on Sea of Nodes programs which determines the regions of the graph where the values of nodes are preserved. Finally, we apply our results to prove the semantic correctness of a redundant zero-check elimination optimization. All the necessary semantic properties have been mechanically verified in the Coq proof assistant.

CCS Concepts • **Software and its engineering → Formal language definitions**; **Compilers**; **Formal software verification**;

Keywords Intermediate Representation, Semantics, Sea of Nodes, SSA, Verified Compilation

ACM Reference Format:
Delphine Demange, Yon Fernández de Retana, and David Pichardie. 2018. Semantic Reasoning about the Sea of Nodes. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179503

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC'18, February 24–25, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179503

1 Introduction

Compiler designers seek for suitable program intermediate representations (IR) that will at the same time *simplify* the design of advanced program optimizations and also enable a very *efficient* implementation of these transformations.

An important breakthrough in the IR landscape is the Static Single Assignment (SSA) form, introduced by Rosen, Wegman and Zadeck [19] in the late 80s. Programs in SSA form have all their variables statically assigned exactly once. This apparently simple property makes program easier to analyze and optimize, as it makes unambiguous the link between the unique definition of a variable and its usage. In contrast, analyzing a non-SSA program requires to reason on all possible reaching definitions. SSA is now widely adopted in many modern compilers (e.g., GCC or LLVM) or code generators [13]. There, the SSA form is built on top of a control-flow-graph representation of the program, with basic blocks holding instruction lists.

Another important milestone in optimization-dedicated program IR is the Sea of Nodes, introduced by Cliff Click in his PhD thesis [7]. This IR was first implemented in the Java HotSpot [14] compiler – where it remains an important building block. It has inspired a lot of variants, in particular in the LibFirm [4] library, and the IR of the Graal framework [12].

In addition to be a SSA form, the salient property of the Sea of Nodes is that it makes program dependencies more explicit, while removing unnecessary constraints in the program control-flow graph. In particular, the total order between instructions of a basic block is relaxed. Instead, elementary instructions are linked to their *immediate* predecessors for data and control-flow dependencies. As an example, for a program fragment like **if** `b` **then** {`x = −a; y = c + 1`}, the assignment `y = c + 1` will *float* in a graph node linked only to the definition of the variable `c` (data dependence) and to another node representing the evaluation of the boolean condition `b` to `true` (control-flow dependence). Here, the program ordering between the assignments to `x` and `y` is unnecessary, and hence simply erased in the IR. It is only later in the compiler back-end that instructions are *scheduled*: instructions are ascribed a specific basic block, and a total order on instructions is decided within each basic block. As pointed at by Braun et al. [4], having in the IR dependencies only is what renders some optimizations simply *inherent* to the program. Some other fast, combined optimizations [6] can also be performed that directly operate on the data and control dependencies.

SSA in Verified Compilers. During the last ten years, the formal verification of realistic compilation techniques has become reality, inspired and piloted by the advent of the CompCert compiler [17], a formally verified C compiler. The SSA representation is arrived more lately in this landscape with the independent efforts around CompCertSSA [1] and Vellvm [23, 24]. The CompCertSSA [1] project uses a complete, verified validator, for the generation of pruned-SSA, based on dominance frontiers [9]. The Vellvm project proves in [23, 24] a simplified version of the LLVM SSA generation, based on register promotion. Recently, the generation algorithm proposed by Braun et al. [3] has been formalized in Isabelle/HOL [5]. Apart from the generation itself, some progress has also been made on the formalization of the useful invariants and properties of SSA that ease the reasoning when it comes to proving optimizations. For instance, [1, 11, 23] formalize the strictness semantic invariant, basic equational reasoning and dominance-region reasoning. These are semantic tools that allow proving formally the correctness of Sparse Conditional Constant Propagation and Common Subexpression Elimination based on Global-Value-Numbering [11], or Copy Propagation and micro memory optimizations [23].

Sea of Nodes Execution Model. The Sea of Nodes representation has not yet been used in a compiler verification context. In [7], Click proposes a Petri net execution model for the control part of its representation. It is only informally described with prose, and has not been, to our knowledge, manipulated in correctness proofs.

Most of the compiler verification literature advocates the use of operational semantics to ease the reasoning about compiler IRs and transformations in a proof assistant. The Petri net model from [7] does not suit this need, and we are not aware of any compiler optimization formalization that directly operates on a dependency graph a la Sea of Nodes.

Interestingly, the earliest mechanized formalization of the SSA form, dating back to the work of Blech et al. [2], investigates how to reflect data-dependencies in the semantics of SSA. Through the use of either *term graphs* or *relational sets*, they respectively try to capture syntactic dependencies inside expressions, or the partial order defining an evaluation strategy for SSA basic blocks. This early work is an interesting outlook on data-dependencies, but it is limited to basic SSA only (with instructions bound to their basic blocks), and they deal in their proof with a simple code generation from a *single* basic block.

In contrast, we want to explore the full-fledged Sea of Nodes from an *optimizing* compilation perspective. In particular, the "floating" nature of many nodes in the representation, while allowing for more flexibility for optimizations, raises several interesting questions. This includes designing a new semantics, necessarily different from the SSA semantics found in verified optimizing compilers, that should i) account for sharing between common subexpression graphs, and ii) reflect the floating nature of nodes. In turn, this new semantics calls for new semantic properties, and optimization proof techniques.

Contributions. In this work, we propose a formal semantics for a Sea of Nodes form, that could serve as a basis for its integration in a verified compiler infrastructure. This requires identifying the important semantic properties that this IR provides, and that justify formally the semantic correctness of program analyses and optimizations. More precisely, we make the following contributions:

- We define a semantics for a Sea of Nodes form (Section 2). First, the evaluation of data nodes uses an intuitive denotational semantics, that closely matches the natural interpretation of data nodes as expressions. Second, the main execution propagates from one region of the graph to the next according to a small-step, operational semantics, as a natural extension of the CompCertSSA [1] or Vellvm [23] semantics.
- We prove a fundamental, semantic property on Sea of Nodes graphs (Section 3): during program execution, the value given to a data node x keeps valid in all regions that are dominated by x.
- We illustrate this property by proving the semantic correctness of a fast redundant zero-check elimination algorithm that directly operates on the dominator tree of the Sea of Nodes program (Section 4).

Our formal development within the Coq proof assistant is available at http://www.irisa.fr/celtique/ext/sea-of-nodes/.

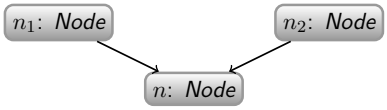
2 Sea of Nodes Syntax and Semantics

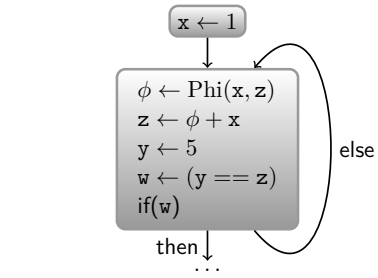
In this section, we give a syntax and semantics for a Sea of Nodes intermediate representation.

2.1 Syntax

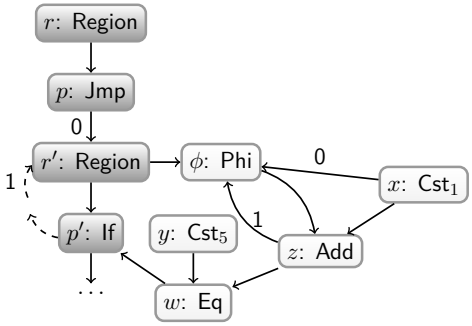
We represent a Sea of Nodes function with a directed graph g. In this graph, nodes represent instructions or auxiliary helpers for the control flow. The edges represent dependencies between nodes: an edge from a node n to a node n' means that n' depends on n. Intuitively, evaluating n' is necessary for evaluating n. The node n is called an *input* or *predecessor* of n', and n' is called an *output*, or *successor* of n.

Each node has a unique identifier in the graph, and is further defined by its kind (defined below), and the list of its input nodes. For a node with identifier n, if its kind is *Node*, and its inputs are n_1 and n_2, we will use the following graphical representation:





(a) Program control-flow graph



(b) Sea of Nodes graph. We indicate the indices of predecessors of phi-nodes and region nodes on arrow labels.

Figure 1. Example of an SSA program and its corresponding Sea of Nodes graph.

In the rest of the paper, to ease visual reading, we use different shades of gray to distinguish the kinds of nodes.

Illustrative Example. Before we go further into the technical definition of the Sea of Nodes IR, let us illustrate it on a simple example. Figure 1 gives an example of a simple SSA program in a basic-block control-flow graph (Figure 1a), and its corresponding Sea of Nodes graph (Figure 1b).

Let us focus on Figure 1a. The program consists of a loop incrementing an integer counter. The counter is initialized to 1 and incremented of 1 at each iteration, until it reaches the value 5. Note that the program is in SSA form. Indeed, each variable is assigned at most once. To ensure this single assignment property at control-flow join points, we recall that, in SSA, for a junction point with m predecessors, a phi-instruction $x \leftarrow \texttt{Phi}(x_1, x_2, \ldots, x_m)$ selects *at run-time*, according to the control-flow path executed, the right definition to use among all definitions x_i reaching that junction point. The corresponding x_k is then assigned to x. In the program example, the control-flow join point is at the loop header, and the phi-instruction $\phi \leftarrow \texttt{Phi(x,z)}$ is used to select the right definition of the counter (among x and z). Hence, when entering the loop body for the first time, x will be selected. Starting from the second iteration of the loop, z will be selected. To define the semantics of phi-instructions, one must consider, in one way or the other, that predecessors of a juntion point are ordered, so as to select the right argument of the phi-instruction.

Let us now describe Figure 1b, and highlight important aspects of the Sea of Nodes form. There are nodes for both data computations (light-grey nodes in Figure 1b) and for control-flow execution (dark-grey nodes in Figure 1b).

As an example of data nodes, z corresponds to the pseudo-code $\texttt{z} \leftarrow \phi + \texttt{x}$, its inputs are nodes ϕ and x. Similarly, w is a data node corresponding to the pseudo-code $\texttt{w} \leftarrow (\texttt{y} == \texttt{z})$.

Nodes related to control-flow execution are more involved. *Region* nodes are analogous to basic-blocks in SSA. In the example graph, there are two such nodes, r and r', one for each of the basic blocks in the pseudo-code program. Now, in order to transit from one region node to another, Sea of Nodes resorts on *control* nodes (p and p' nodes in the example). Control nodes mark the "end" of a region, similarly to branching instructions found at the end of a basic block.

Another point to note is that node ϕ has region r' as an input: we need this dependency edge to define the value computed by ϕ. As for the SSA form, predecessors of region nodes as well as data node inputs of Phi nodes are supposed to be ordered, and their rank is used to trigger the evaluation of the right Phi node input. In figures, for clarity, predecessors index appears as label on dependency edges (see Figure 1b). The dotted arrow from node p' to node r' abstracts an auxiliary control node, namely a *projection* node, that we explain in the next paragraph.

Apart from Phi nodes, data nodes do not depend, at least directly, on a region node (see e.g., data nodes x and y), and are hence considered as "floating" in the graph. In particular, no constraint says whether node x has to be evaluated before or after node y, unlike the pseudo-code version could have led us to think.

Formal Definition of Nodes Table 1 gathers the formal definition of a Sea of Nodes graph. A graph is a (partial) map from node identifiers, and each node carries, in its very syntax, the identifiers of its input nodes.

In the table, and in the rest of the paper, we rely on notational conventions to express the kind of a node in its identifiers (e.g., data nodes range over $x, y, z \ldots$, region nodes range over $r, r' \ldots$).

Data nodes represent instructions that compute a numerical value. These include:

- $\textbf{Cst}_N$ corresponds to a constant N. This node does not depend on a region node (it is floating).
- $\textbf{\textit{binop}}(x_1, x_2)$ corresponds to binary arithmetic operations. It has two data input nodes x_1 and x_2. We consider only arithmetic operations that do not raise exceptions (such as division by zero). These nodes do not depend, at least immediately, on a region node.
- $\textbf{Phi}(r, x_1, \ldots, x_m)$ corresponds to a phi-instruction. It depends on a region node r, and on m input data nodes

Table 1. Sea of Nodes: graph, node syntax, and notations

	Graph	
$g \in$	$id \hookrightarrow node$	graph
id =	$\mathbb{N}$	node identifier
	Nodes	
$node$ ::=	$data$	
	$region$	
	$control$	
	$branch$	
$data$ ::=	Cst_N	numerical constant
	$binop$	binary operation
	phi	
$binop$ ::=	Add(x,y)	addition
	Eq(x,y)	boolean equality test
phi ::=	$\text{Phi}(r, x_1, \ldots, x_m)$	ϕ-node
$region$ ::=	$\text{Region}(p_1, \ldots, p_m)$	region
$control$ ::=	$jump$	
	$cond$	
	$\text{Return}(r, x)$	return
$jump$ ::=	$\text{Jmp}(r)$	jump
$cond$ ::=	$\text{If}(r, x)$	conditional
$branch$ ::=	$\text{IfT}(if)$	then
	$\text{IfF}(if)$	else
	Notations	
id	$\ni x, y, z, w$	for $data$
	$\ni \phi$	for phi
	$\ni r$	for $region$
	$\ni c$	for $control$
	$\ni if$	for $cond$
	$\ni p$	for $jump$ or $branch$
	$\ni n, n'$	for $node$

$x_1, \ldots, x_m$. The dependence on r is intrinsic to SSA: the value of a Phi node is determined by the predecessor of r that lead there.

We turn now our attention to the description of nodes related to the control-flow of the program.

Region($p_1, \ldots, p_m$) a region node with m predecessors as input. Region predecessors are control nodes (see below). Only one region in the graph has no input: it is the entry point of the graph.

There are three kinds of *control* nodes (Jmp, If, Return). For each region node, there is exactly one control node that depends on it. Such a node marks the exit of the region and correspond to the branching instructions traditionally found at the end of a basic block.

Jmp(r) Control node depending on a region r and corresponding to a jump at the end of r to another region.

If(r, x) Control node depending on a region r and corresponding to a conditional branch with condition x.

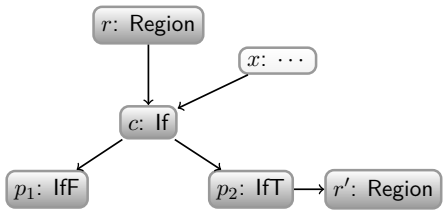


Figure 2. Conditional and Projections Nodes

Return(r, x) Control node depending on a region r. It terminates the execution of the function, returning the value of input data node x.

There are also two auxiliary branch nodes for `then` and `else` branches of an If node, the so-called projection nodes [8], whose purpose is to distinguish the two possible outcomes of the boolean condition of the If node. This is a syntactical method for handling the If result, which is intuitively multi-value, with one value for the `then` branch, and one for the `else` branch. It makes explicit the control dependence of the two If branches and it is useful for optimisations that want to analyse differently the `then` branch from the `else` branch. As proposed by Click [8], we do not attach this information by labeling outgoing edges, but with two extra nodes. This projection nodes where previously omitted in Figure 1 for clarity.

IfF(if), IfT(if)
Projection node depending on an If node if. An IfF node corresponds to the entry of the branch taken in the case where the If condition is `false`, and the IfT node corresponds to the case where it is `true`.

Let us illustrate this feature with Figure 2. The figure shows two regions r and r'. The control node depending on r is an If node. This node has two projection successors: IfF and IfT nodes. The evaluation of node x determines which projection will be evaluated. If this evaluation gives a `true` value, then the IfT node is evaluated, and as a result execution will move from region r to region r'.

Well-formedness conditions. The syntactic definitions we give in Table 1 do not reflect all the syntactic constraints that a Sea of Nodes graph should satisfy to have a well-defined semantics. We list now these well-formedness conditions.

SSA-constraints In standard SSA form, the arity of the ϕ operators found at entry of a basic block must be equal to the number of incoming edges to this block. In Sea of Nodes, as shown in Figure 1b, this constraint is now expressed by the fact that for each ϕ node $\text{Phi}(r, x_1, \ldots, x_m)$ that depends on a region node r, the number of predecessors of r is equal to m, the number of ϕ's input data nodes.

Determinism Each region exits must be unambiguous: for each control node c, there must exist a unique successor region r' that depends on c. For a conditional control node *if*, this condition is *chained* in the following way: there exists exactly one IfT branch node and one IfF branch node that depend on *if* and both of them have unique successor regions.

Well-founded data dependencies The sub-graph of data dependencies between non-Phi data nodes must be acyclic. Cycles would prevent the evaluation of data node values.

Strictness The classical SSA strictness propery must still hold here: each variable use must be dominated by the unique definition of that variable. In Sea of Nodes, the meaning of *use*, *dominated* and *definition* must be adapted but the property remains crucial to ensure that each data node can be given a value during execution.

Extensions In this paper, we do not model function calls. Their treatment is routine, and could be done as in any other intermediate representation. We hence only consider programs with one single function.

In the paper, we also omit for clarity operations related to memory, although the Coq development handles them. Sea of Nodes requires to handle memory loads and stores with an SSA discipline. We hence introduce a specific memory value which represents the different versions of the memory during execution and explictly assigns SSA variables that represent the current value of the memory after each load and store. These two extra instructions do not alter the results we present in this paper.

2.2 Semantics

We now give a semantics to our Sea of Nodes IR. We use a denotational semantics to evaluate data nodes, while we use a small-step semantics for propagating execution from one region node to another. We fix a Sea of Nodes function with graph g, and all nodes are nodes of this graph g.

Environment. The denotational evaluation of data nodes makes use of an environment ρ, a partial mapping from ϕ-node to values. Indeed, ϕ-nodes are the only data nodes whose value depends directly on the control-flow, and as such they require special treatment. Other data nodes (constants and binary operations) values do not need to be stored in the environment ρ.

The small-step semantics updates the environment each time it reaches a junction point: for each ϕ-node of the target region, its value is updated, in a spirit similar to that of the CompCertSSA [1] semantics.

Notations. In the rest of the section, we refer to a numerical value returned by an instruction corresponding to a data node by v. The special values `false` and `true` are aliases for 0 and 1.

Execution states of a function, written σ, are of the form Start (initial state), Exec(r, ρ) (intermediate execution state), or Ret(v) (return state), where:

- r represents the current region node
- ρ is a partial mapping from ϕ-nodes to values
- v is the return value of the function

We use the following notation too:

$\mathrm{op}_{\boldsymbol{binop}}(v_1, v_2)$**:** underlying function of a *binop* node. For example, if the node is an Add, this function is the binary arithmetic operator +.

Data node denotational semantics. The evaluation of data nodes closely matches the natural interpretation of data nodes as expressions in Sea of Nodes. Indeed the sub-graph rooted in a data node x, and composed of data-dependency edges of non-Phi data nodes, is a directed acyclic graph (dag), where constants and ϕ-nodes are leaves, and *binop* nodes play the role of internal nodes. As shown in Figure 1b, the sub-graph rooted in w is composed of all light-grey nodes. This dag structure exactly matches the classic expression representation, but with sub-tree sharing. The ϕ-nodes should be seen as the variables of such an expression.

Thanks to the data dependency well-foundedness constraint we explained in Section 2.1, we can recursively compute the value $[\![x]\!]_\rho$ of a data node x, using the values of ϕ-nodes in the current environment ρ.

$$[\![x]\!]_\rho = \begin{cases} N & \text{if } g(x) = \mathrm{Cst}_N \\ \rho(x) & \text{if } g(x) = \mathrm{Phi}(\ldots) \\ \mathrm{op}_{binop}([\![x_1]\!]_\rho, [\![x_2]\!]_\rho) & \text{if } g(x) = binop(x_1, x_2) \end{cases}$$

For example, for the node w of Figure 1b we obtain:

$$[\![w]\!]_\rho = (5 == \rho(\phi) + 1)$$

The data dependency well-foundedness ensures the termination of the computation of $[\![x]\!]_\rho$, but we also need to ensure that, for all ϕ belonging to x's expression-dag, $\rho(\phi)$ is defined. This is where the strictness assumption comes into play: during a function execution, in every region where we will need to evaluate $[\![x]\!]_\rho$, the regions where all these ϕ nodes belong will already have been reached. Therefore, those ϕ nodes will be defined in ρ.

Operational semantics for region evaluation. We now describe the execution of a Sea of Nodes function with graph g using an operational semantics $\rightarrow_{\text{STEP}}$. The relation takes the form $\sigma \rightarrow_{\text{STEP}} \sigma'$, which corresponds to evaluating a region, and moving execution from one state to another.

The environment ρ in a state Exec(r', ρ) must be updated when reaching a region r with more than one predecessors (junction point with a list of ϕ-instructions). This update uses a parallel evaluation semantics, and affects all the ϕ-nodes that depends on region r.

Formally, we write $p, r, \rho \to_\phi \rho'$ the judgment that specifies how an environment ρ is updated into a new environment ρ' when the execution reaches region r, coming from predecessor p of r. The judgment is defined by the following rule:

$$\text{PHIS}\ \frac{\begin{array}{c}\text{philist}(r) = [\phi_1, \ldots, \phi_m] \qquad \text{index}(p, r) = k \\ \forall i = 1, \ldots, m,\ \text{ntharg}(\phi_i, k) = x_i \qquad [\![x_i]\!]_\rho = v_i\end{array}}{p, r, \rho \to_\phi \rho[\phi_i \mapsto v_i \text{ for all } i]}$$

If n is an input of n' in the graph, we write $\text{index}(n, n')$ the index of n among the list of inputs of the node n'. We write $\text{ntharg}(\phi, k)$ the k-th input data node of a Phi node ϕ, and $\text{philist}(r)$ the list of Phi nodes that directly depend on r (i.e., r is their first input).

The rule reads as follows: we update each Phi node that depends on r with the value computed for their k-th input data node, where k is the index of the predecessor p among the inputs of region r.

In the example of Figure 1, this rule is used to successively update the environment at the loop header. When execution arrives from p to r' initially, we have $\text{index}(p, r') = 0$, so environment is updated so that $\rho_0(\phi) = [\![x]\!]_\rho = 1$. Then, while $[\![w]\!]_{\rho_i} = \texttt{false}$, execution loops from p' to r'. Because in this case $\text{index}(p', r') = 1$, we successively get for the next three iterations: first $\rho_1(\phi) = [\![z]\!]_{\rho_0} = 2$, then $\rho_2(\phi) = [\![z]\!]_{\rho_1} = 3$ and finally $\rho_3(\phi) = [\![z]\!]_{\rho_2} = 4$.

Updating the values of ϕ nodes when reaching the region where they belong is a design choice that allows for a non-instrumented state: if environments were updated at the beginning of a region evaluation, it would be necessary to keep track of the region's predecessor in execution states.

We now describe in detail the different rules for the operational semantics $\to_{\text{STEP}}$. We write $\text{entry}(g)$ to refer to the entry point of the graph g. To alleviate notations, we simply write r, ρ for execution states of the form $\text{Exec}(r, \rho)$, as it does not incur any possible confusion.

The first rule, START, corresponds to the start of execution of the function, and initializes the state: the starting region is $\text{entry}(g)$ and the starting environment is the partial function ρ_{empty}, whose domain is empty.

$$\text{START}\ \frac{\text{entry}(g) = r_0}{\text{Start} \to_{\text{STEP}} r_0, \rho_{\text{empty}}}$$

Next, we present rules for intermediate execution states. The common pattern of these rules is the following: i) evaluate the control node of the current region ii) determine the successor region iii) update the environment ρ on the ϕ nodes of that successor region.

In the JMP rule, the control node of r is a Jmp node. In this case, the step simply goes to the successor region r' – remember it is uniquely determined, according to our well-formedness conditions.

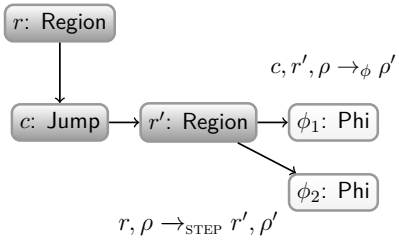


Figure 3. JMP Rule Illustration

$$\text{JMP}\ \frac{\begin{array}{c}g(c) = \text{Jmp}(r) \qquad g(r') = \text{Region}(\ldots, c, \ldots) \\ c, r', \rho \to_\phi \rho'\end{array}}{r, \rho \to_{\text{STEP}} r', \rho'}$$

The environment update can be applied because we know from which precedecessor of r' execution comes, in this case from c (second premise). This pattern for updating the environment is found in every rule for control nodes with successors, i.e., it doesn't apply for Return nodes.

The JMP rule is illustrated in Figure 3. The figure shows two region nodes r and r'. Region r has a Jmp as control node, that induces evaluation of r'. Then environment is updated for ϕ-nodes depending on r'.

The IFF and IFT rules apply when the control node of the current region is an If node with projection nodes IfF and IfT. Both rules are similar to the JMP rule, except that the environment update considers the projection node, rather than If node, as the predecessor of the next region.

$$\text{IFF}\ \frac{\begin{array}{c}g(if) = \text{If}(r, x) \qquad [\![x]\!]_\rho = \texttt{false} \\ g(p) = \text{IfF}(if) \qquad g(r') = \text{Region}(\ldots, p, \ldots) \\ p, r', \rho \to_\phi \rho'\end{array}}{r, \rho \to_{\text{STEP}} r', \rho'}$$

$$\text{IFT}\ \frac{\begin{array}{c}g(if) = \text{If}(r, x) \qquad [\![x]\!]_\rho = \texttt{true} \\ g(p) = \text{IfT}(if) \qquad g(r') = \text{Region}(\ldots, p, \ldots) \\ p, r', \rho \to_\phi \rho'\end{array}}{r, \rho \to_{\text{STEP}} r', \rho'}$$

Figure 4 illustrates the rule IFT. The figure shows two region nodes r and r'. The first has an If as control node. The node x evaluates to `true`, so the IfT branch is evaluated. Then environment is updated for ϕ-nodes depending on r'.

Finally, there is the RET rule, applicable when the control node of the current region is a Return. It corresponds to a function return, and the value of the input data node is returned as part of the state.

$$\text{RET}\ \frac{g(c) = \text{Return}(r, x) \qquad [\![x]\!]_\rho = v}{r, \rho \to_{\text{STEP}} \text{Ret}(v)}$$

Function execution. We have a single START rule, and a rule RET for function return. The execution of a function is thus of the form $\text{Start} \to_{\text{STEP}} \cdots \to_{\text{STEP}} \text{Ret}(v)$.

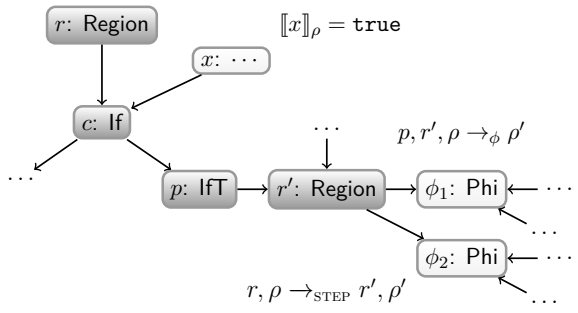


Figure 4. IFT Rule Illustration

In this paper, to alleviate the presentation, we use the return value as the unique observable behavior of a terminating execution of a function. The proofs presented here can naturally be extended to lists of observable events, as would be produced by e.g., a Print instruction.

3 Validity Domains for Values

Many compiler optimizations, e.g., GVN [19], constant propagation, or redundant null-pointer checks elimination [16] require reasoning, statically, about the possible values that variables or expressions may have at certain places in the program. In practice, such optimizations rely on the control-flow structure of programs, using notions like dominance, in order to ease the description of where, in the program, a particular value is valid, or preserved.

In this section we give, in terms of dominance, a fundamental property that is useful for such reasonings in the context of a Sea of Nodes IR. In Section 4, we illustrate how this property allows to prove the semantics preservation of a redundant zero-check elimination.

3.1 Value Preservation Property

We now give the conditions under which the value of a data node is preserved along an execution path. We fix a Sea of Nodes function with graph g from now on.

Phi dependencies of a data node. We use the notation $\text{phideps}(x)$ to refer to the set of all ϕ-nodes appearing in the expression associated to x. More formally:

$$\text{phideps}(x) = \begin{cases} \emptyset & \text{if } g(x) = \text{Cst}_N \\ \{x\} & \text{if } g(x) = \text{Phi}(\ldots) \\ \text{phideps}(x_1) \cup \text{phideps}(x_2) & \\ & \text{if } g(x) = \mathit{binop}(x_1, x_2) \end{cases}$$

In the example graph of Figure 1, we have $\text{phideps}(x) = \emptyset$, $\text{phideps}(y) = \emptyset$, $\text{phideps}(z) = \{\phi\}$, and $\text{phideps}(w) = \{\phi\}$.

The following lemma states that the value of a data node only depends on the value of the Phi nodes it depends on.

Lemma 3.1. *Let x be a data node of a graph g. Let ρ and ρ' be two environments. If for all $\phi \in \text{phideps}(x)$ we have $\rho(\phi) = \rho'(\phi)$, then $[\![x]\!]_\rho = [\![x]\!]_{\rho'}$.*

Proof. First we note that the hypothesis, thanks to well-formedness of data dependencies, ensures that $[\![x]\!]_\rho$ is defined, because all $\rho(\phi)$ for $\phi \in \text{phideps}(x)$ are defined. The proof is done by induction on the definition of $[\![x]\!]_\rho$: the base case of induction is ensured by the fact that both evaluations match on ϕ-nodes. □

The next lemma states that the value of a data node x is preserved as long as execution does not move to a region whose corresponds ϕ nodes belong to $\text{phideps}(x)$.

Lemma 3.2. *Let*

$$\text{Exec}(r_0, \rho_0) \cdots \rightarrow_{\mathit{STEP}} \text{Exec}(r_m, \rho_m)$$

be an execution path. Let x be a data node such that $[\![x]\!]_{\rho_0}$ is defined. Assume that, for all $\phi \in \text{phideps}(x)$, and for all $i = 1 \ldots m$, $\phi \notin \text{philist}(r_i)$. Then $[\![x]\!]_{\rho_0} = [\![x]\!]_{\rho_m}$.

Proof sketch. We prove by induction on the execution path from $\text{Exec}(r_0, \rho_0)$ to $\text{Exec}(r_m, \rho_m)$, that for all $\phi \in \text{phideps}(x)$, $\rho_0(\phi) = \rho_m(\phi)$. The main argument involves noticing that in applications of the PHIS rule along the execution path, no $\phi \in \text{phideps}(x)$ is involved, thanks to the hypothesis. Lemma 3.1 allows to conclude the proof. □

3.2 Dominance-Based Formulation

Dominance is a useful notion to describe control-flow properties when proving the correctness of some optimizations, in particular those involving elimination of redundant computations or run-time checks. In this section we define such a notion for our Sea of Nodes representation, and reformulate Lemma 3.2 in terms of dominance, so that it is easier to apply in proofs of optimizations.

First, we give an analogue of the standard basic-block CFG on Sea of Nodes.

Definition 3.3 (CFG)**.** We call CFG of a Sea of Nodes graph g, the directed graph whose nodes are region nodes, and where there is an edge from node r to node r' if the control node depending on r is:

- a Jmp node whose output is r',
- or an If with an output projection node that has r' as output.

This allows to define on the Sea of Nodes CFG an analogue to the standard dominance relation as follows:

Definition 3.4 (Dominance)**.** A region node r dominates a region node r' if every path in the CFG from the entry point of g to r' passes through r. The dominance is strict if $r \neq r'$.

This dominance relation applies to region nodes only : it is defined at a coarser level than what is done in traditional basic-block CFG. Below, we extend this notion so that it

handles data nodes as well. Informally, for a data node to dominate a region node r, all of its ϕ dependencies should belong to regions that dominate r. Formally:

Definition 3.5. (Data node dominance) A data node x *dominates* a region node r if, for all ϕ and r'_ϕ such as $\phi \in \text{phideps}(x)$ and $g(\phi) = \text{Phi}(r'_\phi, \ldots)$, we have that r'_ϕ dominates r. The dominance is strict if $r'_\phi \neq r$ for all such ϕ.

Dominance and evaluability. In the literature on SSA formalization, the so-called equational lemma from [1] is of particular importance. It states that the equation defining a variable x is valid at the points dominated by the definition point of x. Here, in a Sea of Nodes function, data nodes (excluding Phi nodes) have no real definition point – they float. Yet, this particular property, the equational lemma, is in fact baked in the data node evaluation semantics: data node values are always valid, so to say. They only need to be defined. The following lemma allows to characterize, using dominance, the regions in a Sea of Nodes graph where a data node is indeed defined.

Lemma 3.6. *Let* $\text{Exec}(r, \rho)$ *be a reachable state. Let* x *be a data node such that* x *dominates* r*. Then* $[\![x]\!]_\rho$ *is defined.*

Proof sketch. Let π be an execution path from state Start to state $\text{Exec}(r, \rho)$.

We prove first the property for a node ϕ with $g(\phi) = \text{Phi}(r', \ldots)$. The fact that ϕ dominates r means in this case that r' dominates r. Hence, there exists ρ' such that a state $\text{Exec}(r', \rho')$ appears in π. As a result, our semantics for ϕ-node evaluation when coming to region r' ensures that $\rho'(\phi)$ is defined, and since the environment updating semantics only extends the domain of environments during execution, $\rho(\phi)$ is also defined.

Now, for the other forms of data node x, it is true that for every $\phi \in \text{phideps}(x)$, $\rho(\phi)$ is defined, because, by definition, such a ϕ dominates r too. We conclude the proof with well-foundedness of data node dependencies. □

Note that this specific lemma is not strictly necessary for the other semantics proofs of the paper, and has not been mechanically verified.

Application to value preservation. We obtain the following corollary of Lemma 3.2 for value preservation in terms of dominance.

Theorem 3.7. *Let*

$$\text{Exec}(r_0, \rho_0) \cdots \rightarrow_{STEP} \text{Exec}(r_m, \rho_m)$$

be an execution path. Let x *be a data node such that* $[\![x]\!]_{\rho_0}$ *is defined. If, for all* $i = 1 \ldots m$*,* x *strictly dominates* r_i*, then* $[\![x]\!]_{\rho_0} = [\![x]\!]_{\rho_m}$*.*

Proof sketch. Because x strictly dominates r_i, we know that for all $\phi \in \text{phideps}(x)$, we have $\phi \notin \text{philist}(r_i)$. We can therefore apply Lemma 3.2. □

This theorem states that, as long as execution remains in a particular area of the graph, the area dominated by x, the value of x remains unmodified. As an example, an application of this result to prove semantics preservation of a redundant zero-check transformation can be found in the next section.

Theorem 3.7 is a little less general than Lemma 3.2 because it is restricted to execution path where each region is under the dominance of a given data node x, while executions in Lemma 3.2 can continue outside the dominance zone of x, as long as they do not reach ϕ-nodes in conflict with $\text{phideps}(x)$. But we believe the dominance characterization used in this theorem is closer to compiler designer intuition.

4 Redundant Zero-Check Elimination

Languages with rich and precise exception mechanisms often require the injection in the compiler IR of numerous explicit checks – for null-pointers typically, to guarantee a safe execution of the program. This dramatically reduces the scope program optimizations (and impacts the program execution running time). Redundant checks elimination allows to mitigate this issue.

In this section, we illustrate how to justify formally the semantic correctness of a redundant zero-check elimination on the Sea of Nodes form, using the results of previous sections.

4.1 Language Extension: ZeroCheck Nodes

We extend our Sea of Nodes form with a new control node, ZeroCheck(r, x). It executes like a jump to its successor, unless its input data node x evaluates to zero. In this case the execution goes to an error state Ret(`fail`). Here, `fail` is a new special value, introduced to model failure during program execution.

This gives the following two semantic rules:

$$\text{ZCHK1}\ \frac{g(c) = \text{ZeroCheck}(r, x) \quad g(r') = \text{Region}(\ldots, c, \ldots) \quad c, r', \rho \rightarrow_\phi \rho' \quad [\![x]\!]_\rho = v \neq 0}{r, \rho \rightarrow_{\text{STEP}} r', \rho'}$$

$$\text{ZCHK2}\ \frac{g(c) = \text{ZeroCheck}(r, x) \qquad [\![x]\!]_\rho = 0}{r, \rho \rightarrow_{\text{STEP}} \text{Ret}(\texttt{fail})}$$

Figure 5 gives an example of Sea of Nodes graph with two ZeroCheck nodes c_1 and c_3. This example graph has a loop and three conditionals. We use it in this section to illustrate the presentation.

4.2 Criteria for Redundancy

Deciding whether a ZeroCheck is redundant or not is equivalent to deciding if an arbitrary data node never evaluates to zero, which is undecidable in general.

Therefore, we need an approximation. Here, we use the following structural criteria for redundancy:

Definition 4.1. A region r has a *redundant* ZeroCheck if the following holds: r is a region node whose control node is

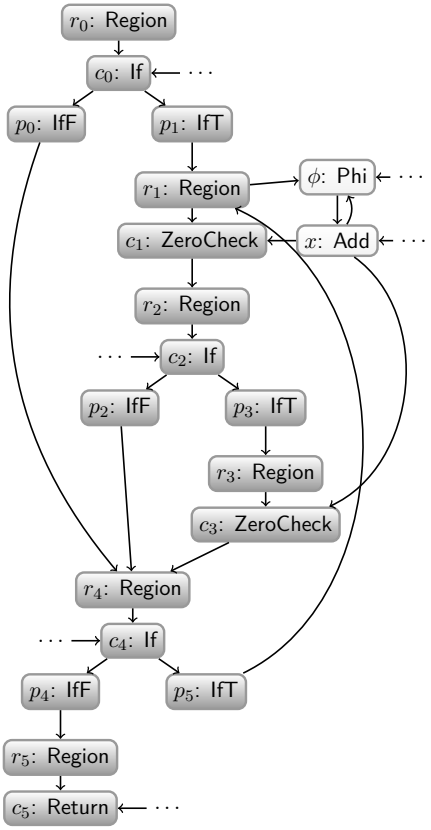


Figure 5. Dominance and Redundant ZeroCheck Illustration

a ZeroCheck(r, x), and there exists a region node r' strictly dominating r whose control node is a ZeroCheck(r', x).

For example, in Figure 5, the criteria is satisfied for r_3: indeed, it is strictly dominated by r_1, and both region nodes have a ZeroCheck control node whose input data node is x.

Value preservation applied to zero-checks. The results of the previous section on validity domains for values can be used to justify the intuition behind our criteria. We have the following corollary of Theorem 3.7:

Corollary 4.2. *Let* Exec(r', ρ') *be a reachable execution state. Let* r *be a region node whose control node is a* ZeroCheck(r, x). *If* r *strictly dominates* r', *then* $[\![x]\!]_{\rho'} \neq 0$.

Proof sketch. By hypothesis, there exists an execution path π to state Exec(r', ρ'). Because r dominates r', there exists a state in π whose region is r. Let us consider the most recent state, in π, whose region node is r. Namely, we can consider the end π' of π:

$$\text{Exec}(r, \rho) \cdots \rightarrow_{\text{STEP}} \text{Exec}(r', \rho')$$

such that for every other state Exec(r'', ρ'') after Exec(r, ρ) in π', $r'' \neq r$. The following fact holds: for every such r'', r dominates r''. Indeed, any path π_1 from the entry point to r'' can be extended to a path $\pi_1 \cdot \pi_2$ from the entry point to r', where π_2 is a suffix of π'. Since r dominates r', r belongs to $\pi_1 \cdot \pi_2$, but r does not belong to π' (neither to π_2), so r belongs to π_1. It proves that r dominates r''.

Moreover, well-formedness properties ensure that a node dominates its usages, so in particular x dominates r. Hence, x dominates strictly any such r'' (dominance is a transitive relation). Because there is a ZeroCheck at r and execution does not end there, ZCHK1 was applied, so $[\![x]\!]_\rho \neq 0$. By Theorem 3.7, we have $[\![x]\!]_\rho = [\![x]\!]_{\rho'}$. Therefore, $[\![x]\!]_{\rho'} \neq 0$. □

This corollary gives hypotheses, in terms of dominance and reachability, under which we know a node evaluates to a non-zero value for a particular environment. If r' has a ZeroCheck as control node, the configuration is exactly that of our redundancy criteria: the ZeroCheck of r' is redundant. This corollary is indeed key to prove formal semantics preservation (subsection 4.4).

4.3 Redundant Zero-Check Detection Algorithm

The algorithm we consider here is a recursive variant of the one found in the Go compiler [18, 20]. It is based on a single traversal of the dominator tree. We just adapted it to fit our IR. In particular, the dominator tree encodes the dominance relation between region nodes of a Sea of Nodes graph. We define it precisely now.

Definition 4.3 (Immediate Dominator)**.** A region node r is an immediate dominator of a region node r' if $r \neq r'$, r dominates r', and r does not dominate any other dominator of r'.

For example, in Figure 5, node r_2 is an immediate dominator of node r_3, but r_1 is not, because r_1 dominates r_2.

A standard property states that there is at most one immediate dominator for every node. This is the basis for ensuring that the dominance relation among the nodes of a graph can be encoded in a tree structure.

Definition 4.4 (Dominator tree)**.** The dominator tree of the CFG of a Sea of Nodes graph is a tree where nodes are the region nodes, and where the children of a node are the region nodes it immediately dominates.

The algorithm we consider is given in Figure 6. In a nutshell, the algorithm performs a single depth-first traversal of the dominator tree (`domtree`(g)), maintaining two maps from nodes to booleans. The first one, `nonZero`, keeps track of whether data nodes are non-zero (`true`) or may evaluate to zero (`false`). The second map, `redundantZC`, flags, for each region nodes, whether they have a redundant ZeroCheck control node.

Both maps are initialized to `emptymap(node, bool)`, i.e., all nodes are mapped to `false`. Then, during the tree traversal, each time a region with a ZeroCheck control node is encountered, the ZeroCheck node is flagged as being non-zero *in the dominator subtree* (i.e., the region nodes it dominates).

```
1  Analyse(g):
2    sdom ← domtree(g)
3    nonZero ← emptymap(node, bool)
4    redundantZC ← emptymap(node, bool)
5    DepthFirst(r):
6      if control(g, r) matches ZeroCheck(r, x):
7        nz ← nonZero[x]
8        redundantZC[r] ← nz
9        nonZero[x] ← true
10       for r' in sdom.successors(r):
11         DepthFirst(r')
12       nonZero[x] ← nz
13     else:
14       for r' in sdom.successors(r):
15         DepthFirst(r')
16   DepthFirst(entry(g))
17   return redundantZC
```

Figure 6. Redundant ZeroCheck Detection

In the algorithm, this is done with a three-step process. First, we save the current value of `nonZero[x]` (Line 7). Second, we set `nonZero[x]` to `true` (Line 9) and explore the subtree (Line 11). Finally, we restore the old value of `nonZero[x]` once subtrees have been traversed (Line 12). When traversing the subtree, any ZeroCheck node on the *same* data node will in turn be flagged as redundant (Line 8).

For example, in the case of Figure 5, when processing node r_1, a ZeroCheck(r_1, x) node c_1 is found with input x. The algorithm then recalls this information when processing the nodes dominated by r_1, that is r_2 and r_3. While processing r_3, the algorithm finds a ZeroCheck(r_3, x) node c_3 with the same input x, it knows it is redundant according to our criteria.

We prove that the algorithm in Figure 6 indeed computes our redundancy criteria for region nodes.

Lemma 4.5. *Let* `redundantZC = Analyse(g)` *and* r *such as* `redundantZC[r] = true`. *Then* r *has a redundant* ZeroCheck *in the sense of Definition 4.1.*

The reciprocal of the previous lemma holds too: if a region node with a ZeroCheck control node satisfies our dominance-based redundancy criteria, then the analysis detects it. For instance, on the graph in Figure 5, we obtain `redundantZC`$[r_3]$ = `true`. More formally:

Lemma 4.6. *Let* `redundantZC = Analyse(g)`. *If* r *has a redundant* ZeroCheck, *then* `redundantZC[r] = true`.

4.4 ZeroCheck Elimination: Semantic Correctness

Let `redundantZC = Analyse(g)`. The redundant ZeroCheck Elimination optimization proceeds by replacing, in the graph g, every ZeroCheck(r, x) for which `redundantZC[r] = true` with a simple Jmp(r) node. We write g' the graph resulting from this transformation. In the case of Figure 5, the analysis gives us that `redundantZC`$[r_3]$ = `true`, so g' is obtained from g by changing the node c_3 by Jmp(r_3).

We prove that this transformation is correct semantically. Namely, that the initial and transformed graphs have the same semantics. We write $\rightarrow_{\text{STEP}_g}$ to mean that the step is done in within a graph g. The theorem we prove is as follows.

Theorem 4.7. *We have* Start $\cdots\rightarrow_{\text{STEP}_g}$ Ret(v) *if, and only if,* Start $\cdots\rightarrow_{\text{STEP}_{g'}}$ Ret(v).

We prove the above theorem by establishing a simulation relation between the two graphs. In the case of the Redundant ZeroCheck Elimination, the link between both execution states is strong: both states are simply equal at each step of the execution.

Given a state σ, the predicate reachable$_g(\sigma)$ means that there exists an execution path from Start to σ in g.

Lemma 4.8. *Let* σ *be state such that* reachable$_g(\sigma)$. *For all* σ', *we have*

$$\sigma \rightarrow_{\text{STEP}_g} \sigma' \text{ if, and only if, } \sigma \rightarrow_{\text{STEP}_{g'}} \sigma'$$

Proof sketch. The only subtle case is the case where σ = Exec(r, ρ) and the control node of r in g is a ZeroCheck(r, x) that has been optimized into a Jmp(r) in g'.

If it was optimized, then it must be that `redundantZC`$[r]$ = `true`. Lemma 4.5 ensures that there exists r' strictly dominating r whose control node is a ZeroCheck(r', x).

Suppose $\sigma \rightarrow_{\text{STEP}_g} \sigma'$. We know that σ is a reachable state, so we can apply Corollary 4.2. Hence, $[\![x]\!]_\rho = v$ with $v \neq 0$, and the ZeroCheck passes (the applicable rule is ZCHK1), and g' can match the step anyway by executing its Jmp node (which has the same successor).

Now, if $\sigma \rightarrow_{\text{STEP}_{g'}} \sigma'$ through the Jmp node, we match this step in g with rule ZCHK1, which we know is applicable (by Corollary 4.2, because σ is reachable in g).

□

5 Conclusion and Future Work

In this paper, we have proposed a formal semantics for a Sea of Nodes representation. The salient aspects of the semantics are twofold. First, it doesn't bind data computation to a particular region node. Rather, data nodes (and their dependencies) are evaluated on demand, using a denotational semantics in an environment that keep tracks of the current values of phi-nodes. Second, the control-flow execution of a Sea of Nodes is defined operationally, using a small-step semantics between region nodes. During the course of our formalization work, we relied on a prototype interpreter reflecting the semantic rules so as to make sure that our formalization was indeed executable.

On top of this semantics, we proved a fundamental property of validity domain for value information of nodes in an area of the graph in terms of dominance. We used this property to prove the semantics preservation of a redundant zero-check elimination algorithm. We think that the same property, and similar reasoning would allow to prove the

semantics correctness of other optimizations, such as Global Value Numbering, which require to justify that some nodes have the same value at two different places.

Our Coq development also includes a formalization of a Simple Constant Propagation algorithm. The simulation argument is similar to the one of the ZeroCheck elimination, but does not involve fundamental properties from a Sea of Nodes point of view (e.g., dominance).

An important question not studied in this paper is the generation of the Sea of Nodes form. Click et al. [8] propose an incremental algorithm for building the SSA form while doing parse-time optimizations. As a first step towards its formal verification, an easier approach would be to rely on previous works on SSA generation [1, 22], and to generate the Sea of Nodes from a basic-block based SSA: the transformation would just need to remove superfluous control-flow dependences, keeping only relevant data and control dependences.

Another interesting topic is the transformation out of the Sea of Nodes form, towards an IR closer to assembly. A possible approach to tackle this difficult problem is to split it into three phases. The first phase is Global Code Motion [6]. In order to actually generate code, it is necessary to ascribe a region to every data node. Our Sea of Nodes representation semantics is not a sequential one: non-ϕ data nodes do not depend on a particular region node, and are evaluated with a denotational semantics. This raises an interesting question, because in order to prove that Global Code Motion is correct, we would need to prove equivalence between our semantics, and a basic-block based semantics. The second phase schedules instructions within basic-blocks. Blech et al. [2] give a basic-block based semantics to instruction evaluation that explicitly integrates dependencies and scheduling into their semantics. Their work allows for a transition between a semantics using only necessary data dependencies *within a basic-block* and a semantics using extra dependencies corresponding to a complete scheduling of the basic-block. Their work could possibly be used for this second part. In the third phase, it would remain to implement ϕ-nodes, i.e., destructing the SSA form. Here, we could possibly make use of some previous work done within the CompCertSSA project [10]. We are confident the work presented here is a solid base to attack these future works, and paves the way to an intregration of Sea of Nodes in formally verified compilers.

More generally, it would be worth investigating whether the results presented here could be applied to other similar intermediate representations [21], such as the Value State Dependent Graph [15].

Acknowledgments

This material is based upon work supported by the Agence Nationale de la Recherche under Grant No. ANR-14-CE28-0004.

References

[1] G. Barthe, D. Demange, and D. Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM TOPLAS* 36, 1 (2014), 1–35.

[2] J.O. Blech, S. Glesner, J. Leitner, and S. Mülling. 2005. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *Proc. of COCV'05 (ENTCS)*. Elsevier, 33–51.

[3] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Proc. of CC'13 (LNCS)*, Vol. 7791. Springer, 102–122.

[4] M. Braun, S. Buchwald, and A. Zwinkau. 2011. *Firm—A Graph-Based Intermediate Representation*. Technical Report 35. Karlsruhe Institute of Technology. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470

[5] S. Buchwald, D. Lohner, and S. Ullrich. 2016. Verified Construction of Static Single Assignment Form. In *Proc. of CC'16*. ACM, 67–76.

[6] C. Click. 1995. Global Code Motion / Global Value Numbering. In *Proc. of PLDI'95*. 246–257.

[7] C. Click and K. D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM TOPLAS* 17, 2 (1995), 181–196.

[8] C. Click and M. Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Proc. of IR'95*. 35–49.

[9] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS* 13, 4 (1991), 451–490.

[10] D. Demange and Y. Fernandez de Retana. 2016. Mechanizing Conventional SSA for a Verified Destruction with Coalescing. In *Proc. of CC'16 (CC 2016)*. 77–87.

[11] D. Demange, L. Stefanesco, and D. Pichardie. 2015. Verifying Fast and Sparse SSA-based Optimizations in Coq. In *Proc. of CC'15 (LNCS)*, Vol. 9031. 233–252.

[12] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proc. of VMIL'13*. ACM, 1–10.

[13] B. Dupont de Dinechin. 2014. Using the SSA-Form in a Code Generator. In *Proc. of CC'14 (LNCS)*, Vol. 8409. Springer, 1–17.

[14] HotSpot 1999. Homepage of the project. http://openjdk.java.net/groups/hotspot.

[15] N. Johnson and A. Mycroft. 2003. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In *Proc. of CC'03 (CC 2003)*. Springer-Verlag, 1–16.

[16] M. Kawahito, H. Komatsu, and T. Nakatani. 2000. Effective Null Pointer Check Elimination Utilizing Hardware Trap. In *Proc. of ASPLOS'00*. 139–149.

[17] X. Leroy. 2009. A Formally Verified Compiler Back-end. *JAR* 43, 4 (2009), 363–446.

[18] Nilcheck elimination in Go 2017. Source code. https://github.com/golang/go/blob/release-branch.go1.9/src/cmd/compile/internal/ssa/nilcheck.go.

[19] B. K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proc. of POPL'88*. 12–27.

[20] The Go programming language 2012. Homepage of the project. https://golang.org/.

[21] The SSA Book 2015. Latest release. http://ssabook.gforge.inria.fr/latest/book.pdf.

[22] J. Zhao. 2013. *Formalizing an SSA-based compiler for verified advanced program transformations*. Ph.D. Dissertation. University of Pennsylvania. Advisor(s) Zdancewic, Steve.

[23] J. Zhao, S. Nagarakatte, M. Martin, and S. Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *Proc. of PLDI'13*. ACM, 175–186.

[24] J. Zhao, S. Zdancewic, S. Nagarakatte, and M. Martin. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformation. In *Proc. of POPL'12*. ACM, 427–440.