Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

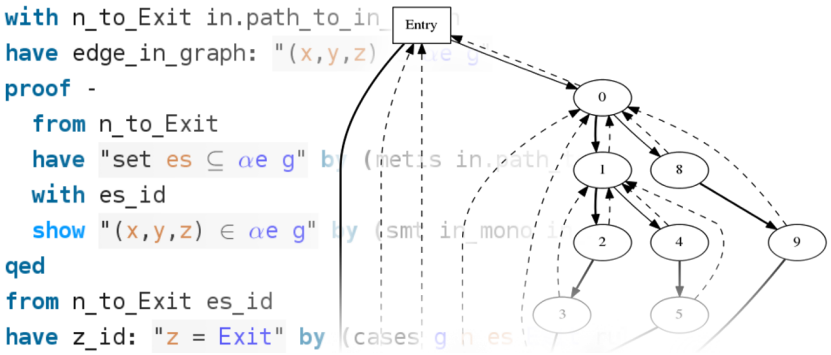
Lehrstuhl Prof. Dr.-Ing. Snelting

Verified Computation of Control Dependencies in a Control Flow Graph

Bachelorarbeit von

Maximilian Wagner

an der Fakultät für Informatik



Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	Dipl.-Inform. Denis Lohner

Bearbeitungszeit: 5. September 2013 – 5. November 2013

Zusammenfassung

Diese Arbeit liefert eine sprachunabhängige Implementierung eines Algorithmus zur Berechnung von Standardkontrollabhängigkeiten in Kontrollflussgraphen nach Ferrante et al. [8] mit Hilfe des Theorembeweisers Isabelle/HOL. Der vorgestellte Algorithmus verwendet eine funktionale Variante des Lengauer-Tarjan-Algorithmus [20] für die hierfür benötigte Berechnung von Postdominatoren, setzt aber nicht auf den von Lengauer und Tarjan verwendeten *EVAL-LINK*-Mechanismus. Die gesamte Arbeit ist zur Erhaltung der Modul- arität innerhalb eines Beweiskontextes gehalten, der auf möglichst wenigen Annahmen über die gegebenen Graphen basiert. Deren Erfüllbarkeit ist für zwei spezifische Instanzen (eine basierend auf einer formalen Definition von Graphen, die andere basierend auf den von Kohlmeyer [17] konstruierten konkreten Kontrollflussgraphen einer einfachen While-Sprache) bewiesen und die gesamte Theorie lässt sich zu Code in verschiedenen funktionalen Sprachen (u.a. Haskell, SML, Scala) kompilieren. Die Termination des Algorithmus und bestimmte Korrektheitsaussagen sind mit Hilfe von Isabelle bewiesen.

Abstract

In this thesis, we present a language independent and machine-checked implementation of an algorithm to compute control dependencies (as defined by Ferrante et al. [8]) in a control flow graph (CFG). The work presented in this document uses the proof assistant Isabelle/HOL. The presented algorithm uses a variant of the Lengauer-Tarjan algorithm for finding dominators, and operates on abstract CFGs. A proof for the satisfiability of the assumptions about the given CFGs is present, thus enabling Isabelle to generate code in a variety of target languages (e.g. Haskell, SML, Scala). We also supply proofs in Isabelle for termination and certain correctness properties.

Contents

1 Introduction

Knowledge of control dependencies enables certain powerful techniques for static analysis, like optimizations performed on the program dependence graph [8, 19], as well as a very general tool for program analysis called Slicing [14].

Slicing is a technique defined by Weiser used in optimization, program analysis, debugging and information flow control [12]. It is a technique for deciding the following problem:

> Given some point p in a program P and a subset V of the variables in P, which statements affect the values of variables in V at p?

We distinguish between at least two kinds of slicing: *static* and *dynamic* slicing, referring to slicing using only statically available information, and slicing augmented with information from specific execution traces. In order to use slicing, we must first decide on a formal criterion for whether a statement influences another statement called the *slicing criterion*. A basic slicing criterion would be deciding which statements influence whether the flow of execution ever reaches p.

Unfortunately, many software analyses published are lacking proofs of correctness or only supply pen & paper proofs, which are error-prone. This is especially tragic in the context of software security analysis, where small errors and oversights can compromise the soundness of whole cryptosystems (for an example, see [4] and [16]). It is for this reason that formal proofs using automated proof assistants or theorem provers are of special interest: In the case of Language Based Security, machine checked proofs can achieve a new level of trust unattainable by conventional means when it comes to properties of language semantics.

With this thesis, we contribute a verified implementation of an algorithm to compute standard control dependencies in a CFG with the goal of providing part of a verified slicing framework to be used in the *"Quis Custodiet"*-project[1] to conduct security analysis.

Section 2 provides an explanation for some of the concepts used in this thesis, Section 3 sets up the general proof context that is used in this work. In Section 4 we explain the different phases of our algorithm and present proofs for their termination and correctness. Finally, in Section 5 we present two possible instantiations of the proof contexts used in this work, thereby proving the satisfiability of their assumptions.

[1] see `http://pp.ipd.kit.edu/projects/quis-custodiet/`

2 Background

This section provides some background knowledge about control flow graphs, domination, control dependence and the Lengauer-Tarjan algorithm. We also provide a short explanation of what Isabelle is and present the framework this thesis uses as its basis.

2.1 Control Flow Graphs

A control flow graph (CFG) is an abstract representation of a program which highlights the possible flow of execution. CFGs are created by grouping statements into *basic blocks* and linking these blocks with edges along which execution can flow. The defining characteristic of basic blocks is that only the last statement of such a block is allowed to have more than one potential jump target, so statements within a block are always executed sequentially. Many formalizations also define a dedicated *Exit*-node (which is reachable by every node) and *Entry*-node (from which every node is reachable), though this thesis only requires the existence of the former. CFGs enable certain compiler optimizations, e.g. elimination of dead code, statement reordering, etc. through construction of the program dependence graph [8] and static analyses like information flow control through slicing [23].

2.2 Depth-First Search

The depth-first search algorithm (DFS) for traversing a graph is a fairly well-known concept in computer science, so we will not give a complete explanation of the algorithm here. We use DFS within this thesis to extract a spanning tree and a node numbering from a graph. We add edges to this spanning tree in the order they are traversed during an execution of DFS, ignoring edges to nodes that have already been discovered. The node numbering is produced in a similar manner by numbering newly discovered nodes in increasing order. This numbering induces an order on nodes, which we will use extensively in this thesis.

2.3 Postdomination

In a rooted graph (i.e. a graph with a designated node called the root, from which every node can be reached), domination is defined as follows:

> A node n' dominates another node n iff every path from the root to n passes through n'.

Similarly, postdomination is defined on graphs with a designated *Exit* node which is reachable from every node:

> A node n' postdominates another node n iff every path from n to the *Exit* node passes through n'.

Postdomination is thus equivalent to domination in the graph where all edges are reversed. (Post)domination is reflexive, transitive and antisymmetric, and as such can be used as the basis for orderings.

Semidominators, denoted $sdom(n)$, provide a convenient intermediate step for calculating dominators. Given a CFG and a node numbering resulting from a DFS run on the graph starting at the *Entry*-node, the semidominator of a node n is defined as:

$$\begin{aligned} sdom(n) = \quad & \min\{m \mid \text{there is a path } m = m_0, m_1, \ldots, m_k = n \\ & \text{such that } m_i > n \text{ for } 1 \leq i \leq k-1\} \end{aligned} \tag{1}$$

Analogously, semi-postdominators (noted $spdom(n)$) can be defined as:

$$\begin{aligned} spdom(n) = \quad & \min\{m \mid \text{there is a path } n = m_0, m_1, \ldots, m_k = m \\ & \text{such that } m_i > n \text{ for } 1 \leq i \leq k-1\} \end{aligned} \tag{2}$$

In the following we will call paths that fit the description in (2) *candidate paths*.

For a node n, n's **immediate dominator** (noted $idom(n)$) is the node $m \neq n$ such that m dominates n and for all other nodes k which dominate n, k dominates m. Informally, it is the "last" dominator of n on the path from *Entry* to n. Immediate postdominators (noted $ipdom(n)$) are defined analogously, being the "first" postdominators on the path from n to *Exit*. Immediate (post-)dominators form a chain from *Entry* to n (from n to *Exit*) in which every (post-)dominator of n is present. Computation of a node's immediate postdominator is possible in linear time [5] and is a major part of the central algorithm of this thesis.

Figure 1 shows the behavior of *sdom* and *idom* under different node numberings and spanning trees.

2.4 Control Dependence

A statement s is control dependent on another statement s' iff the result of s' decides whether s executes. For example, in an *if-then-else* expression, the

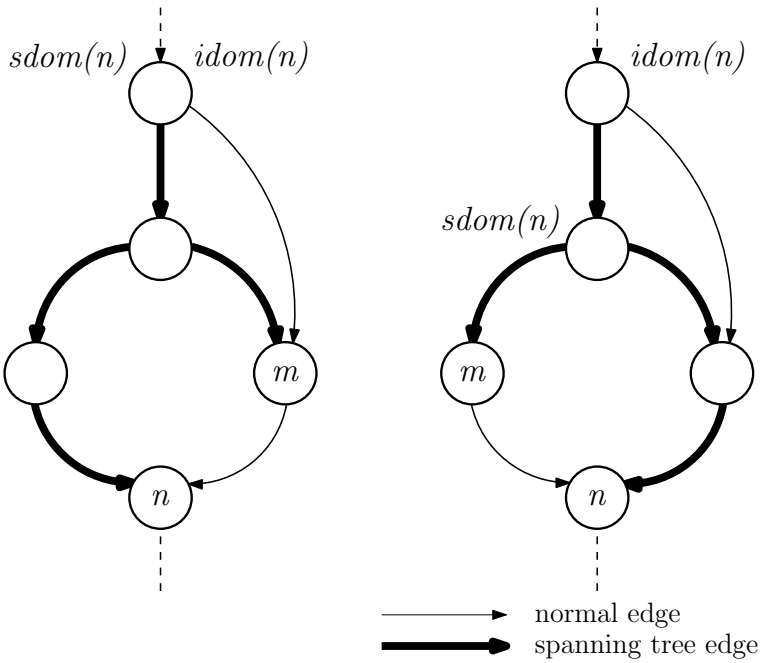


Figure 1: Choice of spanning tree affects $sdom(n)$, but not $idom(n)$. Note that in both these spanning trees, if $m < n$ were true, the edge (m, n) would be a spanning tree edge, so m must be greater than n.

then and *else* blocks are control dependent on the *if*-predicate.

This characterization strongly depends on the semantics of the language that is being used. Ferrante et al. [8] define a commonly used language independent version of this called *standard control dependence* in graph-theoretical terms. This thesis uses a different but equivalent [22] definition originally given by Wolfe:

> A node n' is control dependent on a node n iff n has at least two successors, one postdominated by n', while the other one is not.

2.5 The Lengauer-Tarjan Algorithm

Lengauer and Tarjan provide a fast (quasilinear) algorithm [20] for computing dominators in a flow graph. The general structure of Lengauer and Tarjan's algorithm is as follows:

1. Crawl the graph using depth-first search (starting at the *Entry* node), thus producing a node numbering (called the node's DFS-ID) and a spanning tree.
2. Compute the semidominators of all vertices in decreasing order by DFS-ID.
3. Implicitly define the immediate dominator of each vertex in terms of semidominators of other nodes.
4. Explicitly define the immediate dominator of each vertex whose dominator was not already computed by step 3, carrying out the computation vertex by vertex in increasing order by DFS-ID.

After having completed step 1, the algorithm carries out steps 2 and 3 simultaneously, processing each node using two procedures called *LINK* and *EVAL* which extract information from and modify a forest along the course of computation. More precisely, the forest consists of vertex set V and edge set $\{(parent(w), w) \mid \text{vertex } w \text{ has been processed}\}$. *LINK* and *EVAL* perform the following operations:

LINK(v,w):	Add edge (v, w) to forest.
EVAL(v):	If v is the root of its tree in the forest, return v. Otherwise, return the node on the path from root to v which has the smallest *sdom*.

LINK and *EVAL* thus correspond to operations on disjoint-set data structures. The algorithm state is initialized so that every node initially reports its *sdom* to be itself, and the forest is initialized with a graph containing only nodes and no edges. While processing a node n, the algorithm sets $sdom(n)$ to be the following:

$$\min\{sdom(EVAL(m)) \mid m \text{ is a predecessor of } n\}$$

Then the algorithm executes $LINK(parent(n), n)$. Step 3 is carried out by defining a preliminary *idom* for each node v that is dominated by $parent(n)$ by choosing between $parent(n)$ and *EVAL(v)*, selecting the node with smaller *sdom*. (Note that the computation is done in decreasing order because we need to process cross edges after the "surrounding" tree edges have been processed.[2])
Afterwards, another pass is made over the graph's nodes, this time by increasing DFS-ID, resolving the last incorrect values for *idom* by setting the *idom* of those nodes k with differing *idom* and *sdom* to $idom(idom(k))$. Since $idom(k) < k$, all *idom*s used in this step are already correct by the time they're referenced. The algorithm exploits certain identities that define semidominators in terms of DFS-ID (see Section 2.3), and immediate dominators in terms of semidominators.

While the original version of this algorithm runs in quasilinear time, linear versions have been proposed [11, 1, 6, 9, 5], but most of these have been found to either be overly complicated or nonlinear after all. For a full explanation and analysis of the original algorithm and a (pen and paper) proof of its correctness, see Lengauer and Tarjan [20].

2.6 Isabelle

This thesis employs Isabelle, a powerful and generic theorem prover that can be instantiated with a series of object logics. This thesis uses its default logic, Isabelle/HOL, which provides a large library of lemmas in higher-order logic. Isabelle provides Isar, an intelligible proof structuring language, and supports generation of code from theories, allowing a user to harness the power of Isabelle along with the efficiency of the advanced compilers that exist for the target languages. Code generation into SML, OCaml, Haskell and Scala is supported.

[2] All cross edges are edges (e, f) with $e > f$.

The examples, definitions and code snippets shown in this document have been simplified and stripped of redundant information to improve legibility.

Much of Isabelle/HOL syntax is standard mathematical notation. Function application is written by leaving a space between function symbols. Function types are denoted by the $\Rightarrow$ symbol. Meta-level implication is denoted using $\Longrightarrow$. The *definition* keyword defines a nonrecursive function, the keywords *fun* and *function* define potentially recursive functions using mechanisms like pattern matching, *case _ of* expressions and *let _ in* definitions familiar from other languages. Functions in Isabelle/HOL are total functions which are well-defined for all possible parameters, two properties which first have to be proven by the user. Using *fun*, Isabelle attempts to prove these automatically [18], while *function* requires explicit proofs. Type annotations of the form $t::\tau$ mean that the term t has type τ. Type variables start with a single quote, and *nat*, *bool*, *unit* are types representing natural numbers, boolean values and the unit type respectively. Tuple types are written as cartesian products and internally represented as nested 2-tuples. The empty set is simply written as *{}*, and the image of a set A under a function f is written as f ‘ A. Lists can be either the empty list *[]* or a list consisting of one element x and the rest xs of the list, with such a construct being written as $x\#xs$. In that case, the head of the list is fetched by the function *hd*, and the rest by the function *tl*. Concatenation of two lists xs and ys is noted as $xs@ys$. The *concat* function flattens a list of lists by concatenation, *map* applies a function to all elements of a list, and *butlast* returns the given list stripped of its last element. The *option* types represent either *None* or *Some value*, which can be unwrapped with the function *the*. New types (which mustn't be empty) can be defined by using the *typedef* or *datatype* keywords, with *datatype* providing Haskell-like syntax for defining algebraic data types [3], and *typedef* allowing users to define new types by limiting previously-defined types.

Isabelle locales [2] are a mechanism Isabelle provides for modularizing theories. A *locale* is a named theory context for which a user can postulate certain assumptions and operations (referred to as locale *parameters*, which are *fixed*). All lemmas formulated within the context of a locale have access to the locale's parameters and assume the locale's assumptions to hold. Thus it is of critical importance for the locale assumptions to be satisfiable, as anything can be proven by relying on insatisfiable assumptions. Locales can be extended using the + operator. A locale extension has access to the original locale's parameters, assumptions, and all lemmas proved in the original.

Proving the satisfiability of a locale's assumptions is part of the responsibility of the user. To do this we have to *interpret* the locale, a term which refers to the act of instantiating all locale parameters and proving the locale assumptions to hold for the given parameters. Isabelle's built-in code generation feature can only generate code for interpreted locales.

2.7 Graph Framework

This thesis uses and extends the graph framework built in Isabelle by Kohlmeyer [17] to represent CFGs. The locale signatures and an explanation of the relevant operations follow:

locale *graph* =
fixes
  $\alpha e :: 'g \Rightarrow ('node \times 'edgeD \times 'node)\ set$ **and**
  $invar :: 'g \Rightarrow bool$

The locale *graph* is the most basic version of a graph, providing only a function for fetching a graph's edge set (αe) and a predicate for sanity-checking the given graph (*invar*). The signature of this locale also shows the type parameters involved: *'g* is the graph type, *'node* is the type of nodes in the graph, and *'edgeD* is the type of the edge descriptors (these are not used in this thesis). The *invar* predicate is needed in extensions of this locale for limiting lemmas to valid graphs.

locale *graph-nodes* = *graph* +
fixes
  $\alpha n :: 'g \Rightarrow 'node\ list$
assumes αn*-correct*:
$invar\ g \Longrightarrow set\ (\alpha n\ g) = fst\ `\ \alpha e\ g \cup snd\ `\ snd\ `\ \alpha e\ g$

Locale *graph-nodes* extends *graph* and provides αn, a function which returns a list of all nodes of the given graph. It follows from the assumption αn*-correct* that no isolated node can exist in a graph, as nodes are only characterized as end points of an edge.

locale *graph-empty* = *graph* +
fixes *empty* :: *'g*
assumes
  empty-invar: *invar empty* **and**
  empty-correct: $\alpha e\ empty = \{\}$

locale *graph-addEdge* = *graph* +
fixes *addEdge* :: $'g \Rightarrow 'node \Rightarrow 'edgeD \Rightarrow 'node \Rightarrow 'g$
assumes
addEdge-invar: $invar\ g \Longrightarrow invar\ (addEdge\ g\ f\ d\ t)$ **and**
addEdge-correct: $invar\ g \Longrightarrow e \in \alpha e\ (addEdge\ g\ f\ d\ t)$
$\longleftrightarrow e = (f,\ d,\ t) \vee e \in \alpha e\ g$

The locale *graph-empty* augments the graph-locale with an empty graph, and *graph-addEdge* adds a function for adding edges to a given graph. These two locales allow graphs to be constructed edge by edge.

locale *graph-outEdges* = *graph* +
fixes *outEdges* :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list$
assumes *outEdges-correct*:
$invar\ g \Longrightarrow set\ (outEdges\ g\ n) = \{(f,\ \text{-},\ \text{-}).\ f = n\} \cap \alpha e\ g$

locale *graph-inEdges* = *graph* +
fixes *inEdges* :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list$
assumes *inEdges-correct*:
$invar\ g \Longrightarrow set\ (inEdges\ g\ n) = \{(\text{-},\ \text{-},\ t).\ t = n\} \cap \alpha e\ g$

The locales *graph-outEdges* and *graph-inEdges* provide functions returning a list of a node's outgoing and incoming edges respectively, along with assumptions concerning the correctness of the functions.

3 Technical Foundation

Before getting into the details of the algorithm used in this paper, we first have to create the abstract setting in which the algorithm is defined.

locale *graph-path* = *graph-nodes*
 for αe :: $'g \Rightarrow ('node \times 'edgeD \times 'node)\ set$
 and *invar* :: $'g \Rightarrow bool$
 and αn :: $'g \Rightarrow 'node\ list$

inductive *path-to*
 :: $'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list \Rightarrow 'node \Rightarrow bool$ (- ⊢ - −-→ -)
 for g :: $'g$ **and** n :: $'node$ **where**
 $[\![\ n \in set\ (\alpha n\ g)\ ]\!] \Longrightarrow (g \vdash n -[]\rightarrow n)$ |
 $[\![\ g \vdash n -es\rightarrow n';\ (n',e,n'') \in \alpha e\ g\ ]\!]$
 $\Longrightarrow (g \vdash n -es@[(n',e,n'')]\rightarrow n'')$

The locale *graph-path* augments *graph-nodes* with the recursively defined predicate *path-to*, which expresses the existence of a path within the graph. Paths are modeled as lists of edges, and defined inductively from "front to back", though an introduction rule for prepending edges to a path exists too. Null (i.e. empty) paths are explicitly allowed, so for every node n in the graph g, "$g \vdash n$ -[]→ n" is true. Note that in general inductive predicates in Isabelle are not "executable" in the sense that we can't generate code for obtaining parameter combinations which satisfy the predicate, so this predicate can only be used for proving properties about the objects defined in our locale, not for performing any computation[3].

locale *graph-exit* = *graph-path* αe *invar* αn
 for αe :: $'g \Rightarrow ('node \times 'edgeD \times 'node)\ set$
 and *invar* :: $'g \Rightarrow bool$
 and αn :: $'g \Rightarrow 'node\ list$ +
fixes *Exit* :: $'node$
assumes *Exit-in-graph*: $Exit \in set\ (\alpha n\ g)$
 and *path-to-Exit*: $[\![\ n \in set\ (\alpha n\ g)\ ]\!] \Longrightarrow (\exists\ es.\ g \vdash n -es\rightarrow Exit)$
 and *Exit-is-Exit*:
 $[\![\ g \vdash Exit -es\rightarrow n\ ]\!] \Longrightarrow es = [] \wedge n = Exit$

[3]While there exists a mechanism for attempting to automatically create code for such inductive predicates, this does not always succeed and will in general generate rather inefficient code

The locale *graph-exit* extends *graph-path* by fixing the *Exit*-node and assuming two basic properties about this node:

path-to-Exit: *Exit* is reachable from every node in g.
Exit-is-Exit: *Exit* has no outgoing edges.

The final locale signature presented in this section is *graph-control-Dependencies*. It handles three different graph types:

'g
The input CFG, captured in the namespace "in". It needs to support the operations *invar*, αe, αn, *outEdges* and *inEdges*.

'span
The graph used for building and working with the spanning tree obtained during the computation, captured in the namespace "span". Supported operations are *invar*, αe, αn, *outEdges* and *addEdge*. The existence of an empty graph of this type is also needed for building a graph from scratch. As shorthand, operations on the spanning tree share a *'* as suffix.

'outg
The output graph type, captured in the namespace "out". Supported operations are *invar*, αe, *addEdge*. An empty graph is needed here, too. As shorthand, operations on the output graph share *''* as suffix.

Together with the basic parts of the graph framework presented in Section 2.7, this provides us with all the basic operations we need for computing postdominators and control dependencies.

locale *graph-controlDependencies* =
  in:*graph-nodes*
  + *in*:*graph-inEdges*
  + *in*:*graph-outEdges*
  + *in*:*graph-exit*
  + *span*:*graph-nodes*
  + *span*:*graph-outEdges*
  + *span*:*graph-addEdge*
  + *span*:*graph-empty*
  + *out*:*graph-addEdge*
  + *out*:*graph-empty*
  for αe :: $'g \Rightarrow ('node::linorder \times 'edgeD \times 'node)\ set$
  and $invar :: 'g \Rightarrow bool$
  and $\alpha n :: 'g \Rightarrow 'node\ list$
  and $inEdges :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list$
  and $outEdges :: 'g \Rightarrow 'node \Rightarrow ('node \times 'edgeD \times 'node)\ list$
  and $\alpha e' :: 'span \Rightarrow ('node \times unit \times 'node)\ set$
  and $invar' :: 'span \Rightarrow bool$
  and $\alpha n' :: 'span \Rightarrow 'node\ list$
  and $outEdges' :: 'span \Rightarrow 'node \Rightarrow ('node \times unit \times 'node)\ list$
  and $addEdge' :: 'span \Rightarrow 'node \Rightarrow unit \Rightarrow 'node \Rightarrow 'span$
  and $empty' :: 'span$
  and $\alpha e'' :: 'outg \Rightarrow ('node::linorder \times unit \times 'node)\ set$
  and $invar'' :: 'outg \Rightarrow bool$
  and $addEdge'' :: 'outg \Rightarrow 'node \Rightarrow unit \Rightarrow 'node \Rightarrow 'outg$
  and $empty'' :: 'outg$ +
  assumes *finiteness*: $finite\ (\alpha e\ g)$
  and *g-always-invar*: $invar\ g$

Apart from the assumptions inherited from other graph locales, this locale also assumes the given graph to be finite, as a depth-first search on an infinite graph is problematic at best. For technical reasons and for convenience when defining functions, this locale also assumes that all graphs that serve as input are valid graphs.

4 Construction

This section explains the different steps taken by our algorithm to get from program code in any programming language to the control dependence graph. A high-level view distinguishes four different phases of the algorithm:

Prep Generate a CFG from the given code.

1. Crawl the CFG using a depth-first search, yielding a node numbering and a spanning tree.
2. Calculate semi-postdominators and immediate postdominators.
3. Compute control dependencies and generate the output graph.

4.1 CFG Generation

The first step is to extract the CFG from the program code. This strongly depends on the semantics of the programming language in question, and as this thesis concentrates on computing the control dependence graph from a CFG, this is only listed here for the sake of completeness.

The examples and figures presented in the context of this thesis represent the following simple program[4]:

```
if (x <= 0) {
    if (x < 0) {
        y := y - 1;
        x := 100;
    } else {
        x := y / x;
    }
    z := true;
} else {
    while (x > 0) {
        x := x - 1;
        y := y + 1;
    }
    z := false;
}
```

[4]The actual language for which we supply a locale interpretation only supports the operators “=”, “&”, “<”, “+” and “-”. Though this set is formally complete, it lacks several operators shown in the example. We still use these here for didactic purposes.

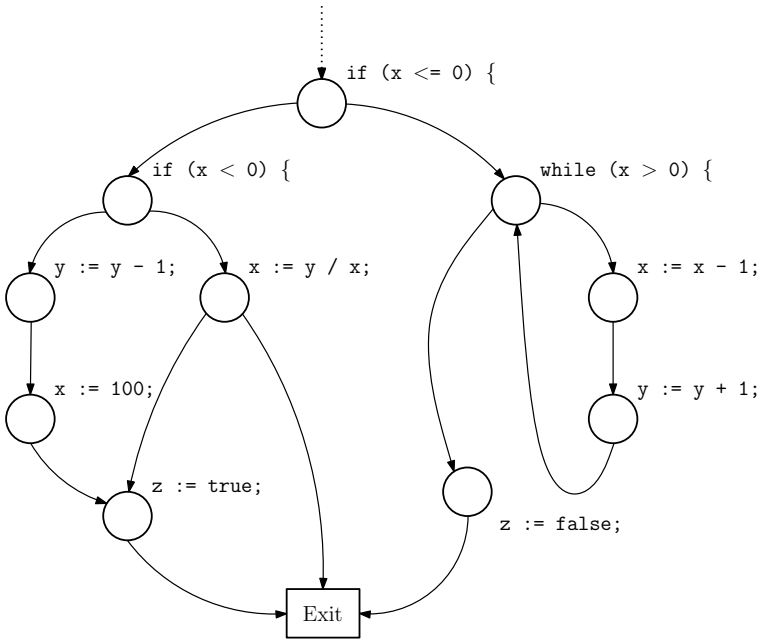


Figure 2: A control flow graph for the example program. Note the edge from "`x := y / x;`" to *Exit*. This is due to the possibility of a division-by-zero-exception.

The execution of line 2 depends on line 1, lines 3 and 4 depend directly on line 2, and so on. It is important to note that while lines 3 and 4 also depend on line 1 indirectly, this thesis does not concern itself with indirect dependencies, as indirect dependence can easily be computed by forming the transitive closure of the direct dependencies.

A possible control flow graph for this program is shown in Figure 2.

4.2 Depth-First Search

In order to compute postdominators, we first need to extract a spanning tree and the corresponding node numbering from the graph. As the original version of Lengauer-Tarjan considers dominators and we're interested in postdominators, we could use the algorithm on an edge-reversed graph[5]. Instead, we simply treat edges as if they were reversed and start the DFS on

[5]By reversing all edges, a node's dominators become its postdominators and vice versa.

the *Exit* node, thus saving us the trouble of reversing the graph twice.

The depth-first search is split into three parts: A single-step function, *dfs-step*, a recursive function *dfs'* and a wrapper *dfs* around *dfs'* with the initial parameters. This splitting of one function into three parts is done in order to make the function more tractable for proofs. But first, some preliminaries:

type-synonym *'node numbering* = (*'node*, *nat*) *mapping*

type-synonym (*'node*, *'edgeD*, *'span*) *dfs-state* =
  ((*'node* × *'edgeD* × *'node*) *list* × *'node numbering* × *'span*)

The function symbols *getTo* and *getFrom* are abbreviations for the functions which return the first and third entry in a tuple, respectively. Their use is to make function definitions handling edges and their end points more readable. We define a numbering to be a mapping from type *'node* to the natural numbers. In Isabelle/HOL, *mapping*s provide an abstract view on partial functions which can be updated[6]. They support code generation and are used as key-value stores.

Furthermore, to simplify type signatures we define *dfs-state* to be the type synonym to a tuple consisting of an edge list, a node numbering and a graph[7]. During the computation of DFS, this tuple will hold the stack of edges we have yet to look at, our mapping of visited nodes (the node numbering), and the part of the spanning tree constructed so far.

[6]using the function *Mapping.update :: 'a ⇒ 'b ⇒ ('a, 'b) mapping*
[7]Technically, all these are still type parameters at this point and could be anything.

```
fun dfs-step :: 'g ⇒ ('node, 'edgeD, 'span) dfs-state ⇒ (('node, 'edgeD, 'span)
dfs-state) option
  where
  dfs-step g ([],visited,span) = None |
  dfs-step g (x#xs,visited,span) = (
    if getFrom x ∈ set (αn g) then
      (case (getFrom x ∈ Mapping.keys visited) of
        True ⇒ Some (xs,visited,span) |
        False ⇒ Some (
          (inEdges g (getFrom x)) @ xs,
          Mapping.update (getFrom x) (Mapping.size visited) visited,
          addEdge' span (getFrom x) () (getTo x)
          )
      )
    else None)
```

Function *dfs-step* is the aforementioned single-step function. It takes a graph to search and an algorithm state and computes the state of the algorithm after the next step of DFS.
If the stack is empty, we've already crawled the whole graph, and return *None*, as there is nothing left to be done. If we still have edges to process, we pop the topmost edge off the stack and check if its source node[8] is in the set of all nodes we've found (and numbered) so far. If so, both the spanning tree and the node numbering remain untouched and are returned together with the rest of the stack as the next *dfs-state*.
If however the edge we're processing originates from a node we haven't encountered yet, we push this node's incoming edges on the stack, update the numbering to incorporate the new node[9] and add this edge to the spanning tree.

As only "new" nodes are added to the graph and to the numbering, our numbering will not get corrupted by overwriting values and we're not introducing a new loop into the spanning tree.

[8] In a conventional depth-first search, we'd examine the edge's target node, but as we're interested in postdominators, we treat edges as if they were reversed.

[9] We use "*Mapping.size visited*" as a convenient way to identify the nodes with the order in which they were found without introducing another parameter.

function (*sequential*) *dfs'* :: *'g* ⇒ (*'node*, *'edgeD*, *'span*) *dfs-state*
⇒ (*'node numbering* × *'span*)
where (*dfs' g* (*st*,*vs*,*span*)) = (*case dfs-step g* (*st*,*vs*,*span*) *of*
None ⇒ (*vs*,*span*)
| *Some result* ⇒ (*dfs' g result*))

The function *dfs'* does little else but recursively call *dfs-step* until this returns no new state. This happens as soon as the stack is exhausted. We can prove that this eventually happens by examining the number of nodes left undiscovered and the size of the stack over the course of the algorithm's run time. The stack is only "refilled" with new edges if we find a node that has never been discovered. As there is only a limited number of nodes in the graph, this only happens a finite number of time. Every call to *dfs-step* that doesn't increase the stack's size decreases it, so the stack will eventually be exhausted and *dfs'* will come to a halt.

definition *dfs* :: *'g* ⇒ (*'node numbering* × *'span*)
where *dfs g* = *dfs' g* ((*inEdges g Exit*), *Mapping.update Exit 0 Mapping.empty*, *empty'*)

The final piece of the DFS implementation is *dfs*, which calls *dfs'* with the appropriate parameters and returns a tuple containing the final node numbering and spanning tree. The stack used for this contains all incoming edges into *Exit*, the initial numbering only contains *Exit* with DFS-ID zero, and the empty graph as initial spanning tree[10]. Figure 3 shows the result of applying *dfs* to our running example. Note that in the initial node numbering, *Exit* is treated as having DFS-ID zero.

definition *dfs-state-consistent* :: *'g* ⇒ (*'node*, *'edgeD*, *'span*) *dfs-state* ⇒ *bool*
where *dfs-state-consistent g* (*st*,*vs*,*span*) = *[...]*

As a first step in proving the correctness of this DFS implementation, we define a consistency predicate on a *dfs-state* named *dfs-state-consistent* and prove that this is preserved across the execution of *dfs-step*. The properties ensured by *dfs-step-consistent* include:

- *span*'s nodes are a subset of *g*'s.
- All nodes in *vs* have different DFS-ID.

[10]Due to the empty initial spanning tree but nonempty initial numbering, this initial state is not considered "consistent" by the version of the consistency predicate defined in this document. This discrepancy between spanning tree and numbering is unavoidable, since the graph framework this work uses doesn't allow the existence of unconnected nodes in a graph.

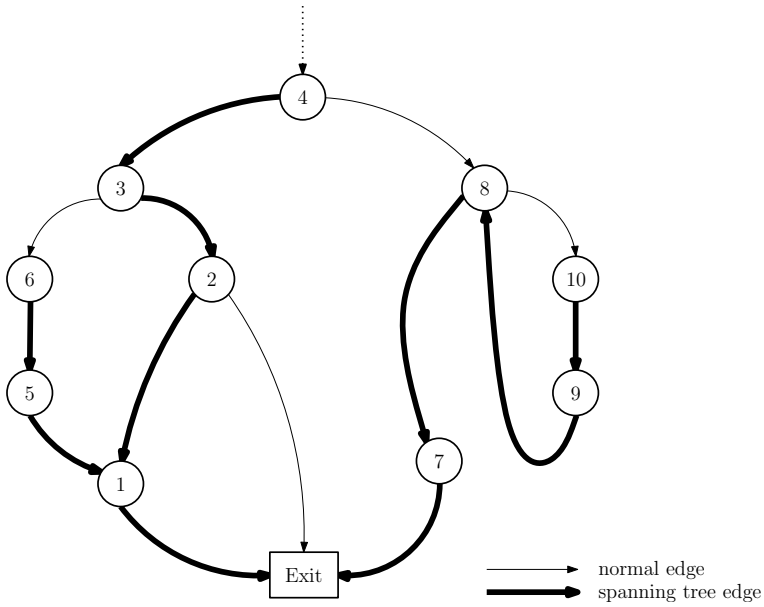


Figure 3: The node numbering and spanning tree superimposed on the CFG after one possible execution of the first phase of the algorithm.

- All edges in the stack lead to nodes that are already in *span*.
- All edges in g for which only the target node is in *vs* are in the stack.
- For every node n not in *span*, there exists a path from n through an edge that is in the stack.
- All nodes have a DFS-ID smaller than the size of the numbering.
- The target node of all edges in the stack is in *span*.
- The numbered nodes are exactly the nodes in *span*.
- *span* is a reversed minheap, i.e. a node n's tree parent is always smaller than n.

As a next step, we prove that after one execution of *dfs-step*, the state of the algorithm as started by *dfs* fulfills *dfs-state-consistent*. By induction, we obtain that the result of *dfs* also fulfills this predicate.

4.3 Computation of Postdominators

In Section 2.5 we've already presented the classical Lengauer-Tarjan algorithm for computing dominators in a flow graph. However, as Lengauer-Tarjan (and its variants) derive much of their efficiency from using highly specialized mutable data structures which are not readily available in a functional setting, we've opted for implementing the functions computing a node's semi-postdominator and immediate postdominator more directly according to their formal definitions. This has the consequence that the implementation discussed here doesn't fulfill the same asymptotic speed bound as classical Lengauer-Tarjan or its linear-time variants, but it also greatly increases the formal tractability of the code.

Before elaborating on the intricacies of *spdom* and *ipdom* computation, we will introduce some helper functions:

The functions *spanning-tree*, *spanning-tree-parent* and *dfs-id* provide high-level access to the results of *dfs*, and behave as one would expect. Their definitions are as follows.

definition *tree-parent* :: *'span* ⇒ *'node* ⇒ *'node option*
  where *tree-parent s n* = (*if outEdges' s n* = [] *then*
    None
  else
    Some (*getTo* (*hd* (*outEdges' s n*))))

definition *spanning-tree* :: *'g* ⇒ *'span*
  where *spanning-tree g* = *snd* (*dfs g*)

definition *spanning-tree-parent* :: *'g* ⇒ *'node* ⇒ *'node option*
  where *spanning-tree-parent g n* = *tree-parent* (*spanning-tree g*) *n*

definition *dfs-id* :: *'g* ⇒ *'node* ⇒ *nat*
  where *dfs-id g n* = *the* (*Mapping.lookup* (*fst* (*dfs g*)) *n*)

Next we define the function *tree-path-nodes*, which takes a graph g and two nodes n and m, and returns the list of nodes in the spanning tree path from n to m (including n and m) in g, if such a path exists, and *None* otherwise. It operates by first checking if such a path exists between n's spanning tree parent and m and if so, returns this path, prepended with n.

```
function tree-path-nodes :: 'g ⇒ 'node ⇒ 'node ⇒ ('node list) option
  where tree-path-nodes g n m = (
    if (n = m) then
      Some [n]
    else
      (if spanning-tree-parent g n = None then
        None
      else
        Option.map (op # n)
        (tree-path-nodes g (the (spanning-tree-parent g n)) to)
      )
    )
```

The following definition provides us with *min-target-node*, which takes an edge as a seed and a list of edges and returns the smallest target node of all these edges.

```
fun min-target-node :: 'g ⇒ ('node × 'edgeD × 'node) ⇒ ('node × 'edgeD × 'node)
list ⇒ 'node
  where
  min-target-node g e [] = getTo e |
  min-target-node g accu (x#xs) = (
     if dfs-id g (getTo x) < dfs-id g (getTo accu) then
        min-target-node g x xs
     else
        min-target-node g accu xs
  )
```

4.3.1 Semi-Postdominators

The Lengauer-Tarjan algorithm computes semi-postdominators with its *EVAL-LINK*-mechanism, which is based on a disjoint-set forest. The algorithm presented in this thesis however does not use this mechanism and employs a more naive and direct implementation based on the formal definition of *spdom*. In Section 2.3 we've established the following:

$$spdom(n) = \min\{m \mid \text{there is a path } n = m_0, m_1, \ldots, m_k = m \text{ such that } m_i > n \text{ for } 1 \leq i \leq k-1\} \tag{2}$$

First, we define the function *non-smaller-reachable-nodes*, which takes a graph g, a node n, a list *visited* of nodes already visited, and a node m, and returns all nodes reachable from m without ever crossing one a node k with

$k < n$ or a node in the *visited* list.

```
function non-smaller-reachable-nodes :: 'g ⇒ 'node ⇒ 'node list ⇒ 'node ⇒ 'node
list
  where non-smaller-reachable-nodes g n visited m = (
    if m ∈ set visited ∨ m ∉ set (αn g) ∨ (dfs-id g m) < (dfs-id g n) then
      []
    else
      m # concat
        (List.map (non-smaller-reachable-nodes g n (m#visited) ∘ getTo)
        (outEdges g m))
    )
```

As we're adding the nodes we've already visited to the list *visited* used for the recursive call, the number of nodes we avoid increases with each recursive step. Since all CFGs handled by this thesis are finite, this can only happen a finite number of times, and we can guarantee that this function always terminates. We can use *non-smaller-reachable-nodes* to compute all nodes reachable by a node n without ever crossing a smaller node by choosing the parameters $m = n$ and $visited = []$. Figures 4 to 6 show an example of this computation.

With *non-smaller-reachable-nodes* we have everything we need to define the semi-postdominator function *spdom*:

```
fun spdom :: 'g ⇒ 'node ⇒ 'node option
  where spdom g n = (
    if n = Exit then
      None
    else
      Some (
        let reachable-outEdges = concat (
              List.map (outEdges g)
              (n#(non-smaller-reachable-nodes g n [] n)))
        in min-target-node g (hd reachable-outEdges) reachable-outEdges
      )
    )
```

As the result of *non-smaller-reachable-nodes g exit-path n* won't include any nodes smaller than n, applying *outEdges* to this list will still leave the resulting node list filled with viable candidates for the semi-postdominator (as no further restrictions are placed on the last node of a candidate path)

while decreasing the minimum node in the list[11]. Thus, the function *spdom* performs as needed and computes a node's semi-postdominator.

4.3.2 Immediate Postdominators

Lengauer et al. proved the following identity [20][12]:

> Let n be a node different from $Exit$ and let u be a node for which $sdom(u)$ is minimum among nodes satisfying $n \xrightarrow{*} u \xrightarrow{+} spdom(n)$[13].

$$ipdom(n) = \begin{cases} spdom(n) & \text{if } spdom(n) = spdom(u) \\ ipdom(u) & \text{otherwise} \end{cases} \tag{4}$$

We can even restrict the definition of u further without violating this identity by choosing u to be the node with minimum *spdom* on the spanning tree path from w to $spdom(w)$: suppose the node u referenced above is not on the tree path from n to $spdom(n)$. Then the node k at which the spanning tree path from n to $spdom(n)$ and the path from n to u diverge will have $spdom(k) = spdom(u)$[14]. Thus the spanning tree path from n to $spdom(n)$ contains all nodes necessary for deciding whether $ipdom(n) = spdom(n)$.

With this result, we can compute immediate dominators using the following function:

```
fun smallest-spdom :: 'g ⇒ 'node ⇒ 'node list ⇒ 'node
  where smallest-spdom g accu [] = accu |
   smallest-spdom g accu (x#xs) = (
     if (spdom g accu = None) ∨
        ¬ (dfs-id g (the (spdom g accu)) < dfs-id g (the (spdom g x))) then
     smallest-spdom g x xs else
     smallest-spdom g accu xs
   )
```

[11]Every node has at least one outgoing edge: the spanning tree edge, which is known to lead to a node of smaller DFS-ID.

[12]The version given here has already been adjusted to fit immediate *postdominators*, while their version spoke of immediate dominators.

[13]The notation $n \xrightarrow{+} m$ means "there exists a nonnull path from n to m", $n \xrightarrow{*} m$ also allows null paths (i.e. $n = m$ is allowed).

[14]This is true as all nodes from k to u are not in the spanning tree path from k to $spdom(k)$ and because u has minimal *spdom* of all relevant nodes.

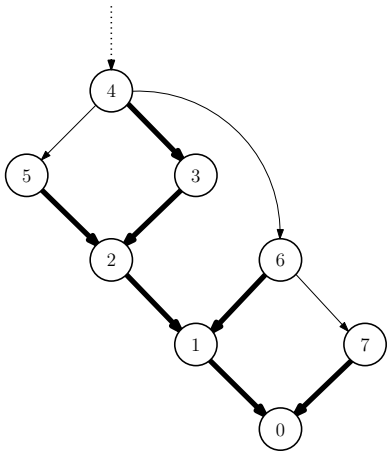


Figure 4: An example graph to show the execution of *non-smaller-reachable-nodes* as used by *spdom* on node 4.

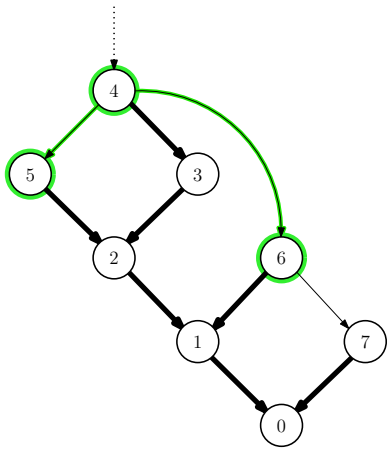


Figure 5: The green nodes are those found by *non-smaller-reachable-nodes* considering a recursion depth of 1.

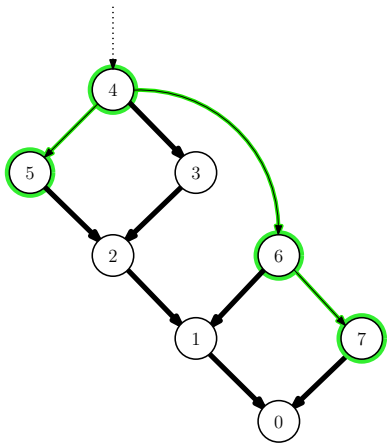


Figure 6: The nodes found considering a recursion depth of 2. As no new nodes greater than 4 are reachable, the function terminates, returning all nodes colored in green.

The function *smallest-spdom* takes a node as accumulator seed and a list of nodes and performs a fold over the list selecting for smaller *spdom*. It thus returns the node with the smallest *spdom* of seed and the list. In this respect, it is very much like *min-target-node*. With this, defining a function returning a node's *ipdom* is fairly straightforward:

```
function ipdom :: 'g ⇒ 'node ⇒ 'node option
  where ipdom g n = (
    if (spdom g n = None) ∨ (tree-path-nodes g n (the (spdom g n)) = None)
    then
      None
    else
      (let spdom-path =
      butlast (the (tree-path-nodes g n (the (spdom g n))));
      relevant-node = smallest-spdom g spdom-path in
      if dfs-id g (the (spdom g relevant-node)) = dfs-id g (the (spdom g n))
      then
        spdom g n
      else
        ipdom g relevant-node
      )
    )
```

This function always terminates because we know *relevant-node* to be in the spanning tree path from n to $spdom(n)$ and (in the case of the recursive application of *ipdom*) to be different from n. Together, we have that in the recursive case, *relevant-node* is always smaller than n, so it must at some point terminate (as DFS-ID is defined on natural numbers only).

Figure 7 shows how *spdom* and *ipdom* behave on our running example.

4.4 Computation of Control Dependencies

Our algorithm computes control dependencies matching the definition of standard control dependence outlined by Wolfe [26] (proven by Wasserrab to be equivalent to the definition of Ferrante et al. [8]):

> A node n' is control dependent on a node n, if n has at least two successors, one postdominated by n', while the other one is not.

It can easily be seen that control dependencies on a node n can only exist for the nodes on the paths from n to $ipdom(n)$ excluding n and $ipdom(n)$ themselves. As n's postdominators are all on the spanning tree path from

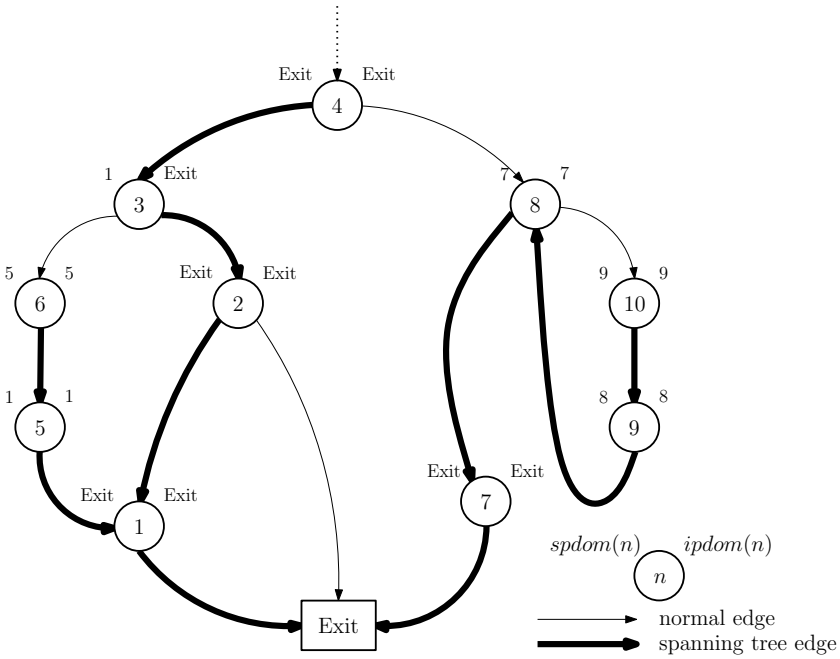


Figure 7: Our CFG, along with the information extracted from the depth-first search (node numbering and spanning tree) and semi-postdominator and immediate dominator. Take special note of node 3, which is the only node with differing *spdom* and *ipdom*.

n to *Exit* and as the immediate dominator is unique for every node, they can be ordered by increasing DFS-ID. In this ordering, each node in the list is the immediate postdominator of the next (per definition of immediate postdomination), starting with *Exit* up to n. This also means that we can iterate over all postdominators of n by chaining applications of *ipdom*. This, together with the fact that all nodes that may be control dependent on n must occur on some path between n and $ipdom(n)$[15], provides us with an efficient way of iterating over all nodes which are standard control dependent on any node n.

The function *cdeps-in-chain* shown below returns the list of all tuples (k, n) such that k postdominates m and $ipdom(n)$ postdominates k if given a node m and a direct predecessor[16] n as parameters. For fixed n, this coin-

[15]By $ipdom(n)$, all diverging paths starting at n must have converged again by definition of postdomination.

[16]This refers to the predecessor in the sense of the CFG, not the spanning tree.

cides exactly with all tuples (k, n) such that k is standard control dependent on n.

```
function cdeps-in-chain :: 'g ⇒ 'node ⇒ 'node ⇒ ('node × 'node) list
  where cdeps-in-chain g n topnode = (
    if (ipdom g topnode = Some n) ∨
       (topnode = n) ∨
       (ipdom g topnode = None) ∨
       (ipdom g n = None)
    then
      []
    else
      (n, topnode) # cdeps-in-chain g (the (ipdom g n)) topnode
    )
```

As *cdeps-in-chain* follows the *ipdom* chain on every recursive call, execution will sooner or later encounter *ipdom*(*topnode*) or *Exit* (for which *ipdom g n* = *None* is true), so we can guarantee termination for every possible input.

The next function, *cdeps-from-node*, provides the last piece needed for computing standard control dependencies for arbitrary nodes: for a node n, it calls *cdeps-in-chain* for all of n's successors with n as the *topnode* parameter, combining the result into a single list of all tuples (k, n) with k is standard control dependent on n.

```
fun cdeps-from-node :: 'g ⇒ 'node ⇒ ('node × 'node) list
  where cdeps-from-node g n =
   concat (List.map (λm. cdeps-in-chain g m n) (List.map getTo (outEdges g n)))
```

The function *control-deps* applies the computation of standard control dependencies to all nodes in the graph and concatenates the results into a single list:

```
fun control-deps :: 'g ⇒ ('node × 'node) list
  where control-deps g = concat (List.map (cdeps-from-node g) (αn g))
```

As we're interested in having the possibility of delivering the result of this computation as a control dependence graph, the final two functions discussed in this section will provide just this functionality: *tuple-list-to-graph* converts a list of tuples to a graph, leaving the edge descriptor of type *unit*, and *control-dependence-graph* is essentially an abbreviation for performing this computation on a graph, yielding the control dependence graph.

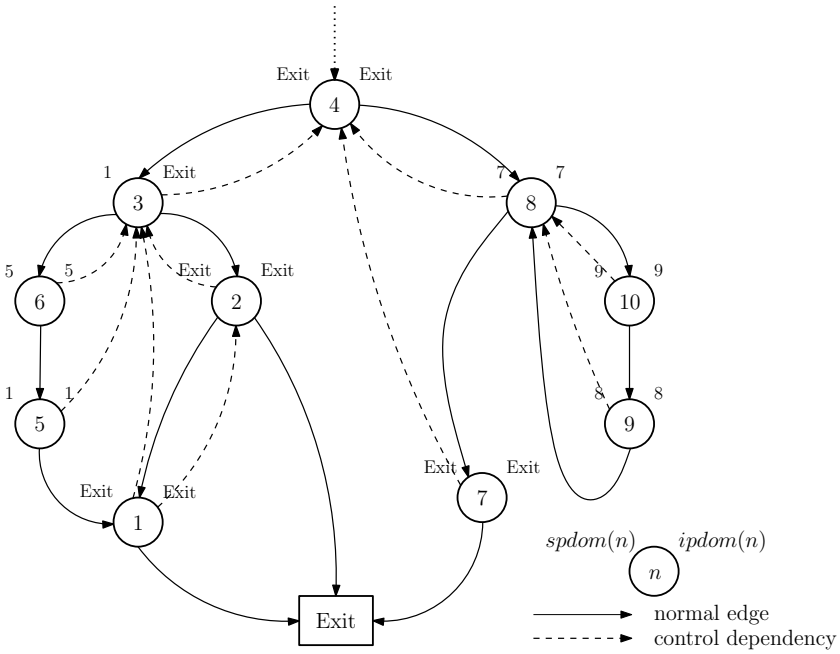


Figure 8: The CFG overlaid with some of the information we have extracted: For each node, its DFS-ID, *spdom*, *ipdom*, and control dependence edges are shown.

```
fun tuple-list-to-graph :: ('node × 'node) list ⇒ 'outg ⇒ 'outg
  where
  tuple-list-to-graph [] g = g |
  tuple-list-to-graph ((from,to)#xs) g = (
    if (from,(),to) ∈ αe'' g then
      tuple-list-to-graph xs g
    else
      tuple-list-to-graph xs (addEdge'' g from () to)
    )

definition control-dependence-graph :: 'g ⇒ 'outg
  where control-dependence-graph g = tuple-list-to-graph (control-deps g) empty''
```

Figure 8 shows the CFG previously constructed for our example overlaid with the control dependence graph generated by this function, and Figure 9 shows the result of all this combined work on the code of our running example.

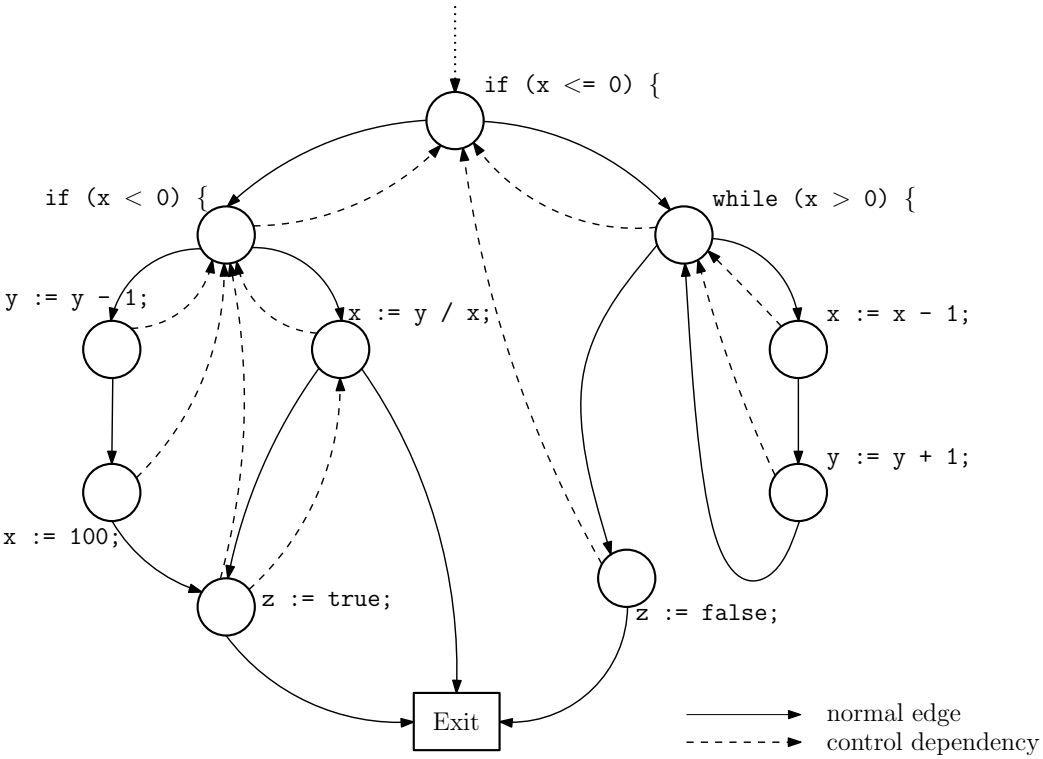


Figure 9: An annotated version of the CFG of our example showing that the control dependencies computed by this thesis are indeed the control dependencies present in the code.

5 Interpretations

An Isabelle locale that cannot be interpreted is useless for any practical purpose. Within the scope of this thesis, we supply two interpretations of our theory: One which works directly on flowgraphs and one which uses the CFGs constructed by the work of Kohlmeyer [17]. As the while language handled with this framework only includes structured control flow, the interpretation on raw graphs serves to illustrate the workings of the code of this thesis on nonreducible CFGs.

5.1 Graph Interpretation

As we have to ensure that the graphs we're using for this interpretation contain an Exit node, we first define a new datatype called *cfg-node* which differentiates between the special node *Exit* and any other node.

datatype $'n$ *cfg-node* = *Node* $'n$ | *Exit*

For the basic graph locales defined in Section 2.7, there is an implementation based on red-black trees readily available that we can extend and reuse. The functions provided by this implementation carry the names also used in this thesis, but are prefixed with "*mg-*", so the *invar* predicate provided by this package is called *mg-invar* and so on.

As we cannot guarantee all graphs that can be built by this framework to be valid CFGs, we restrict the graph type to a new type which describes only graphs that fulfill our assumptions. This is done using the *typedef* keyword, which requires us to supply a proof that the type to be defined is not empty.

typedef $('n,'ed)$ *sane-graph* = $\{g :: (('n::linorder)\ cfg\text{-}node,\ 'ed)\ graph.$
$\quad mg\text{-}invar\ g\ \wedge$
$\quad Exit \in set\ (mg\text{-}\alpha n\ g)\ \wedge$
$\quad (\forall n.\ (n \in set\ (mg\text{-}\alpha n\ g) \longrightarrow (\exists\ es.\ g \vdash n\ {-es\rightarrow}\ Exit)))\ \wedge$
$\quad (\forall es\ n.\ g \vdash Exit\ {-es\rightarrow}\ n \longrightarrow es = []\ \wedge\ n = Exit)\}$

The restrictions imposed on this type match the assumptions made by the locales *graph-exit* and *graph-controlDependencies* in Section 3, save for the assumption that all graphs are finite, which *mg-*αe already guarantees. In order to prove that this type is not empty, we provide one instance of a graph which fulfills these requirements: the graph containing only one edge with undefined edge descriptor from an undefined Node to *Exit*.

definition *sane-graph-instance*
where *sane-graph-instance = addEdge mg-empty (Node undefined) undefined Exit*

As *undefined* is an instance of every type, this graph is certainly of type *g :: (('n::linorder) cfg-node, 'ed) graph*. The empty graph *mg-empty* fulfills *mg-invar* and *mg-addEdge* preserves this, so *sane-graph-instance* also fulfills *mg-invar*. Since the set of nodes in *sane-graph-instance* contains only *Node undefined* and *Exit*, the other assertions are trivially true.

The *mg-* functions are defined on the type *('n, 'ed) graph* and not on the type *sane-graph*, hence the need for another set of functions fulfilling the role of *mg-addEdge*, *mg-αe* etc. Fortunately, we can fully automate the tedious task of redefining these functions to differently-typed variants by using the *lifting* package [15]. For example, for *mg-αn*:

setup-lifting *type-definition-sane-graph*

lift-definition *sg-αn* :: (('*n*::*linorder*), '*ed*) *sane-graph* ⇒ ('*n cfg-node*) *list* **is** *mg-αn*..

We continue this for all *mg-* functions, obtaining a full set of operations on this new graph type prefixed with *sg-* instead of *mg-*. Once we've done this, we can use our lifted definitions to finally interpret *graph-controlDependencies*:

interpretation *sg-conDep*: *graph-controlDependencies cfg-node.Exit sg-αe sg-invar sg-αn sg-inEdges sg-outEdges mg-αe mg-invar mg-αn mg-outEdges mg-addEdge mg-empty mg-αe mg-invar mg-addEdge mg-empty*

With this interpretation, we can finally execute the theory and generate code. In order to show that the running example we've been using isn't completely spurious, we can manually build exactly the graph we've been using and to execute our code on it[17]. The results are shown in Figures 10 to 12.

[17] As the while language used for the other interpretion only supports structured control flow, yielding a graph like this one is impossible with the other interpretation.

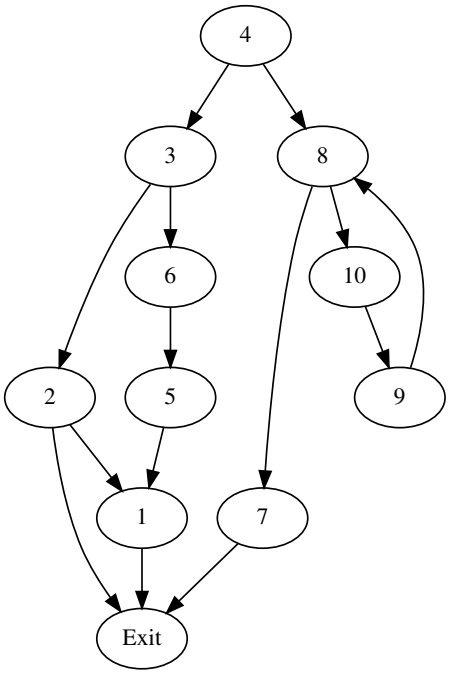


Figure 10: A handcrafted version of the CFG which represents our running example.

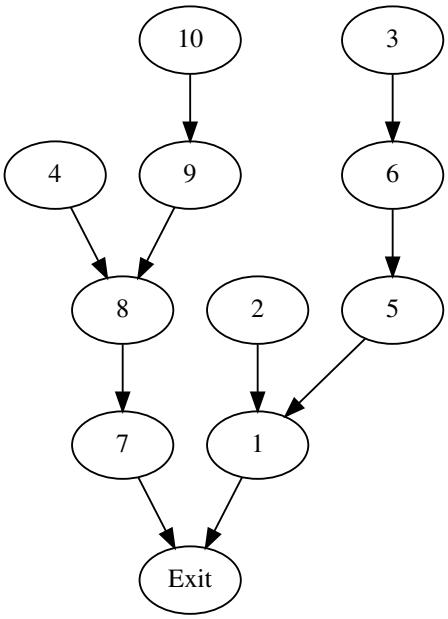


Figure 11: The spanning tree generated by DFS. This is a different spanning tree than the one shown on the other examples, but this won't influence control dependencies.

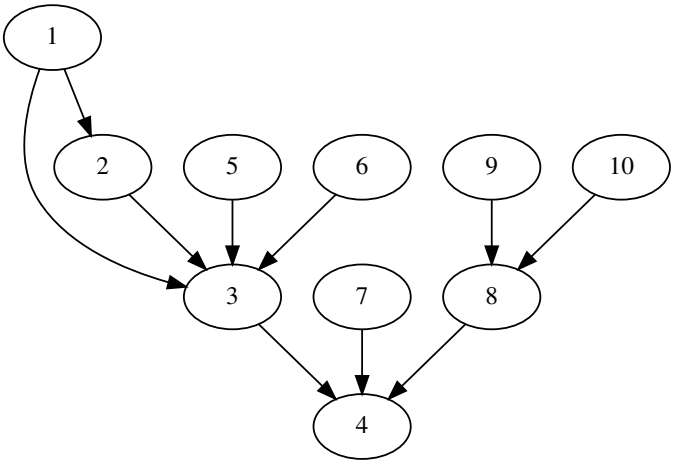


Figure 12: The control dependence graph of our example, as direct output of our generated code.

5.2 While Interpretation

The work of Kohlmeyer [17] provides a mechanism for generating CFGs from code in a simple while language. These CFGs not only supply an Exit node, but also an Entry node:

datatype *w-node* = *Node nat* ('('- - '-'))
| *Entry* ('('-*Entry*'-'))
| *Exit* ('('-*Exit*'-'))

The input "graph" type used represents abstract syntax trees (ASTs) of while programs and is called *cmd*, while all operations on this type use the function *build* to convert these ASTs to graphs, thus enabling us to work directly on programs:

abbreviation *while-edges c* ≡ *mg-αe* (*build c*)
abbreviation *while-nodes c* ≡ *mg-αn* (*build c*)
abbreviation *while-outEdges c* ≡ *mg-outEdges* (*build c*)
abbreviation *while-inEdges c* ≡ *mg-inEdges* (*build c*)
definition *while-invar c* ≡ *mg-invar* (*build c*)

With the type signature of *build* being:

build :: cmd ⇒ (w-node, state edge-kind) graph

The CFG generation theory supplies its own predicate *path* (noted "$g \vdash n$ -*es*→∗ n") for reasoning about paths in the generated CFGs, very much like the *path-to* predicate defined in *graph-path*. In fact, both are equivalent, a fact which we prove in order to reuse the lemmas about *path*:

lemma *path-equiv*: $(g \vdash n$ −*es*→∗ $n') \longleftrightarrow (g \vdash n$ −*es*→ $n')$

Using this equivalence, we can directly use some lemmas about *path* which directly coincide with the assumptions we need to prove to interpret our theory, for example[18]:

lemma *while-invar*: *while-invar c*
lemma *Exit-in-while-nodes*: (-*Exit*-) ∈ *set* (*while-nodes c*)
lemma *valid-node-Exit-path*:
assumes *valid-node g n* **shows** ∃ *as*. $g \vdash n$ −*as*→∗ (-*Exit*-)

[18] We shall not dwell on *valid-node g n*. Suffice it to say that it follows from the combination of the correctness of αn and "$n \in \alpha n\ g$".

Combining all these results finally lets us interpret the theory, using the *while-* functions for the input graph, and the *mg-* functions for the spanning tree graph type and output graph type.

interpretation *while-conDep*: *graph-controlDependencies* (*-Exit-*) *while-edges while-invar while-nodes while-inEdges while-outEdges mg-αe mg-invar mg-αn mg-outEdges mg-addEdge mg-empty mg-αe mg-invar mg-addEdge mg-empty*

After having interpreted the *graph-controlDependencies* locale, we can generate code for the theory in a variety of functional languages, e.g. Haskell:

export-code *while-conDep.control-dependence-graph* **in** *Haskell*

To deliver a concrete example of the code "in action", consider the following code snippet written in the while language:

```
if (y < (x - 1)) {
  if (x < y) {
    x := y;
  } else {
    y := x;
  }
  x := x;
} else {
  c := y;
}
y := y;
```

The CFG generated from this code snippet by Kohlmeyer's thesis is shown in Figure 13, and the control dependence graph computed by the exported Haskell code is shown in Figure 14

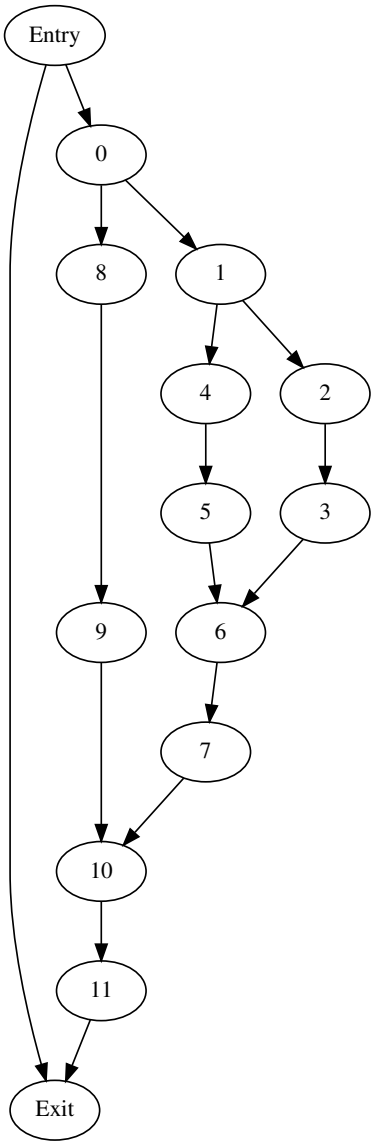


Figure 13: The CFG built from the code example. Note that not every node can directly be identified with a statement, as the conversion algorithm inserts "empty" nodes at around certain instruction types. Other than that, the CFGs generated by Kohlmeyer are regular CFGs with minimal basic blocks.

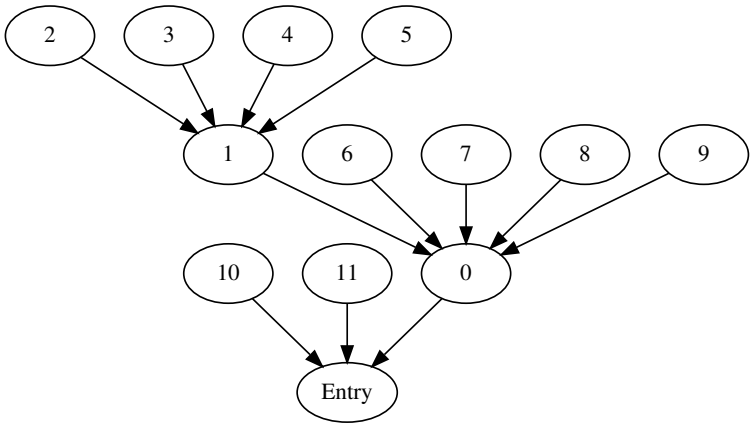


Figure 14: The control dependence graph computed by the exported Haskell code.

6 Related Work

While this thesis is the first work known to us to attempt the machine-assisted verification of an algorithm for computing the control dependence graph from a CFG, a lot of work has been done in recent years in related topics:

Neumann [21] provides a verified generic formalization of depth-first algorithms in Isabelle/HOL using a similar approach to the one used in this thesis which also supports efficient code generation. To the best of our knowledge, there is not yet a published and verified formalization of *disjoint-set* operations which could be used for improving the algorithm presented in this paper.

Cooper et al. [7] provide a simple iterative approach to computing dominators reaching an efficiency comparable to that of Lengauer-Tarjan in common cases. To the best of our knowledge, there is not yet a machine-checked proof for the correctness of their algorithm. Buchsbaum et al. [5] present linear-time algorithms for several graph-theoretical problems including computing dominators, thereby attempting to once and for all provide solutions to these problems that can both be called simple and efficient. Their algorithm for computing dominators might be a suitable candidate as a replacement for the dominator algorithm used in this thesis. The Haskell package Data.Graph.Dom[19] provides an implementation of Lengauer-Tarjan. While this supplies a functional implementation of Lengauer-Tarjan, it is not verified in any way and is written in a style using many monadic structures, and is as such unsuitable for porting to Isabelle/HOL.

Zhao and Zdancewic [27] provide an abstract specification of dominance analysis and supply verified implementations of two different approaches to computing dominance for the Vellvm project[20]. Their work uses the Cooper-Harvey-Kennedy algorithm [7] instead of the slightly faster Lengauer-Tarjan algorithm for computing postdominators. All proofs in their work are carried out using the Coq proof assistant. Harrold et al. [13] generalize control dependencies to interprocedural control dependencies and present an efficient algorithm for computing these dependencies. (This work only concerns itself with intraprocedural control dependencies)

As for work on topics that build upon the existence of control dependen-

[19] See http://hackage.haskell.org/package/dom-lt-0.1.3/docs/Data-Graph-Dom.html
[20] See http://www.cis.upenn.edu/~stevez/vellvm/

cies:

Horwitz et al. [14] present a technique for performing interprocedural slicing based on the information in dependence graphs. Hammer and Snelting [10] present a formalization of a flow-sensitive, context-sensitive, and object-sensitive technique for performing information flow control based on program dependence graphs, which Wasserrab and Lohner [23] draw upon to present a language independent machine-checked correctness proof for information flow noninterference based on interprocedural slicing.

7 Conclusion

In this thesis we presented a functional algorithm for computing standard control dependencies based on a variant of the classical Lengauer-Tarjan algorithm for finding dominators in a flowgraph along with a combination of pen & paper proofs and machine checked proofs for the correctness of this algorithm. The complete work is executable and can be exported into a variety of functional languages. As proven by construction in Section 5, the assumptions made by the used proof context are satisfiable.

Complications

Proving the necessary properties of the implementation (especially of the depth-first search) turned out to be more arduous than anticipated, and the lack of unconnected nodes in the used graph formalization further complicated this task. Our limited experience with theorem provers and Isabelle in particular certainly did not help, requiring us to learn the tools of the trade as we went along.

Future Work

In the future, this work can be extended by providing Isar proofs for those correctness properties only proven in this thesis by pen & paper or with old-style proof scripts, thus making this work into a maintainable and fully machine-checked building block. The *graph-controlDependencies* locales currently contains the fully verified implementation of DFS, the postdominator algorithm and the control dependency algorithm. In order to improve modularity and reusability of this work, this locale can be split up into its parts. In order to improve the performance of the generated code, the naive implementation of semi-postdominator and immediate postdominator computation can be replaced by a more efficient variant, and the work provided by this paper can be complemented by formalizing (and proving correct) an algorithm for computing data dependencies, thus providing more of the requirements for building a fully verified slicing framework.

References

[1] Alstrup, S., Harel, D., Lauridsen, P. W., and Thorup, M. (1999). Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132.

[2] Ballarin, C. (2004). Locales and locale expressions in Isabelle/Isar. In *Types for Proofs and Programs: International Workshop, LNCS 3085*, pages 34–50. Springer.

[3] Berghofer, S. and Wenzel, M. (1999). Inductive Datatypes in HOL - Lessons Learned in Formal-Logic Engineering. In *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, London, UK. Springer-Verlag.

[4] Boldyreva, A., Gentry, C., O'Neill, A., and Yum, D. H. (2007). Ordered multisignatures and identity-based sequential aggregate signatures, with applications to secure routing. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 276–285, New York, NY, USA.

[5] Buchsbaum, A. L., Georgiadis, L., Kaplan, H., Rogers, A., Tarjan, R. E., and Westbrook, J. (2008). Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573.

[6] Buchsbaum, A. L., Kaplan, H., Rogers, A., and Westbrook, J. R. (1998). A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems*, 20(6):1265–1296.

[7] Cooper, K. D., Harvey, T. J., and Kennedy, K. (2001). A simple, fast dominance algorithm. Available at `http://www.cs.rice.edu/~keith/Embed/dom.pdf`.

[8] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

[9] Georgiadis, L. and Tarjan, R. E. (2004). Finding dominators revisited: extended abstract. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 869–878, Philadelphia, PA, USA.

[10] Hammer, C. and Snelting, G. (2009). Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422.

[11] Harel, D. (1985). A linear algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC, pages 185–194, New York, NY, USA.

[12] Harman, M. and Hierons, R. M. (2001). An overview of program slicing. *Software Focus*, 2(3):85–92.

[13] Harrold, M. J., Rothermel, G., and Sinha, S. (1998). Computation of interprocedural control dependence. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis*, pages 11–20, New York, NY, USA.

[14] Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60.

[15] Huffman, B. and Kunčar, O. (2012). Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Proceedings of the 3rd International Conference on Certified Programs and Proofs*.

[16] Hwang, J. Y., Lee, D. H., and Yung, M. (2009). Universal forgery of the identity-based sequential aggregate signature scheme. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 157–160, New York, NY, USA.

[17] Kohlmeyer, K.-S. (2012). Funktionale Konstruktion und Verifikation von Kontrollflussgraphen. Bachelor's thesis, Karlsruher Institut für Technologie (KIT).

[18] Krauss, A. (2009). *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, München.

[19] Lattner, C. (2002). Llvm: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, Computer Science Dept.

[20] Lengauer, T. and Tarjan, R. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1:121–141.

[21] Neumann, R. (2012). A framework for verified depth-first algorithms. In *Proceedings of the Workshop on Automated Theory Exploration (ATX 2012)*, pages 36–45.

[22] Wasserrab, D. (2010). *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik.

[23] Wasserrab, D. and Lohner, D. (2010). Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop - VERIFY-2010.*

[24] Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA.

[25] Wenzel, M. (2002). *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD thesis, Technische Universität München.

[26] Wolfe, M. J. (1995). *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[27] Zhao, J. and Zdancewic, S. (2012). Mechanized verification of computing dominators for formalizing compilers. In *Proceedings of the Second international conference on Certified Programs and Proofs*, CPP'12, pages 27–42.

Erklärung

Hiermit erkläre ich, Maximilian Wagner, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift