**ForwardInterproceduralAnalysis.cpp --Class ForwardInterProceduralAnalysis<M,N,A>**

A generic forward-flow inter-procedural analysis which is fully context-sensitive.

This class essentially captures a forward data flow problem which can be solved using the context-sensitive inter-procedural analysis framework as described in [InterProceduralAnalysis](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html).

This is the class that client analyses will extend in order to perform forward-flow inter-procedural analysis.

|  |
| --- |
| **Fields inherited from class** [**InterProceduralAnalysis**](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html) |
| [contexts](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html#contexts), [contextTransitions](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html#contextTransitions), [reverse](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html#reverse), [worklist](http://padhye.org/vasco/apidocs/vasco/InterProceduralAnalysis.html#worklist) |

|  |  |
| --- | --- |
| protected  [Context](http://padhye.org/vasco/apidocs/vasco/Context.html)<[M](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html),[N](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html),[A](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html)> | [**initContext**](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html#initContext(M,%20A))([M](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html) method, [A](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html) entryValue)           Creates a new value context and initialises data flow values for its nodes. |

The following steps are performed:

1. Construct the context.
2. Initialise IN/OUT for all nodes and add them to the work-list
3. Initialise the IN of entry points with a copy of the given entry value.
4. Add this new context to the given method's mapping.
5. Add this context to the global work-list.

**Parameters:**

method - the method whose context to create

entryValue - the data flow value at the entry of this method

Context<M, N, A> ForwardInterProceduralAnalysis<M, N, A>::initContext(M method, A entry\_value) {

Context<M, N, A> \*context = new Context<M, N, A>(method, false);

std::vector<N> cfg = context->getControlFlowGraph();

for(int i=0; i<cfg.size(); i++) {

context->setValueBefore(cfg[i], topValue());

context->setValueAfter(cfg[i], topValue());}

context->addToWorklist(cfg[cfg.size() - 1]);

context->setEntryValue(copy(entry\_value));

context->setValueBefore(context->getEntryNode(), copy(entry\_value));

context->setExitValue(topValue());

contexts[method].push\_back(std::ref(\*context));

workList.push\_back(std::ref(\*context));

DBG(llvm::errs() << "initContext: " << context->getId() << " Fname: " << method->getName() << "\n";)

return \*context;

}

|  |  |
| --- | --- |
| void | [**doAnalysis**](http://padhye.org/vasco/apidocs/vasco/ForwardInterProceduralAnalysis.html#doAnalysis())()           Performs the actual data flow analysis. |

A work-list of contexts is maintained, each with it's own work-list of CFG nodes to process. For each node removed from the work-list of the newest context, the meet of values along incoming edges (in the direction of analysis) is computed and then the flow function is processed depending on whether the node contains a call or not. If the resulting data flow value has changed, then nodes along outgoing edges (in the direction of analysis) are also added to the work-list.

Analysis starts with the context for the program entry points with the given boundary values and ends when the work-list is empty.

See the SOAP '13 paper for the full algorithm in Figure 1.

void ForwardInterProceduralAnalysis<M, N, A>::doAnalysis(void) {

M e\_method = getEntryMethod();

initContext(e\_method, boundaryValue(e\_method));

while(!workList.empty()) {

// TODO: Will probably need to make this a pointer

std::reference\_wrapper<Context<M, N, A>> current\_context = workList.back();

if(current\_context.get().isEmptyWorklist()) {

current\_context.get().markAnalysed();

workList.pop\_back();

continue;

}

N node = current\_context.get().getAndPopWorklist();

if(node != NULL){

llvm::errs() << "---------------------doAnalysis Node being processed: " << current\_context.get().getId() << " " << node->getParent()->getName() << ":" << node->getName() << "\n";

} else {

llvm::errs() << "---------------------doAnalysis Node being processed: " << current\_context.get().getId() << " " << current\_context.get().getMethod()->getName() << ":NULL" << "\n";

}

for(N &e: current\_context.get().getWorklist()) {

if(e == NULL) {

DBG(llvm::errs() << current\_context.get().getMethod()->getName() << ":NULL" << "\n";)

} else {

DBG(llvm::errs() << e->getParent()->getName() << ":" << e->getName() << "\n";)

}

} if(node != NULL) {

std::vector<N> predecessors = current\_context.get().getPredsOf(node);

if(predecessors.size() != 0) {

A in = topValue();

for(int i=0; i<predecessors.size(); i++) {

A predOut = current\_context.get().getValueAfter(predecessors[i]);

in = meet(in, predOut);

}

current\_context.get().setValueBefore(node, in);

}

A prevOut = current\_context.get().getValueAfter(node);

A in = current\_context.get().getValueBefore(node);

A out;

if(current\_context.get().isCall(node)) {

out = callCustomFlowFunction(current\_context, node, in);

} else {

out = normalFlowFunction(current\_context, node, in);

}

out = meet(out, prevOut);

DBG(llvm::errs() << "setting afterValue value:" << "\n";)

for(auto &x : out) {

DBG(llvm::errs() << "\t " << x.first << " " << SIGN\_toString(x.second) << "\n";)

}

current\_context.get().setValueAfter(node, out);

if(isEqual(out, prevOut) == false) {

std::vector<N> successors = current\_context.get().getSuccsOf(node);

for(int i=0; i<successors.size(); i++) {

DBG(llvm::errs() << "successors:" << successors[i]->getName() << "\n";)

current\_context.get().addToWorklist(successors[i]);

}

}

std::vector<N> tails = current\_context.get().getTails();

for(N &e: tails) {

DBG(llvm::errs() << "Tails:" << e->getParent()->getName() << ":" << e->getName() << "\n";)

}

if(current\_context.get().getLastWorklist() != NULL){

DBG(llvm::errs() << "getLastWorklist " << current\_context.get().getLastWorklist()->getName() << "\n";)

} else {

DBG(llvm::errs() << "getLastWorklist NULL" << "\n";)

}

DBG(llvm::errs() << "Node " << node->getName() << "\n";)

if(current\_context.get().isEmptyWorklist()) {

if(std::find(tails.begin(), tails.end(), node) != tails.end()) {

current\_context.get().addToWorklist(NULL);

}

}

} else {

for(N &e: current\_context.get().getWorklist()) {

if(e == NULL) {

DBG(llvm::errs() << current\_context.get().getMethod()->getName() << ":NULL" << "\n";)

} else {

DBG(llvm::errs() << e->getParent()->getName() << ":" << e->getName() << "\n";)

}

}

assert(current\_context.get().isEmptyWorklist());

A exit\_value = topValue();

std::vector<N> tails = current\_context.get().getTails();

for(int i=0; i<tails.size(); i++) {

DBG(llvm::errs() << "setting exit value:" << tails[i]->getName() << "\n";)

A tail\_out = current\_context.get().getValueAfter(tails[i]);

for(auto &x : tail\_out) {

DBG(llvm::errs() << "\t " << x.first << " " << SIGN\_toString(x.second) << "\n";)

}

exit\_value = meet(exit\_value, tail\_out);

}

A prev\_exit\_value = current\_context.get().getExitValue();

DBG(llvm::errs() << "get exit value:" << "\n";)

for(auto &x : current\_context.get().getExitValue()) {

DBG(llvm::errs() << "\t " << x.first << " " << SIGN\_toString(x.second) << "\n";)

}

DBG(llvm::errs() << "set exit value:"<< "\n";)

for(auto &x : exit\_value) {

DBG(llvm::errs() << "\t " << x.first << " " << SIGN\_toString(x.second) << "\n";)

}

current\_context.get().setExitValue(exit\_value);

current\_context.get().markAnalysed();

std::vector<CallSite<M, N, A>> callers = context\_transitions.getCallers(current\_context.get());

if(!callers.empty() and !isReturnEqual(prev\_exit\_value, exit\_value)) {

for(auto &call\_site: callers) {

std::reference\_wrapper<Context<M, N, A>> calling\_context = getContextbyId(call\_site.second->getParent(), call\_site.first.getId());

N call\_node = call\_site.second;

DBG(llvm::errs() << "call\_site: call node " << call\_site.first.getId() << ":" << call\_node->getParent()->getName() << ":" << call\_node->getName() << "\n";)

if(calling\_context.get().getLastWorklist() == NULL)

calling\_context.get().addToWorklist(call\_node);

workList.push\_back(calling\_context);

}

}

}