Formalizing a Framework for Dynamic Slicing of Program Dependence Graphs in Isabelle/HOL

Daniel Wasserrab and Andreas Lochbihler[*]

Universität Karlsruhe,
{wasserra,lochbihl}@ipd.info.uni-karlsruhe.de

Abstract. Slicing is a widely-used technique with applications in e.g. compiler technology and software security. Thus verification of algorithms in these areas is often based on the correctness of slicing, which should ideally be proven independent of concrete programming languages and with the help of well-known verifying techniques such as proof assistants. As a first step in this direction, this contribution presents a framework for dynamic slicing based on control flow and program dependence graphs and machine checked in Isabelle/HOL. Abstracting from concrete syntax we base the framework on a graph representation of the program fulfilling certain structural and well-formedness properties.

1 Introduction

Slicing is a widely-used technique with applications in e.g. compiler technology, debugging and software security. Thus, many algorithms in these areas rely on the different variants of slicing being correct. Suppose there was a tool with which to prove slicing correct. Clearly, this would immensely faciliate verification of such algorithms. Ideally, such a tool would not be restricted to a specific programming language or syntax and utilize well-known verification techniques such as proof assistants. In this contribution, to tackle a subtask of this idea, we present a framework for dynamic slicing based on control flow and program dependence graphs and machine-checked in Isabelle/HOL.

In aiming for versatility, our approach rests upon a special graph representation in the style of control flow graphs (CFG) and not on concrete syntax. If such a representation fulfills certain basic structural and well-formedness properties, we call it trace control flow graph (TCFG) and associate a path of edges to every program trace. For this graph, which can even be infinite, we define the program dependence graph (PDG) using the standard notions of control and data dependence. Then we compute the backward slice and obtain the sliced path from the original TCFG path by invalidating (i.e. replacing them with an operation doing nothing) all operations triggered by nodes that are not in the backward slice. Our main theorem shows that executing the remaining operations on a sliced path yields the same result as performing those on the initial path w.r.t. the variables used at the target node for every suitable input to the program trace.

In the second part, we also present how to embed a simple While language (without procedures) in the framework. On the one hand, we illustrate this way both how to construct a trace control flow graph for a semantics and how to validate the required

---

well-formedness properties using the programming language semantics. On the other hand, this demonstrates that these properties are indeed sensible and chosen well.

1.1 Slicing

To collect all program points that can influence a certain statement in a program a program analysis called *slicing* was defined by Weiser [18]. There are many approaches to how to accomplish this task, for an overview, see Tip [15] or Krinke [8]. Our formalization uses the graph-based approach in relying on the dynamic equivalents of control flow (CFG) and program dependence graphs (PDG). CFGs consist of nodes denoting the program statements and edges that represent the order, in which these statements are executed. Two relations on these nodes, called data and control dependence, determine the edges, which connect the CFG nodes to constitute the PDG. A *backward slice* of a node is then defined as the set of all nodes on which the given node is transitively data and control dependent. This set is a conservative approximation of all program points that can influence this statement. Our approach is dynamic as we can have infinite graphs (e.g. because of method inlining) and we only apply slicing to paths which are executable in a certain state, not to the graph as a whole.

1.2 Isabelle

The formalization is written completely in Isabelle/HOL [12], including all lemmas and theorems, i.e. every single proof is machine-checked. To be as generic as possible with respect to possible languages/CFGs instantiating the framework, we extensively use the Locales concept in Isabelle [3]. This encapsulation enables one to write generic functions and predicates, limited only by imposing certain constraints on them.

Isabelle also provides the means to automatically generate LaTeX documents (such as this one) based on Isabelle input files by automatically replacing references to definitions and lemmas in the LaTeX text with the respective pretty-printed and typeset formulae. This avoids typing errors introduced via the transfer from Isabelle to LaTeX files and shows the convenient, human readable syntax of the formalization.

1.3 Notation

Types include the basic types of truth values, natural numbers and integers, which are called *bool*, *nat*, and *int* respectively. The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. $t::\tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

Sets (type $'a\ set$) follow the usual mathematical convention.

Lists (type $'a\ list$) come with the empty list [], the infix constructor ·, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in "s" usually stand for lists and $|xs|$ is the length of *xs*. If $i < |xs|$ then $xs_{[i]}$ denotes the i-th element of *xs*. The standard function *map*, which applies a function to every element in a list, is also available.

datatype $'a$ *option* $=$ *None* | *Some* $'a$ adjoins a new element *None* to a type $'a$. All existing elements in type $'a$ are also in $'a$ *option*, but are prefixed by *Some*. Hence *bool option* has the values *Some True*, *Some False* and *None*.

Case distinctions on data types use guards, where every guard must be followed by a data type constructor. E.g. *case x of Some y* $\Rightarrow$ *f y* | *None* $\Rightarrow$ *g* means that if *x* is some *y* then the result is *f y* where *f* may refer to value *y*, and if *x* is *None*, then the result is *g*.

Partial functions are modeled as functions of type $'a \Rightarrow 'b$ *option*, where *None* represents undefinedness and *f x* = *Some y* means *x* is mapped to *y*. Instead of $'a \Rightarrow 'b$ *option* we write $'a \rightharpoonup 'b$ and call such functions **maps**.

Function update is defined as follows: $f(a := b) \equiv \lambda x.$ *if* $x = a$ *then* b *else* $f x$, where $f :: {}'a \Rightarrow {}'b$ and $a :: {}'a$ and $b :: {}'b$.

2   The Framework

Our framework is generic, i.e. we do not restrict ourselves to a specific programming language (in fact, not even to a certain programming paradigm such as imperative or object oriented programming). To construct a PDG, on which to perform dynamic slicing, we instantiate the framework with a so called *trace control flow graph*, representing code from any source code language. E.g. this trace CFG can be obtained by inlining the CFG for every procedure at the respective calling site in a certain program, thus being potentially infinite (e.g. if we have a recursive function). The constraints on this trace CFG has to fulfill are described in detail in the next section.

2.1   The Input Trace Control Flow Graph

The trace CFG (called TCFG in the following) consists of nodes of type $'node$ and edges of type $'edge$, with an edge *a* being in the set of edges if it fulfills some property *valid-edge a*, a parameter of the instantiating language. Using the functions *sourcenode*, *targetnode* and *kind*, we can determine the source node, target node and edge kind of an edge, respectively. An edge kind describes the action taken when traversing this edge, so we have two different edge kinds of type $'state$ *edge-kind*, both parameterized with a state (or *context* as it is called in other programming languages) type variable $'state$: updating the current state with a function $f :: {}'state \Rightarrow {}'state$, written $\Uparrow f$, or assuring that predicate $Q :: {}'state \Rightarrow bool$ in the current state is fulfilled, written $(Q)_{\sqrt{}}$. We define a function *transfer* such that traversing an edge in a state *s* then means that we apply this function to the corresponding edge kind, thus calculating *f s*, if we have an update edge $\Uparrow f$, otherwise leaving the state unchanged. Function *pred* determines if predicate *Q* of a predicate edge $(Q)_{\sqrt{}}$ in a certain state *s* is fulfilled, returning *True* for update edges.

A node *n* is in the node set of a TCFG, if it fulfills the property *valid-node n* with the following definition: *valid-node n* $\equiv \exists a.$ *valid-edge a* $\wedge$ (*n* = *sourcenode a* $\vee$ *n* = *targetnode a*). Furthermore we need two special nodes (*-Entry-*), which may only have outgoing edges, and (*-Exit-*), having incoming edges only. Also there is a special edge which has (*-Entry-*) as source and (*-Exit-*) as target node, the respective edge kind being $(\lambda s.\ False)_{\sqrt{}}$, a predicate which can never be fulfilled. The last restrictions on the TCFG are that we do not allow multi edges nor self loops, so if the source and target nodes of two

$$\dfrac{}{Def\ (\text{-}Entry\text{-}) = \emptyset \wedge Use\ (\text{-}Entry\text{-}) = \emptyset} \qquad \dfrac{}{Def\ (\text{-}Exit\text{-}) = \emptyset \wedge Use\ (\text{-}Exit\text{-}) = \emptyset}$$

$$\dfrac{valid\text{-}edge\ a \qquad V \notin Def\ (sourcenode\ a)}{state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ s\ V} \qquad \dfrac{valid\text{-}edge\ a \qquad kind\ a = (Q)_{\sqrt{}}}{Def\ (sourcenode\ a) = \emptyset}$$

$$\dfrac{valid\text{-}edge\ a \qquad \forall\,V\in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V}{\forall\,V\in Def\ (sourcenode\ a).\ state\text{-}val\ (transfer\ (kind\ a)\ s)\ V = state\text{-}val\ (transfer\ (kind\ a)\ s')\ V}$$

$$\dfrac{valid\text{-}edge\ a \qquad pred\ (kind\ a)\ s \qquad \forall\,V\in Use\ (sourcenode\ a).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V}{pred\ (kind\ a)\ s'}$$

Fig. 1. The well-formedness properties of a trace CFG

valid edges coincide, so do the two edges, and the source and target node of no valid edge may be equal.

Based on these constraints, we can now specify a property *inner-node n*, which defines the subset of valid nodes not being (*-Entry-*) or (*-Exit-*). We also allow to combine edges in paths, written $n -as\rightarrow* n'$, meaning that node *n* can reach *n'* via edges *as*::$'edge$ *list*. These paths are constructed via the following two rules:

$$\dfrac{valid\text{-}node\ n}{n -[]\rightarrow* n} \qquad \dfrac{n'' -as\rightarrow* n' \qquad valid\text{-}edge\ a \qquad sourcenode\ a = n \qquad targetnode\ a = n''}{n -a{\cdot}as\rightarrow* n'}$$

Moreover we define *sourcenodes*, *targetnodes* and *kinds* as mappings of the respective functions to edge lists using *map*. Using these, we can show that the source node of an edge in a TCFG path edge list matches the target node of the edge preceding it, and the target node of an edge matches the source node of the edge succeeding it.

We also lift *transfer* and *pred* to lists of edge kinds via

$$\begin{aligned} transfers\ []\ s &= s & transfers\ (e{\cdot}es)\ s &= transfers\ es\ (transfer\ e\ s) \\ preds\ []\ s &= True & preds\ (e{\cdot}es)\ s &= pred\ e\ s \wedge preds\ es\ (transfer\ e\ s) \end{aligned}$$

After having defined the structural properties of the TCFG, we furthermore need some well-formedness properties for its edges and the *Def* and *Use* sets of the source nodes of its edges, which collect the defined and used variables in this node, respectively, and a function *state-val s V* returning the value currently in variable *V* in state *s* (the formal rules are shown in Fig. 1):

- *Def* and *Use* sets of (*-Entry-*) and (*-Exit-*) are empty,
- if the source node of an edge does not define a variable *V*, the value of *V* in the state after traversing the edge is the same as its value in the initial state,
- the source node of a predicate edge does not define any variables.
- if two states agree on all variables in the *Use* set of the source node of an edge, after traversing this edge the two states agree on all variables in the *Def* set of this node, so different values in the variables not in the *Use* set cannot influence the values of the variables in the *Def* set,
- if two states agree on all variables in the *Use* set of the source node of a predicate edge and this predicate is valid in one state, it is also valid in the other one,

If we also have a semantics of the language – where $\langle c,s \rangle \Rightarrow \langle c',s' \rangle$ means that evaluating statement c in state s results in fully evaluated statement c' and final state s' – and a mapping from a node n to its corresponding statement c via n *identifies* c, we have another well-formedness property (called *semantically well-formed*):

$$\frac{n \text{ identifies } c \qquad \langle c,s \rangle \Rightarrow \langle c',s' \rangle}{\exists\, n' \text{ as. } n -as \rightarrow * \, n' \wedge \text{ transfers } (\text{kinds as}) \, s = s' \wedge \text{ preds } (\text{kinds as}) \, s \wedge n' \text{ identifies } c'}$$

This property states that if the complete evaluation of statement c in state s results in a state s' and node n corresponds to statement c, then there is a path in the TCFG beginning at n, on which, taking s as initial state, all predicates in predicate edges hold and the traversal of the path edge kinds also yields state s'.

2.2 Constructing the Program Dependence Graph

Though we instantiate the framework with a possibly infinite trace CFG, we call the data structure constructed in the following program dependence graph (PDG), as we are using the standard definitions of control and data dependence described e.g. in [15, Sec. 2]. For control dependence, the trace CFG must contain all possible paths and not only a special one, i.e. a trace.

Control Dependence: First we formalize the notion of *postdomination*. A node n' postdominates a node n, if both are valid nodes and n' must lie on every path from n to (*-Exit-*). Note: If no path from n to (*-Exit-*) exists (e.g. because of unstructured control flow), every valid node postdominates n.

Definition 1. *(Postdomination)*

$$n' \text{ postdominates } n \equiv \text{valid-node } n \wedge \text{valid-node } n' \wedge$$
$$(\forall \text{ as. } n -as \rightarrow * (\text{-Exit-}) \longrightarrow n' \in \text{set } (\text{sourcenodes as}))$$

Using this notion, control dependence is straightforward (see [19], Sec. 3.3). A node n' is control dependent on a node n via edges as, iff there is a path from n to n' using edges as and n' postdominates the targetnode of the first edge of as but does not postdominate the targetnode of another valid edge a' leaving n. The notion $n' \notin \text{set } (\text{sourcenodes as})$ states that as is a minimal path without loops starting at n'.

Definition 2. *(Control Dependence)*

n controls n' via $as \equiv n' \notin \text{set } (\text{sourcenodes as}) \wedge n -as \rightarrow * \, n' \wedge (\exists\, a \, a' \, as'. \, as = a \cdot as' \wedge$
$\quad\quad\quad\quad \text{sourcenode } a = n \wedge n' \text{ postdominates targetnode } a \wedge \text{ valid-edge } a' \wedge$
$\quad\quad\quad\quad \text{sourcenode } a' = n \wedge \neg \, n' \text{ postdominates targetnode } a')$

In Lem. 1 we prove that this definition is equivalent to the one given in [15, Sec. 2], which says that node n' is control dependent on node n via edges as, iff there is a path from n to n' using edges as and n' postdominates every node on this path except n – since every node in this path except n is a target node of an edge of this edge list as, we can rewrite this proposition using *targetnodes* – and n' does not postdominate n. Property $n' \notin \text{set } (\text{sourcenodes as})$ again guarantees minimal paths without loops starting at n' and we have to explicitly forbid n being the (*-Exit-*) node.

Lemma 1. *(Control Dependence Variant)*

$$n \text{ controls } n' \text{ via } as \equiv n -as \rightarrow * \, n' \wedge \neg \, n' \text{ postdominates } n \wedge$$
$$(\forall \, n'' \in set \, (targetnodes \, as). \, n' \text{ postdominates } n'') \wedge$$
$$n' \notin set \, (sourcenodes \, as) \wedge n \neq (\text{-Exit-})$$

The next lemma shows that every inner node *n*, which is reachable via a path from (*-Entry-*) and has a path leading to (*-Exit-*), has a node n' on which it is control dependent. Note that there may be more than one such node if control flow is unstructured.

Lemma 2. *(Control Dependence Predecessor)*

$$\frac{\textit{inner-node } n \qquad (\text{-Entry-}) -as \rightarrow * \, n \qquad n -as' \rightarrow * (\text{-Exit-})}{\exists \, n' \, as. \, n' \text{ controls } n \text{ via } as}$$

Data Dependence: Node n' is data dependent on a node *n*, iff there is a variable *V* which gets defined at *n* and used at n' and there is a path from *n* to n' using a nonempty list of edges *as* such that no node in the path redefines *V*.

Definition 3. *(Data Dependence)*

$$n \text{ influences } V \text{ in } n' \text{ via } as \equiv V \in Def \, n \wedge V \in Use \, n' \wedge n -as \rightarrow * \, n' \wedge$$
$$(\exists \, a' \, as'. \, as = a' \cdot as' \wedge (\forall \, n'' \in set \, (sourcenodes \, as'). \, V \notin Def \, n''))$$

This definition forbids data dependences from a node to itself as well as source node *n* occurring twice on path *as*.

The PDG: A PDG consists of edges of two different types: control dependence edges $n -as \rightarrow_{cd} n'$ and data dependence edges $n -\{V\}as \rightarrow_{dd} n'$. The definitions are straightforward, using the definitions above:

Definition 4. *(PDG Edges)*

$$\frac{n \text{ controls } n' \text{ via } as}{n -as \rightarrow_{cd} n'} \qquad \frac{n \text{ influences } V \text{ in } n' \text{ via } as}{n -\{V\}as \rightarrow_{dd} n'}$$

A path in the PDG using edges *as*, written $n -as \rightarrow_d * \, n'$, is constructed by concatenating PDG edges (if the respective source and target nodes match), *as* being the concatenation of the CFG edge lists of all PDG edges.

2.3 Dynamic Backward Slicing with respect to a Node

In the standard understanding of slicing in dependence graph-based approaches, the *slicing criterion* corresponds to a node; thus, we call this node *slicing node*. In the following, we compute a dynamic backward slice of a node n' and show that the slice fulfills the fundamental property of dynamic slicing in Theorem 1, i.e. for all variables used in n' traversing the sliced path returns the same result as traversing the respective non-sliced TCFG path. Since we formalise dynamic backward slices as the backward traversal of PDG edges and do not model def-def dependence edges, we only regard variables in the *Use* set of the slicing node, not those in the *Def* set.

Computing a Dynamic Path Slice: The only relevant information the slice of a path must provide is if a certain edge gets included in it or not – as their respective edge kind carries the transition information in this framework. Thus we can model a path slice as a bit vector, i.e. a *bool list*, of the same length as the edge list *as* of the path, being *True* at position i iff the edge at position i of edge list *as* has to be considered. An edge has to be considered if its source node has a PDG path to the slicing node n' with an edge list corresponding to the according suffix of *as*. Function *slice-path as* computes this bit vector by traversing the edge list *as*. Note that the last node of the reduced path being the slicing node n' is invariant throughout this computation:

Definition 5. *(Dynamic Path Slice)*

$$slice\text{-}path\ [] \equiv []$$
$$slice\text{-}path\ (a{\cdot}as) \equiv let\ n' = last(targetnodes\ (a{\cdot}as))\ in$$
$$(sourcenode\ a\ -a{\cdot}as\rightarrow_d *\ n'){\cdot}slice\text{-}path\ as$$

The fact that we only consider PDG paths via the executed CFG edge list makes this slicing dynamic, static slicing would consider all possible dependences, i.e. also dependences via other CFG edge lists.

Bit vectors can be compared via the less-or-equal relation $\preceq_b$, where $bs \preceq_b bs'$ holds if $|bs| = |bs'|$ and bs' is *True* at least at those elements where bs is *True*. Thus we can say that bs contains less or equal information on the former edge list than bs'. The maximal elements for relation $\preceq_b$, are the bit vectors which are *True* at every entry.
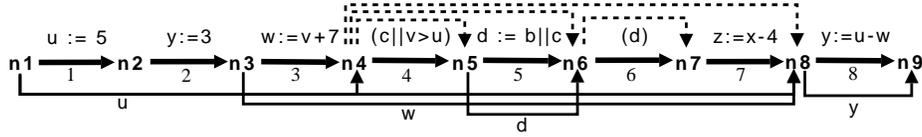
To obtain the edge kind list of the sliced path, we have to combine the bit vector with the initial edge list of the path. If the bit vector is *True* for a certain edge, we copy the respective edge kind in the list, otherwise, if the edge kind is an update edge, we include $\Uparrow id$ which does not alter the state, if it is a predicate edge, we include $(\lambda s.\ True)_{\sqrt{}}$ being *True* in any state. Again the calculation is done via iteration over the edge list:

Definition 6. *(Edge Kind List for a Sliced Path)*

$$select\text{-}edge\text{-}kinds\ []\ [] \equiv []$$
$$select\text{-}edge\text{-}kinds\ (a{\cdot}as)\ (b{\cdot}bs) \equiv (if\ b\ then\ kind\ a$$
$$else\ (case\ kind\ a\ of\ \Uparrow f \Rightarrow \Uparrow id\ |\ (Q)_{\sqrt{}} \Rightarrow (\lambda s.\ True)_{\sqrt{}})))$$
$${\cdot}select\text{-}edge\text{-}kinds\ as\ bs$$
$$slice\text{-}kinds\ as \equiv select\text{-}edge\text{-}kinds\ as\ (slice\text{-}path\ as)$$

See Fig. 2 for an example of a path, the bit vector representing its sliced path and the edge kind list of the respective sliced path. In this and all following examples, we number edges and use a simplified language where we replaced the update functions with easier to understand variable assignments, writing predicates as boolean expressions over variables in brackets without the $_{\sqrt{}}$. Data dependences are indicated with solid arrows and annotated with the respective variable name, control dependence edges with dashed arrows. All the variables on the right hand side of an assignment as well as those used in a predicate are in the *Use* set of the source node of the respective edge, the variables on the left hand side of an assignment constitute its *Def* set.

Dependent Live Variables: Our next aim is to prove that dynamic path slicing is indeed correct, i.e. traversing the edge kinds in the sliced path returns the same result as

u := 5   y:=3   w:=v+7   (c||v>u)   d := b||c   (d)   z:=x-4   y:=u-w

n1 →1→ n2 →2→ n3 →3→ n4 →4→ n5 →5→ n6 →6→ n7 →7→ n8 →8→ n9

u   w   d   y

Assumption: *Use* **n9** = { x, y }

Bit vector representing the sliced path: [*True*, *False*, *True*, *True*, *False*, *False*, *False*, *True*]

Sliced path edge kind list: [u:=5, **id**, w:=v+7, (c||v>u), **id**, **True**, **id**, y:=u-w]

Fig. 2. Example of how to calculate the edge kinds of a sliced path with slice node **n9**

traversing those on the original path for all variables used in the slicing node and if all predicates on the original path are satisfiable, so are they in the sliced path. An auxiliary definition to reach this goal is the notion of *dep-live-vars n′ as*, a collection of variables whose altering can change the value of a variable used in node n'. In fact, we compute a kind of *Live Variables Analysis* as described in [11], restricted to only one path and ignoring those nodes, on which the parameter node is not (transitively) dependent.

But collecting just the variables is not enough for our goal, we furthermore need information about via which TCFG edges we can reach the node where this variable was used (and not redefined in between) starting from our current position in the path. Hence it is possible to have the same variable multiple times in the set, but the respective edge list component differs. *dep-live-vars* has two parameters, first the node n' for which this calculation is made, and second the edge list *as* we have already traversed previous the iterations (note that this traversal happens from right to left). Example: $(V, as') \in$ *dep-live-vars n′ as* states that a node where V is used can be reached via edges as' from the node from which edges *as* lead to n' and no node on as' redefines V. The formal rules for *dep-live-vars*:

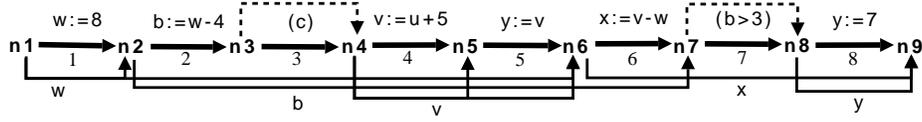**Definition 7.** *(Dependent Live Variables)*

$$\frac{V \in Use\ n'}{(V, [])\ \in\ dep\text{-}live\text{-}vars\ n'\ []}$$

$$\frac{V \in Use\ (sourcenode\ a) \qquad sourcenode\ a\ -a \cdot as' \rightarrow_{cd} n'}{(V, [])\ \in\ dep\text{-}live\text{-}vars\ n'\ (a \cdot as')}$$

$$\frac{V \in Use\ (sourcenode\ a) \qquad sourcenode\ a\ -a \cdot as' \rightarrow_{cd} n'' \qquad n'' -as'' \rightarrow_d * n'}{(V, [])\ \in\ dep\text{-}live\text{-}vars\ n'\ (a \cdot (as'\ @\ as''))}$$

$$\frac{(V, as') \in dep\text{-}live\text{-}vars\ n'\ as \qquad V' \in Use\ (sourcenode\ a)}{n' = last\ (targetnodes\ (a \cdot as)) \qquad sourcenode\ a\ -\{V\}a \cdot as' \rightarrow_{dd} last(targetnodes\ (a \cdot as'))}{(V', [])\ \in\ dep\text{-}live\text{-}vars\ n'\ (a \cdot as)}$$

$$\frac{(V, as') \in dep\text{-}live\text{-}vars\ n'\ as}{n' = last\ (targetnodes\ (a \cdot as)) \qquad \neg\ sourcenode\ a\ -\{V\}a \cdot as' \rightarrow_{dd} last(targetnodes\ (a \cdot as'))}{(V, a \cdot as')\ \in\ dep\text{-}live\text{-}vars\ n'\ (a \cdot as)}$$

An easy corollary of this definition is that for all $(V, as') \in$ *dep-live-vars n′ as*, list as' is a prefix of list *as*. For an example of a calculation of the dependent live variables, see Fig. 3. Note that there is no PDG path from **n5** to **n9** (as there is no dependency edge leaving **n9**), so we can ignore the variables used at this node.

w:=8   b:=w-4   (c)   v:=u+5   y:=v   x:=v-w   (b>3)   y:=7

n1 → n2 → n3 → n4 → n5 → n6 → n7 → n8 → n9
   1   2   3   4   5   6   7   8

w   b   v   x   y

Calculation of *dep-live-vars* for node n9 (if x and y are in its *Use* set):

$as = \ldots$	elements in set *dep-live-vars* n9 *as*
[]	$\{(x,[]),(y,[])\}$
[8]	$\{(x,[8])\}$
[7,8]	$\{(x,[7,8]),(b,[])\}$
[6,7,8]	$\{(b,[6]),(v,[]),(w,[])\}$
[5,6,7,8]	$\{(b,[5,6]),(v,[5]),(w,[5])\}$
[4,5,6,7,8]	$\{(b,[4,5,6]),(u,[]),(w,[4,5])\}$
[3,4,5,6,7,8]	$\{(b,[3,4,5,6]),(u,[3]),(w,[3,4,5]),(c,[])\}$
[2,3,4,5,6,7,8]	$\{(w,[]),(u,[2,3]),(w,[2,3,4,5]),(c,[2])\}$
[1,2,3,4,5,6,7,8]	$\{(u,[1,2,3]),(c,[1,2])\}$

Fig. 3. Example for the calculation of Dependent Live Variables for node **n9**

Lemma 3. *(Dependent Live Variables and Use sets)*
If $(V, as') \in$ *dep-live-vars* n' as and $n -as \rightarrow * n'$ then
$V \in Use\ n' \land (\forall n'' \in set\ (sourcenodes\ as).\ V \notin Def\ n'') \land as = as'$ or
$(\exists nx\ as''.\ as = as'\ @\ as'' \land n -as' \rightarrow * nx \land nx -as'' \rightarrow_d * n' \land V \in Use\ nx \land$
$(\forall n'' \in set\ (sourcenodes\ as').\ V \notin Def\ n''))$.

This lemma, which is proved by induction on the dependent live variables rules, now leads us directly to an important statement coded in Corollary 1: Suppose we have a variable V with edge list component as' in the *dep-live-vars* set of node n' and path as and a TCFG path from the target node of a valid edge a leading to n' also using edges as. If now V gets defined at the source node of edge a, there is a PDG path from *sourcenode* a via edges as to n' with a leading data dependency edge for variable V:

Corollary 1. *(Dependent Live Variables and PDG paths)*
If $(V, as') \in$ *dep-live-vars* n' as and *targetnode* $a -as \rightarrow * n'$ and
$V \in Def$ (*sourcenode* a) and *valid-edge* a then
$(\exists nx\ as''.\ as = as' @ as'' \land$ *sourcenode* $a -\{V\}a \cdot as' \rightarrow_{dd} nx \land nx -as'' \rightarrow_d * n')$

Machine Checked Dynamic Slicing: Now, before proving the desired fundamental property of path slicing, we show a more general lemma. We have a TCFG path $n -as \rightarrow * n'$, n' not being (*-Exit-*), and two bit vectors $bs \preceq_b bs'$, the first one the result of *slice-path* as. es and es' are their respective edge kind lists obtained using *select-edge-kinds* on edges as. Furthermore we have two states s and s' agreeing on all variables in the *dep-live-vars* set for the slicing node n' on edge list as. If now all predicates hold while traversing edge kinds es' with starting state s', so do all predicates while traversing edge kinds es with starting state s and the value of any variable V in the *Use* set of slicing node n' agrees in the states yielded by both traversals.

Lemma 4. *(Generalized Fundamental Property of Dynamic Path Slicing)*

$$\frac{\begin{array}{c} n -as \longrightarrow * \; n' \qquad bs \preceq_b bs' \\ \textit{slice-path as} = bs \qquad \textit{select-edge-kinds as bs} = es \qquad \textit{select-edge-kinds as bs}' = es' \\ \forall \; V \; xs. \; (V, xs) \in \textit{dep-live-vars } n' \; as \longrightarrow \textit{state-val s } V = \textit{state-val s}' \; V \qquad \textit{preds es}' \; s' \end{array}}{(\forall \; V \in \textit{Use } n'. \; \textit{state-val } (\textit{transfers es s}) \; V = \textit{state-val } (\textit{transfers es}' \; s') \; V) \land \textit{preds es s}}$$

We prove this lemma by induction on *bs*, where the base case (*bs* = [] and thus *bs'* = []) is trivial. In the induction step we know that since *bs* is nonempty *as* consists of a (valid) leading edge *a'* and the tail list *as'*. The proof then does the following case analysis: if traversing edge list *as* changes one of the values in the *Use* set of *n'* compared to traversing just the tail edge list *as'*, the source node *n* of the leading edge *a'* has to define a variable in the dependent live variables set of *n'* reached via *as'* – otherwise no such influence would be possible. This implies by Corollary 1 that there is a PDG path from *n* to *n'*, so edge *a'* must be part of the slice, i.e. the first element of *bs* must be *True* and so is the first element of *bs'* by definition of $\preceq_b$. If however traversing edge list *as* gives the same values in the *Use* set of *n'* as traversing just the tail edge list *as'*, the traversal of the leading edge *a'* does not matter for the variables used in slicing node *n'*. This proposition, combined with the induction hypothesis and the well-formedness properties of the TCFG, leads to the proof of this lemma.

Replacing *bs'* with the maximal bit vector w.r.t. $\preceq_b$ of the matching size, using the definition of *slice-kinds* (see Def. 6) and instantiating *s* and *s'* with the same state *s*, the fundamental property is now an easy consequence:

Theorem 1. *(Fundamental Property of Dynamic Path Slicing)*

$$\frac{n -as \longrightarrow * \; n' \qquad \textit{preds } (\textit{kinds as}) \; s}{\begin{array}{c} (\forall \; V \in \textit{Use } n'. \; \textit{state-val } (\textit{transfers } (\textit{slice-kinds as}) \; s) \; V = \textit{state-val } (\textit{transfers } (\textit{kinds as}) \; s) \; V) \\ \land \; \textit{preds } (\textit{slice-kinds as}) \; s \end{array}}$$

Provided that the TCFG is also semantically well-formed, we easily extend this theorem to the semantics: a statement *c* evaluated in state *s* returns residual statement *c'* and state *s'*. Then there exists a path between the corresponding nodes of the statements and after traversing the sliced version of this path, all variables that are used in the target node have the same value as in state *s'*, the result of the semantics evaluation.

Theorem 2. *(Dynamic Path Slicing and Semantics)*

$$\frac{n \; \textit{identifies } c \qquad \langle c,s \rangle \Rightarrow \langle c',s' \rangle}{\begin{array}{c} \exists \; n' \; as. \; n -as \longrightarrow * \; n' \land \textit{preds } (\textit{slice-kinds as}) \; s \land n' \; \textit{identifies } c' \land \\ (\forall \; V \in \textit{Use } n'. \; \textit{state-val } (\textit{transfers } (\textit{slice-kinds as}) \; s) \; V = \textit{state-val s}' \; V) \end{array}}$$

3 Instantiation of the Framework with a Simple While-Language

We demonstrate that the framework is applicable and the well-formedness conditions are sensible by showing how to embed a simple imperative While-language (without procedures) called WHILE. In this section we use the basic definitions shown in Sec. 2.1, instantiating the type variables accordingly.

The Language: Our language features two value types *Intg*::*val* and *Bool*::*val*, which represent integer and boolean (i.e. *true* and *false*) values. Expressions consist of constant values, variables and binary operators. We support five different statements of type *cmd*: the no-op statement *Skip*, assignment of expression e to a variable V, written $V:=e$, sequential composition of statements ;;, conditionals *if* (b) c_1 *else* c_2 and while loops *while* (b) c'. Defining the state is easy: it is just a simple mapping from variables to values *var* $\rightharpoonup$ *val*. The partial function $[\![e]\!]s$ returns *Some v*, if expression e evaluates to value v in state s, *None*, if e cannot be evaluated in state s (e.g. in the case of non well-formed programs).

The Control Flow Graph: As WHILE does not provide procedures, its trace CFG is equal to its CFG. Nodes in this CFG are of the type *w-node*, which incorporates inner nodes (*- l -*) bearing a label l of type *nat*, and the special nodes (*-Entry-*) and (*-Exit-*). $n \oplus i$ adds i to the label number of n and returns a new node bearing this number as label, if n is an inner node, otherwise leaving (*-Entry-*) and (*-Exit-*) unchanged. #:c denotes the number of inner nodes we need for a CFG of statement c.

WHILE CFG edges have the type *w-edge* $=$ *w-node* $\times$ *state edge-kind* $\times$ *w-node*. A CFG edge valid for program *prog* is written *prog* $\vdash n -et\rightarrow n'$ and consists of a description of the program *prog* of type *cmd*, i.e. the statement for which this CFG is generated, a source node n, an edge kind et of type *state edge-kind* and a target node n'. Basically, the CFG is constructed via first constructing recursively the CFGs for the substatements, then combining these graphs into a single one, eventually adjusting the labels so that they remain unique. Also, we add one additional node after every variable assignment node and one after every *while* node (reachable via the edge, where the loop predicate is false). We will describe the motivation for this later.

Definition 8. *(WHILE CFG Edges)*
Basic rules:

$$prog \vdash (\text{-Entry-}) -(\lambda s.\ False)_{\sqrt{}} \rightarrow (\text{-Exit-}) \qquad prog \vdash (\text{-Entry-}) -(\lambda s.\ True)_{\sqrt{}} \rightarrow (\text{- 0 -})$$

Skip: $Skip \vdash (\text{- 0 -}) -\Uparrow id \rightarrow (\text{-Exit-})$

V:=e:

$$V:=e \vdash (\text{- 0 -}) -\Uparrow \lambda s.\ s(V := [\![e]\!]s) \rightarrow (\text{- 1 -}) \qquad V:=e \vdash (\text{- 1 -}) -\Uparrow id \rightarrow (\text{-Exit-})$$

c_1 ;;c_2 : $\dfrac{c_1 \vdash n -et\rightarrow n' \qquad n' \neq (\text{-Exit-})}{c_1 \ ;;\ c_2 \vdash n -et\rightarrow n'} \qquad \dfrac{c_1 \vdash n -et\rightarrow (\text{-Exit-}) \qquad n \neq (\text{-Entry-})}{c_1 \ ;;\ c_2 \vdash n -et\rightarrow (\text{- 0 -}) \oplus \#:c_1}$

$$\dfrac{c_2 \vdash n -et\rightarrow n' \qquad n \neq (\text{-Entry-})}{c_1 \ ;;\ c_2 \vdash n \oplus \#:c_1 \ -et\rightarrow n' \oplus \#:c_1}$$

if (b) c_1 **else** c_2 : *if* (b) c_1 *else* $c_2 \vdash (\text{- 0 -}) -(\lambda s.\ [\![b]\!]s = Some\ true)_{\sqrt{}} \rightarrow (\text{- 0 -}) \oplus 1$

$$\text{if } (b)\ c_1 \text{ else } c_2 \vdash (\text{- 0 -}) -(\lambda s.\ [\![b]\!]s = Some\ false)_{\sqrt{}} \rightarrow (\text{- 0 -}) \oplus (\#:c_1\ + 1)$$

$$\dfrac{c_1 \vdash n -et\rightarrow n' \qquad n \neq (\text{-Entry-})}{\text{if } (b)\ c_1 \text{ else } c_2 \vdash n \oplus 1 -et\rightarrow n' \oplus 1}$$

$$\dfrac{c_2 \vdash n -et\rightarrow n' \qquad n \neq (\text{-Entry-})}{\text{if } (b)\ c_1 \text{ else } c_2 \vdash n \oplus (\#:c_1\ + 1) -et\rightarrow n' \oplus (\#:c_1\ + 1)}$$
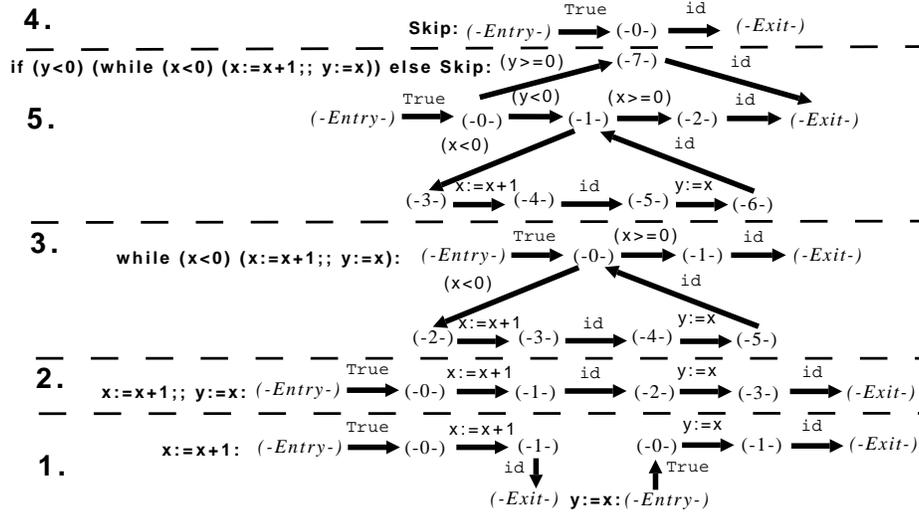
Fig. 4. Construction of the CFG for *if* $(y<0)$ (*while* $(x<0)$ $(x:=x+1;;y:=x)$) *else Skip*

while (**b**) **c'**: *while* (b) $c' \vdash (\text{- }0\text{ -}) -(\lambda s.\ [\![b]\!]s = Some\ true)_{\surd} \to (\text{- }0\text{ -}) \oplus 2$

    while (b) $c' \vdash (\text{- }0\text{ -}) -(\lambda s.\ [\![b]\!]s = Some\ false)_{\surd} \to (\text{- }1\text{ -})$

    while (b) $c' \vdash (\text{- }1\text{ -}) -\Uparrow id \to (\text{-}Exit\text{-})$ $\dfrac{c' \vdash n -et \to (\text{-}Exit\text{-}) \qquad n \neq (\text{-}Entry\text{-})}{while\ (b)\ c' \vdash n \oplus 2 -et \to (\text{- }0\text{ -})}$

$\dfrac{c' \vdash n -et \to n' \qquad n \neq (\text{-}Entry\text{-}) \qquad n' \neq (\text{-}Exit\text{-})}{while\ (b)\ c' \vdash n \oplus 2 -et \to n' \oplus 2}$

In Fig. 4, we show schematically the single steps 1.-5. of the recursive CFG construction for the statement *if* $(y<0)$ (*while* $(x<0)$ $(x:=x+1;;y:=x)$) *else Skip*. Now we have to prove that these definitions fulfill the well-formedness criteria given in Sec. 2.1. Hence we define *sourcenode*, *targetnode* and *kind* accordingly, say that a is a valid edge iff for a program *prog* $prog \vdash sourcenode\ a -kind\ a \to targetnode\ a$ holds and that n is a valid node iff it is the source or target node of a valid edge. Also we define the *Use* set as the set of all variables in the expression on the right hand side of a variable assignment and those occuring in a condition or loop predicate. The *Def* set contains only the variable that eventually gets assigned. Using these, the basic and well-formedness properties of the WHILE CFG are easily shown via rule induction on the CFG edge rules.

The Semantics: The rules for a small step or structural operational semantics then look as expected – only the *while*-rule has been split in two cases instead of a translation into a conditional –, $\langle c,s \rangle \to \langle c',s' \rangle$ stating that reduction of statement c in state s results in remainder statement c' and state s':

Definition 9. *(Small Step Semantics of* WHILE*)*

$\langle V:=e,s \rangle \to \langle Skip,s(V := [\![e]\!]s) \rangle$ $\dfrac{\langle c,s \rangle \to \langle c',s' \rangle}{\langle c;;\ c_2,s \rangle \to \langle c';;\ c_2,s' \rangle}$ $\langle Skip;;\ c_2,s \rangle \to \langle c_2,s \rangle$

$$\frac{[\![b]\!]s = \textit{Some true}}{\langle \textit{if }(b)\ c_1\ \textit{else } c_2\ ,s\rangle \rightarrow \langle c_1\ ,s\rangle} \qquad \frac{[\![b]\!]s = \textit{Some false}}{\langle \textit{if }(b)\ c_1\ \textit{else } c_2\ ,s\rangle \rightarrow \langle c_2\ ,s\rangle}$$

$$\frac{[\![b]\!]s = \textit{Some true}}{\langle \textit{while }(b)\ c,s\rangle \rightarrow \langle c;;\ \textit{while }(b)\ c,s\rangle} \qquad \frac{[\![b]\!]s = \textit{Some false}}{\langle \textit{while }(b)\ c,s\rangle \rightarrow \langle \textit{Skip},s\rangle}$$

To prove that the WHILE CFG in combination with this semantics is semantically well-formed, we need some kind of simulation of reducing statements in the semantics via following the CFG edges. Thus, we need a one-to-one mapping from the current position in the CFG to the current state of reduction in the semantics, i.e. from node n to statement c, which we called n *identifies* c in Sec. 2.1. Predicate *labels prog n c* accomplishes this task in program *prog*, defined by recursively computing the predicate for substatements and adjusting the labels if necessary (taking care of the additional nodes for $V{:}{=}e$ and *while*, being in fact *Skip* nodes).

Definition 10. *(Mapping from Nodes to Statements)*

$$\textit{labels } c\ 0\ c \qquad\qquad \textit{labels }(V{:}{=}e)\ 1\ \textit{Skip}$$

$$\frac{\textit{labels } c_1\ l\, c}{\textit{labels }(c_1 ;;\ c_2\,)\ l\ (c;;\ c_2\,)} \qquad\qquad \frac{\textit{labels } c_2\ l\, c}{\textit{labels }(c_1 ;;\ c_2\,)\ (l + \#{:}c_1\,)\ c}$$

$$\frac{\textit{labels } c_1\ l\, c}{\textit{labels }(\textit{if }(b)\ c_1\ \textit{else } c_2\,)\ (l+1)\ c} \qquad\qquad \frac{\textit{labels } c_2\ l\, c}{\textit{labels }(\textit{if }(b)\ c_1\ \textit{else } c_2\,)\ (l + \#{:}c_1\ +1)\ c}$$

$$\frac{\textit{labels } c'\ l\, c}{\textit{labels }(\textit{while }(b)\ c')\ (l+2)\ (c;;\ \textit{while }(b)\ c')} \qquad\qquad \textit{labels }(\textit{while }(b)\ c')\ 1\ \textit{Skip}$$

Trying to define that traversing the CFG edges conforms to reducing the semantics will fail as the semantics is not strong enough, since it reduces (and thus destroys) the initial statement and does not contain any information on node labels. So we need another means for reducing statements, called *label semantics*, which contains both statement and label information and also remembers the initial statement. A step in this semantics is written $prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle$, meaning that in program *prog* (the initial statement), statement c in state s with label l reduces to a remainder statement c' in state s' with label l'. The label semantics rules can be divided into two groups: first, we have seven rules corresponding to the seven rules of the standard semantics in Def. 9, with identical rules for $V{:}{=}e$, *if* $(b)\ c_1\ $*else* c_2 and *while* $(b)\ c'$ (only label information added). The rules for reducing a sequential composition need more premises to guarantee that the reduction provides the correct label for the right hand side of the composition:

$$\frac{\textit{labels }(c_1 ;;\ c_2\,)\ l\ (\textit{Skip};;\ c_2\,) \qquad \textit{labels }(c_1 ;;\ c_2\,)\ \#{:}c_1\ c_2 \qquad l < \#{:}c_1}{c_1 ;;\ c_2 \vdash \langle \textit{Skip};;\ c_2\,,s,l\rangle \rightsquigarrow \langle c_2\,,s,\#{:}c_1\,\rangle}$$

$$\frac{\textit{labels }(\textit{while }(b)\ c')\ l\ (\textit{Skip};;\ \textit{while }(b)\ c')}{\textit{while }(b)\ c' \vdash \langle \textit{Skip};;\ \textit{while }(b)\ c',s,l\rangle \rightsquigarrow \langle \textit{while }(b)\ c',s,0\rangle}$$

Second, we have five more rules to take care of valid semantics steps in substatements also being valid in composite statements (i.e. ;;, *if* $(b)\ c_1\ $*else* c_2 and *while* $(b)\ c'$). We show two examples of these rules:

$$\frac{prog \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle}{c_1 ;;\ prog \vdash \langle c,s,l + \#{:}c_1\,\rangle \rightsquigarrow \langle c',s',l' + \#{:}c_1\,\rangle}$$

$$\frac{cx;;\ \textit{while }(b)\ cx \vdash \langle c,s,l\rangle \rightsquigarrow \langle c',s',l'\rangle \qquad l < \#{:}cx \qquad l' < \#{:}cx}{\textit{while }(b)\ cx \vdash \langle c,s,l+2\rangle \rightsquigarrow \langle c',s',l'+2\rangle}$$

Using *labels* we now show that a step in the label semantics simulates a step in the small step semantics (proven via induction on *c*):

Lemma 5. *(Label Semantics Simulates Semantics)*

$$\frac{labels\ prog\ l\ c \qquad \langle c,s \rangle \rightarrow \langle c',s' \rangle}{\exists l'.\ prog \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle \wedge labels\ prog\ l'\ c'}$$

The label semantics now provides all information we need to show that traversing a CFG edge simulates a reducing step in it:

Lemma 6. *(CFG Edge Simulates Label Semantics)*

$$\frac{prog \vdash \langle c,s,l \rangle \rightsquigarrow \langle c',s',l' \rangle}{\exists et.\ prog \vdash (\text{-}l\text{-}) -et\rightarrow (\text{-}l'\text{-}) \wedge transfer\ et\ s = s' \wedge pred\ et\ s}$$

When proving this proposition by rule induction on the label semantics rules, the need for the additional *Skip* nodes after variable assignments and while loops gets clear: as the reduction of a variable assignment or while loop with invalid predicate leads to a *Skip* statement, we also need a CFG edge from the respective nodes to a node corresponding to a *Skip* statement. As the outgoing edge of this node is by construction of the CFG a no-op edge (function *id* is applied to the state, see the second rule for $V{:}{=}e$ and the third rule for *while* (b) c' in Def. 8), this adjustment is valid.

Showing that the WHILE CFG is semantically well-formed w.r.t. its semantics is just an easy consequence of Lem. 5 and 6, lifted to the corresponding transitive closures.

4 Related Work

Our slicing formalization approach is related to the work by Agrawal and Horgan [1], primarily to Approach 3, since nodes can occur multiple times in path, which is our equivalent to their *execution history*. From such an execution history, they build the *Dynamic Dependence Graph*, but this graph does not correspond to the PDG computed here as the latter contains all paths, not only a selected one. Although data dependence can be computed for a single path in isolation, we need this additional information about all possible paths for computing the control dependence relation. Agrawal and Horgan also use (in their case static) PDG information to determine the control dependences in their execution history. Having all possible traces in a TCFG is only a formalization trick that is of course not applicable in algorithms really computing dynamic slices because TCFGs are potentially infinite .

The quite natural idea of replacing parts irrelevant for slicing with operations doing nothing can also be found in the work by Amtoft [2]. He uses a *code map* for mapping CFG nodes to the corresponding program statements. Slicing then modifies this code map for all nodes not in the slice, mapping statement nodes to **skip** (equivalent to our $\Uparrow id$), predicate nodes to $true\textbf{?}$ or $false\textbf{?}$. Encoding the predicate directly in the edges, we can restrict ourselves to $(\lambda s.\ True)_\checkmark$ for these edges in the dynamic case.

Jhala and Majumdar [6] also use the notion of *path slicing*, but they focus on eliminating all edges not relevant for the (un)reachability of the target location. The resulting

(path) slices have two main properties: First, if the slice is infeasible, so is the original path, and second, if the slice is feasible, then the target location is reachable. Similar to our approach, they do not focus on one special path or trace, but also take alternative paths into consideration, e.g. to find feasible variants of the path under consideration.

Ranganath et al. [13] discuss interesting questions that arise if some nodes in the CFG cannot reach the Exit node. Our formalization is not immune to this problem as unstructured control flow can lead to this undesired situation. At the moment we can only guarantee that paths are sliced correctly for graphs without such nodes, otherwise certain control dependences may not be recognized, i.e. slices may be too small. However the main theorems remain correct, but the slices do not necessarily conform the standard understanding of correct slices. Since we also want to apply our framework to unstructured control flow, future efforts will be made to tackle this problem.

There exist some formal approaches to program slicing and its correctness. Reps and Yang showed in [14] the correctness of static PDG based slicing, restricted to a simple While language without procedures. The approach of Gouranton and Le Métayer [5] is similar to ours as they present a language independent framework and use it to show the correctness of dynamic slicing. Instead of using graph structures, they base their work on natural semantics. The slicing itself uses annotations where program points with annotation *False* are treated as *Skip*, the same strategy we pursue with our notion of bit vectors. They also present the embedding of three different languages in their framework: an imperative, a logic programming and a functional one. In [16], Ward and Zedan model slicing as a program transformation, thereby abstracting from specific representations. A program transformation in their sense is any operation on a program which generates a semantically equivalent program. The aim of their work is to provide a unified mathematical framework for sequential programs. We think that our approach is closer to the intuitive understanding of PDG based slicing than the last two methods as we use the standard notions of control flow, and control and data dependence. Also, both works rely on pen-and-paper proofs whereas our framework is fully machine-checked.

The notion of control flow graphs can be found in various works on verification using theorem provers, e.g. see [10] and [4], using control flow implicitly, i.e. as a relation, not as a real graph structure. Lammich and Müller-Olm [9] define a parallel flow graph similar in structure to our control flow graph (but formalising procedures and parallelism), which is not restricted to a certain language either. While our work uses flow graphs to construct dependence graphs and to prove certain properties of them, they focus on the correctness of analyses on the flow level.

5 Conclusion and Future Work

We have presented a generic framework for dynamic slicing using a PDG based approach and proven it correct in the proof assistant Isabelle/HOL. This framework is independent from concrete programming language syntax but uses a graph structure fulfilling certain structural and well-formedness properties. Moreover we have presented how to embed a simple imperative While language (without procedures) in this framework to show that the preconditions imposed on the graph structure are sensible. The formalization, including the instantiation with a While language has about 6000 lines of code and took one man-year of work.

Our next goal is to instantiate this framework with more sophisticated object oriented languages like Jinja [7] or CoreC++ [17] and to apply it to a formalization of the JVM [7] with unstructured control flow. Also we are working on expanding the framework to the intricacies of static slicing including, amongst other things, a formalization of system dependence graphs (SDGs) and an approximation of the peculiarities of dynamic binding. In the remote future we plan to extend the formalization to concurrency.

References

1. Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proc. of PLDI '90*, pages 246–256. ACM Press, 1990.
2. Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processig Letters*, 106(2):45–51, 2008.
3. Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi et al., editor, *Proc. of TYPES'03*, volume 3085 of *LNCS*, pages 34–50. Springer-Verlag, 2004.
4. Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *Proc. of SEFM'05*, pages 200–209. IEEE, 2005.
5. Valerie Gouranton and Daniel Le Métayer. Dynamic slicing: a generic analysis based on a natural semantics format. *Journal of Logic and Computation*, 9(6):835–871, 1999.
6. Ranjit Jhala and Rupak Majumdar. Path slicing. In *Proc. of PLDI'05*, pages 38–47. ACM Press, 2005.
7. Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
8. Jens Krinke. Program slicing. *Handbook of Software Engineering and Knowledge Engineering*, 3:307–332, 2004.
9. Peter Lammich and Markus Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation and procedures. In L. Caires and V.T. Vasconcelos, editors, *Proc. of CONCUR*, volume 4703 of *LNCS*, pages 287–302. Springer-Verlag, 2007.
10. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. of POPL'06*, pages 42–54. ACM Press, 2006.
11. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
12. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. 2002.
13. Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM TOPLAS*, 29(5):27, 2007.
14. Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report CS-TR-1988-777, University of Wisconsin-Madison, 1988.
15. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
16. Martin Ward and Hussein Zedan. Slicing as a program transformation. *ACM TOPLAS*, 29(2):1–53, 2007.
17. Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *Proc. of OOPSLA'06*, pages 345–362. ACM Press, 2006.
18. Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, 1979.
19. Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.