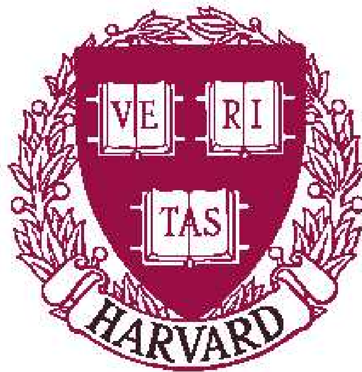
# Varia: optimization by logic programming

Kevin Redwine
and
Kelly Heffner

TR-13-05




Computer Science Group
Harvard University
Cambridge, Massachusetts

# Varia: optimization by logic programming
# Harvard University TR-13-05

Kevin Redwine and Kelly Heffner

June 10, 2005

## Abstract

We have designed a prototype compiler optimization infrastructure called Varia and demonstrated its potential to explore the space of optimizations. In Varia everything is represented as logic: instructions and blocks are axioms, analyses and transformations are inference rules, and optimization proceeds by forward-chaining deduction. Varia makes adding and combining optimizations simple—every rule may be eligible to fire at each step, so optimizations are automatically combined. We have coded five optimizations in our system and demonstrate that Varia can combine them. We describe how to translate normal iterative data-flow analyses into Varia, briefly discuss the performance of the rules engine, and finally, propose changes in the engine to make it more suitable for optimization.

Things we need to edit:

- Massage an example so we can fully demonstrate how hypothetical facts allow the combination of two analyses/optimizations. Ie. How the hypothetical fact allows us to trigger an optimization, which then discharges the hypothesis, thus reaching a fixed point. I would really be happy if we could find some strong relation to logical axioms for this. I need to read all my modal logic books.

## 1 Introduction

Compiler optimizations are intended to improve code. However, generating good code is difficult for a given source program. Not every optimization will improve the code, an optimization may impair or enable later optimizations, and mutually enabling optimizations may be combined. Finally, the idea of "good code" is difficult to define precisely and can be interpreted in many ways: fast execution, small code size, low power requirements, etc.

The compiler or compiler writer must choose which optimizations to apply, the order to apply them in, and which to combine. Generating good code requires making these sophisticated decisions on a program-by-program basis as well as understanding the target execution environment. It may also be important to consider the amount of time an optimization will take and weigh that against the expected improvement of code, especially for JIT compilation.

We are designing a compiler infrastructure called Varia with two aims: to provide an experimental infrastructure for testing phase ordering and optimization combination decisions, and to provide a flexible infrastructure for use in compilers.

Using inference rules and forward-chaining, Varia combines data-flow analyses and transformations by running them simultaneously. One of our design goals is to make it easy to drop-in new optimizations or to refine existing optimizations; our immediate goal, however, has been simply to show that our design is plausible. To this end, we built a prototype and implemented constant propagation, constant folding, branch elimination, and reachability—in the paper we also walk through a liveness example. As proof-of-concept we combine these analyses and transformations and demonstrate we can achieve the benefits cited in the literature.

Although the system may prove suitable to experiment with the order that optimizations are applied, the phase ordering problem is not currently the focus of this work. At present, all transformations are assumed to be beneficial and are therefore applied immediately.

## 2 Related work

This is not a new problem: *super-analyses*, or a data-flow analysis that combines several smaller analyses at once have been hand-written in many compilers to reap the benefits from mutual interaction [15, 12, 14, 6, 11, 5, 3]. The engineering effort required to construct and test a super-analysis is dissatisfying, while the variety of these super-analyses only suggests that there exists more interactions that have not been noticed with the naked eye.

The Rhodium project is one attempt to ameliorate the process of composing optimizations[4, 8]. Rhodium composes phases that operate in the same direction (forwards or backwards) into a larger phase.

## 3 Execution model

Our method of constructing and running compiler optimizations is non-standard. Most compilers run a sequence of iterative analyses and transformations. In Varia, however, several analyses and transformations can run simultaneously. There are many possible ways to represent optimizations and to run them simultaneously. We feel that a particularly simple and modular way to represent an optimization is as a set of inference rules. Each optimization in our system is represented by several inference rules that work together to compute data-flow facts and possibly to produce a different CFG. Composing optimizations with this representation is just a matter of loading more rules. Varia is implemented using Jess [7], a rule engine written in Java.

Our rules execute by *forward chaining*. We start with an empty database of facts. If each precondition of a rule is met by a fact in the database, that rule may fire. When a rule fires there are several possible actions it can take: it may add a new fact to the database, it may remove a fact (referenced by the preconditions) from the database, or it may modify a fact (referenced by the preconditions) in the database. In rare cases a rule may perform more than one action. Optimizations are run simultaneously simply by running all of the rules together. If several rules are eligible to fire, we non-deterministically choose one. Execution continues until no rule can fire.

Intuitively, interactions between analyses and transformations that fire before the entire graph is analyzed are sound because at no point should an analysis assert a fact about the program that is not true. This is different from traditional iterative analysis that only guarantees sound information once the analysis reaches fix-point. It is important to note that when a fact, $f$, is changed or retracted from the system, all new facts derived from $f$ are retracted and the rule set must re-derive them if they are in fact still true.

An important design question is "how do we know whether a set of rules will terminate on a given CFG?" Ideally, we would like to be able to prove that our rules will terminate when run on any CFG. That has yet to be proven formally, but we offer (in Section 6) an informal argument that the particular rules we have implemented will always terminate. It is, of course, possible to write down a set of rules that won't terminate—we have done this ourselves in the course of developing Varia.

We feel confident that it will be possible to reformulate the rule language such that users are, at the very least, able to tell whether they are doing something dangerous with regard to termination—it is our hope, however, that we can actually rule out non-terminating optimizations. In either case this will remain a very important design point.

## 4 Speculation

Recall that in standard data-flow, analyses that use the meet operator intersection normally initialize each fact to the universe, $U$. This is done so that as the analysis iterates, invalid facts are "intersected out" (since they are missing on at least one edge at a join point). In Varia, all facts that are asserted are true, therefore it is impossible to initialize facts to a universal value and iterate to a fixed point. We solve this problem by introducing *speculative facts*. A speculative fact is a normal fact in our system, except that it is dependent on other semantic facts in the program. In other words, it is a fact that is only true if the system can deduce its dependencies. Varia uses speculative facts in order to compute simple data-flow fact loop invariants in the presence of back-edges.We go into more detail about this in section 5.4.

In order to produce a speculative fact, a rule needs to check a condition on *all* of a certain kind of data. For example, we might like to know that the variable $x$ has constant value 5 on *every* incoming edge. There are two issues here. One problem with this is our database of facts evolves over time, so if we model the CFG and instructions as facts and derive edge information from them, then at some point in time it's

possible that not all edge information has been computed. We solve this problem by keeping information like outgoing edges that, conceptually, we want to loop over all together in lists. When we loop over all elements in a list of outgoing edges from a block then we know we have checked each relevant edge, for example. Another problem is that when we speculate something about the incoming edges to a block, like $x$ is constant 5, eventually we need a way to figure out that it really was true—that is, $x$ really was 5 on each incoming edge. That may only be determined after pushing the speculation all the way through the CFG and along some back edge in a loop. The critical point is when we have determined that "if x was 5 on this edge into block b then x was constant 5 into block b". We want to be able to reduce speculation of that sort into just the bare fact "x is constant 5 into block b". We will explain more about how this is accomplished in Section 5.

# 5 Adding analyses

Analyses are added to the system via new rules and a new fact type, if necessary. Depending on the structure of the analysis, the rules will be a subset of:

- Sets of syntactic and possibly semantic conditions that are necessary to generate a new fact at the program point before or after an instruction.
- Sets of syntactic and semantic conditions necessary to propagate a fact from the program point before an instruction to the program point after an instruction (or vice versa).
- Definition of the meet operator necessary for propagating facts on incoming edges to the program point before the first instruction in the block.
- Lattice information necessary for propagating a fact from the program point after the last instruction in a block to each edge out of the block. We will go into more detail on this later.
- Lattice information necessary for properly reducing speculative analysis. Again, we will go into more detail on this throughout the section.

While a normal iterative analysis needs to provide all of the rules described above, it is possible to implement an analysis with only a subset. This would be appropriate when the analysis is simply very specialized additional rules. An example of this is a special rule for conditional branches, that asserts simply that the condition tested is true on the true edge in the control flow graph.

In order to illustrate these rules, we will walk through an implementation of live variable analysis. Recall that classical live variable analysis is a backwards analysis and uses union for the meet operator. A variable, $x$, is live at a program point, $p$, if there is a path from $p$ to the sink of the CFG in which $x$ is accessed.

## 5.1 Fact Design

The first step of implementing the analysis is to decide what facts we want to generate. The obvious answer for liveness is we want something like $\langle x$ is live at point $p\rangle$. However, how is liveness information used? We do dead variable elimination using the fact $\langle x$ is not live at point $p\rangle$. Since it is unsound to do a transformation of the absence of a fact (*not* $\langle x$ is live at point $p\rangle$), we actually need to generate facts describing the points where $x$ is dead [13]. [1] So, our facts will have the skeleton $\langle$*variable* is dead *location*$\rangle$.

## 5.2 Fact Generation Rule

As mentioned above, we need to provide rules for generating facts from syntactic conditions. This is very similar to the process of determining the *gen sets* in iterative analysis. For our dead variable analysis example, the condition we need is if a variable is defined in an instruction, $i$, and not used in $i$, that variable is dead at the program point before $i$. Informally, this rule looks like:

$$\frac{\begin{array}{c}\text{assignment instruction } i\\ i \text{ contains destination } x\\ i \text{ does not contain source } x\end{array}}{\langle x \text{ is dead before } i\rangle}$$

## 5.3 Fact Propagation Rule

The fact generation rules create semantic facts about the program points where the information is generated, but provide no facts about other points in the

[1] In order to assure flexibility of our analysis for future needs (such as register allocation), we could actually generate facts about where we know $x$ is dead and where we know $x$ is not dead. But, to keep the example short, we just propagate information about deadness.

program. An analysis must also provide rules for propagating the fact to other program points if necessary. For deadness, we are concerned with *all* points where a variable is dead, so we need to specify the conditions in which deadness before one instruction implies deadness before another instruction. Simply, if a variable, $x$, is dead after an instruction, $i$, and $i$ does not read from $x$, then $x$ is dead before $i$.[2] This is one rule that manifests that liveness is a backwards analysis. Informally, this rule looks like:

$$\frac{\langle x \text{ is dead after } i\rangle \qquad i \text{ does not contain source } x}{\langle x \text{ is dead before } i\rangle}$$

## 5.4 Meeting Facts at Join Points

In section 4, we discussed the need for making speculative assumptions in order to try to deduce a loop invariant data-flow fact. In fact, we use speculation to deduce data-flow facts that are higher (more precise) on the lattice than could be deduced normally. Speculations can only be added at join points and reduce on edges. Speculation directly corresponds to the intersection meet operator; since intersection requires a fact to be on all edges at a join point in order for that fact to be propagated through the join.

To illustrate more clearly, let us consider our deadness analysis example. In traditional liveness, the live variables at the bottom of a block are determined by taking the union of the live variables on each of the edges out of the block. Using DeMorgan's Law, we can see that traditional iterative deadness would use intersection as the meet operator. More directly, a variable is dead at the bottom of a block only if it is dead on all of the edges out of the block. To phrase this speculatively, we can assert that if we know a variable, $x$, is dead on one edge out of a block, $b$, then if we knew that $x$ was dead on the other edges out of $b$, we could deduce that $x$ is dead at the bottom of $b$. Informally the rule looks like:

$$\frac{\langle x \text{ is dead on edge } b_1 \to b_2\rangle}{\begin{array}{c}\forall e \in (\{b_1 \to *\} \setminus \{b_1 \to b_2\}).\langle x \text{ is dead on edge } e\rangle \vdash_\ell \\ \langle x \text{ is dead at bottom of } b_1\rangle\end{array}}$$

Note that this introduces speculative assumptions that the generated fact is dependent on. Every rule that uses the fact needs to add these assumptions to the resulting deductions. We rewrite our informal rules to reflect this in figure 1, but omit them for simplicity here.

[2]We note here that the engine should be smart enough to determine that the program point before $i$ is the same as the program point after $i$'s predecessor.

## 5.5 Asserting Facts on Edges

In order to speculate, there need to be facts about program points which are edges. For our deadness analysis, we deduce that a variable, $x$, is dead on an edge into a block, $b$, if $x$ is dead before the first instruction in $b$. Informally, the rule looks like:

$$\frac{\langle x \text{ is dead at the top of } b\rangle}{\forall e \in \{* \to b\}.\langle x \text{ is dead on edge } e\rangle}$$

Again, the backward nature of liveness is manifest in the rule.

Figure 1 contains the full version of the rule that propagates speculative assumptions. The simplicity of the lattice for deadness makes this rule very simple; there is no way to create a strange fact like

$$\langle x \text{ is live on edge } e\rangle \vdash_\ell \langle x \text{ is dead on edge } e\rangle$$

. If our analysis contained facts about liveness and deadness, we would still want to prevent useless facts like this from being asserted. However, if we consider a more complex lattice, such as the one for constant propagation, the similar ill-formed fact

$$\langle x = 3 \text{ on edge } e\rangle \vdash_\ell \langle x = 2 \text{ on edge } e\rangle$$

is not only useless, it poses a dangerous problem. Consider the simple control flow graph fragment:

The naive rule for constant propagation to an edge would look something like:

$$\frac{A \vdash_\ell \langle x = c \text{ at the bottom of } b\rangle}{\forall e \in \{b \to *\}.A \vdash_\ell \langle x = c \text{ on edge } e\rangle}$$

which simply states that if $x$ is equal to the constant $c$ at the bottom of $b$, propagate $x = c$ onto the edges out of $b$, preserving the speculative facts that allowed us to deduce $x = c$ at the bottom of $b$. However, this rule allows us to deduce the facts:

$$\{\langle x = 1 \text{ on } b_2 \to b_2\rangle\} \vdash_\ell \langle x = 2 \text{ on } b_2 \to b_2\rangle$$

$$\begin{array}{l}\{\langle x = 1 \text{ on } b_2 \to b_2\rangle, \\ \quad \langle x = 2 \text{ on } b_2 \to b_2\rangle\} \vdash_\ell \langle x = 3 \text{ on } b_2 \to b_2\rangle\end{array}$$

$$\begin{array}{l}\{\langle x = 1 \text{ on } b_2 \to b_2\rangle, \\ \quad \langle x = 2 \text{ on } b_2 \to b_2\rangle, \\ \quad \langle x = 3 \text{ on } b_2 \to b_2\rangle\} \vdash_\ell \langle x = 4 \text{ on } b_2 \to b_2\rangle\end{array}$$

$$\text{GEN}\ \frac{A \vdash_\ell \text{assignment instruction } i \qquad A \vdash_\ell i \text{ contains destination } x \qquad A \vdash_\ell \text{ does not contain source } x}{A \vdash_\ell \langle x \text{ is dead before } i \rangle}$$

$$\text{PROP-INSTR}\ \frac{A_1 \vdash_\ell \langle x \text{ is dead after } i \rangle \qquad A_2 \vdash_\ell i \text{ does not contain source } x}{A_1 \cup A_2 \vdash_\ell \langle x \text{ is dead before } i \rangle}$$

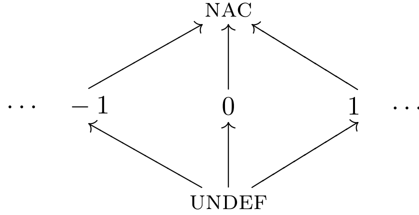
$$\text{MEET}\ \frac{A \vdash_\ell \langle x \text{ is dead on edge } b_1 \to b_2 \rangle}{A \cup (\forall e \in (\{b_1 \to *\} \setminus \{b_1 \to b_2\}).\langle x \text{ is dead on edge } e \rangle) \vdash_\ell \langle x \text{ is dead at bottom of } b_1 \rangle}$$

$$\text{EDGE}\ \frac{A \vdash_\ell \langle x \text{ is dead at the top of } b \rangle}{\forall e \in \{* \to b\}. A \vdash_\ell \langle x \text{ is dead on edge } e \rangle}$$

$$\text{ELIM-DEAD}\ \frac{A_1 \vdash_\ell f_1 \qquad f_2 = \langle x \text{ is dead on edge } e \rangle \in A \qquad A_2 \subseteq A_1 \vdash_\ell e \text{ is not reachable}}{(A_1 \cup A_2) - f_2 \vdash_\ell f_1}$$

Figure 1: The inference rules for our deadness analysis. Note that in generate rule, each antecedent has the same assumption set; this is just reflecting that in the actual implementation, the facts about instruction $i$ are about the same instruction based on the same assumptions.

and continue with this *analysis by simulation* without reaching a fixed point. Just as in traditional data-flow, we solve this problem by defining the semi-lattice[3] of possible values for an integer variable:




and changing the rule so that it performs the appropriate operation, *join* ($\vee$), between the newly deduced fact and any speculative assumption it depends on. Informally, we split the rule into two cases for clarity:

$$\frac{\begin{array}{c}A \vdash_\ell \langle x = c_1 \text{ at the bottom of } b \rangle \\ e \in \{b \to *\} \qquad \langle x = c_2 \text{ on edge } e \rangle \in A\end{array}}{A \vdash_\ell \langle x = (c_1 \vee c_2) \text{ on edge } e \rangle}$$

$$\frac{\begin{array}{c}A \vdash_\ell \langle x = c_1 \text{ at the bottom of } b \rangle \\ e \in \{b \to *\} \qquad \langle x = c_2 \text{ on edge } e \rangle \notin A\end{array}}{A \vdash_\ell \langle x = c_1 \text{ on edge } e \rangle}$$

[3]Yes, this lattice is the reverse of the new Dragon book [?] presentation. We respect this lattice in the use of the terms *meet* and *join* for lattice values; please reverse your terminology if this lattice is upside down to you.

Appropriately, this rule now yields fact:

$$\{\langle x = 1 \text{ on } b_2 \to b_2 \rangle\} \vdash_\ell \langle x = NAC \text{ on } b_2 \to b_2 \rangle$$

but, this is still not very useful, since we have just created a fact that will often never reduce. As the final correction to our rule, we observe that by joining the speculative fact with the deduced fact (on the same edge), we have computed the fixed point for the loop, given the other speculations in set $A$. Therefore, we can safely remove the speculation $\langle x = 1 \text{ on edge } b_2 \to b_2 \rangle$ since its been incorporated into the fixed point. We change the rules to reflect this fix-point reduction:

$$\frac{\begin{array}{c}A \vdash_\ell \langle x = c_1 \text{ at the bottom of } b \rangle \\ e \in \{b \to *\} \qquad f = \langle x = c_2 \text{ on edge } e \rangle \in A\end{array}}{A - f \vdash_\ell \langle x = (c_1 \vee c_2) \text{ on edge } e \rangle}$$

$$\frac{\begin{array}{c}A \vdash_\ell \langle x = c_1 \text{ at the bottom of } b \rangle \\ e \in \{b \to *\} \qquad \langle x = c_2 \text{ on edge } e \rangle \notin A\end{array}}{A \vdash_\ell \langle x = c_1 \text{ on edge } e \rangle}$$

The interesting result of this lattice theoretic interpretation of the rule is that the circular elimination rule discussed in section 4 is subsumed since $c \vee c = c$, and the speculation $x = c$ on edge $e$ will be dropped[4].

[4]This is an excellent result, since the original presentation of this elimination from our talk is difficult to accept intuitively

### 5.6 Speculation Elimination

We have seen how lattice information allows us to elegantly express one of the logical deductions, fixpoint elimination. There are a few other situations in which the system needs information to eliminate a speculation. Consider the graph fragment:

We draw the edge from block 2 to block 3 in red to denote that it will be determined unreachable (a dead edge). However, since it is possible that our deadness analysis evaluated this join point before the edge is determined unreachable, we need to allow our system to reduce speculations about dead edges in the appropriate way. For our analysis, we can intuitively say that if an edge is dead, any variable on that edge can be considered dead as well. We informally write the rule:

$$\frac{A \vdash_\ell f_1 \qquad f_2 = \langle x \text{ is dead on edge } e \rangle \in A \quad e \text{ is not reachable}}{A - f_2 \vdash_\ell f_1}$$

and in figure 1 we see that we also need to include any speculations that were needed to deduce that $e$ is not reachable.

Varia as it is currently implemented requires a few other elimination rules, such as transitivity. These rules are not inherent to a specific data-flow analysis and will be discussed as deductions the logic system should handle in section 8.

## 6 Termination

Given the execution model described in section 3, we give an intuitive argument that properly defined rules (as described in section 5) create a system that terminates.

Speculations are only added to the system at join points; speculations are limited by the number of join points and edges per join point.

We say that two facts are *comparable* if they are of the same fact type, describe the same program information (for example, the same variable in deadness analysis), and label the same program point. A fact in the system can depend on a large set of speculative facts, however no two facts in this set can be comparable. Therefore there is a hard upper bound on the number of speculations possible for a fact of

$$\begin{aligned}\mathcal{W}[\![\, speculations \,]\!] = &|\text{edges in any join point}| \times \\ &|\text{types of facts}| \times \\ &|\text{information items per fact type}|\end{aligned}$$

Since all facts in the system are true, given an attempt to insert two comparable facts with the same speculations, we always keeps the more informative of the two facts[5]; this allows opaque sharing in the system without unnecessarily propagating information that is already deemed over-conservative. More directly, there are never two comparable facts with the same speculations in the system. Therefore there is worst-case upper bound on the number of facts in the system of:

$$\begin{aligned}\mathcal{W}[\![\, facts \,]\!] = \mathcal{W}[\![\, speculations \,]\!] \times &|\text{types of facts}| \times \\ &|\text{information items per fact type}| \times \\ &|\text{program points}|\end{aligned}$$

Overall, this gives a worst case cap on the number of facts:

$$\begin{aligned}\mathcal{W}[\![\, facts \,]\!] = &|\text{edges in any join point}| \times \\ &|\text{types of facts}|^2 \times |\text{program points}| \times \\ &|\text{information items per fact type}|^2 \times\end{aligned}$$

which is dissatisfying for performance, but we have implemented some optimizations to our system that try to keep the size of speculations small and therefore the number of facts small. We will go into more detail on this in section 7.1.

As described in our introduction, Jess [7] allows any rule to assert, retract, or modify facts. Our rules never retract or modify facts directly. Optimizations modify program information and the infrastructure retracts the relevant facts; rules assert new facts using a special abstraction that handles discerning comparable facts (and dropping overly-conservative facts as mentioned above). With the exception of optimizations, this means that the set of facts is monotonically increasing. Also, as discussed in section 5.5, at any point where we derive a fix point, we monotonically join the lattice values. As a point of future work, we hope to explore the conditions under which two optimizations could make "opposite" changes to the program such that the system thrashes going back and forth between the two desired transformations.

[5]It should never be the case that the system derives two facts of the same type about one program point where one lattice value $x$ is not on the same chain as $y$. This is a corollary from the claim that every fact in the system is true.

| Test | Instr | Time (ms) | Rules fired | Facts added | removed | modified | Final facts |
|---|---|---|---|---|---|---|---|
| dep1 | 6 | 202 | 300 | 677 | 498 | 12 | 179 |
| dep2 | 6 | 437 | 606 | 1346 | 986 | 15 | 360 |
| dep3 | 6 | 723 | 578 | 1221 | 931 | 28 | 290 |
| irr | 14 | 1899 | 1980 | 3302 | 2654 | 95 | 648 |
| cc | 11 | 18228 | 6911 | 12587 | 10002 | 65 | 2585 |

Table 1: Test suite run in Jess

# 7 Performance

Performance has not been a large concern for us up to this point. Our effort has been focussed on developing ideas and getting Varia to compute what we want it to compute. We do, however, have a small test suite of small CFGs. All of the CFGs we tested on have six basic blocks except the final example which had eight blocks. A block, for us, may include a single exit point at the end of the block.

We ran each test on a 550Mhz Pentium 3 with 256 Mb of RAM. All of the rules in our system are encoded in the Jess language and are executed using the Jess interpreter—an expert system written in Java. Using the Jess API, we wrote a Java program that collects statistics as our rules run on a CFG. The relevant statistics include the time in milliseconds it takes to complete a run, which does not include the startup time for Java or the time Jess takes to read in our rules, the total number of rules fired when running a test, the total number of facts added to the fact database, the number of facts removed from the fact database, the number of facts modified, and finally, the total number of facts left in the database when the program terminates.

Table 7 shows the results of running our tests. Test "dep1" is a CFG with a branch and a loop where we hope that the constant propagation rules will be able to figure out that the variable "x" is always 4—this requires the branch eliminator to run as well. Test "dep1" is shown in figure 2. Test "dep3" represents exactly the same CFG, however we have shuffled the order that blocks are declared in to get Jess to evaluate in a different order—as you can see, the order that rules end up firing in can have a drastic effect on runtime. Test "dep2" is similar to the CFG in "dep1", but the branch has been switched so that the variable "x" is no longer constant. This forces Varia to evaluate without finding any facts it can reduce—when it doesn't find what it is looking for, Varia usually takes longer. Test "irr" represents an irreducible CFG, and is shown in figure 3. We thought this was an interesting example because it is larger than other CFGs we have and because it is a case that we believe Rhodium can't deal with. Varia does find all the data-flow facts we expected in "irr". Finally, the test "cc" is our encoding of an example from Click and Cooper's paper[6]. Since we haven't yet coded global value numbering, Varia is incapable of finding all the unreachable code and constant propagation opportunities that we would hope. This still makes an interesting test case because it forces Varia to run to completion without being able to find what it's looking for—again this causes a longer runtime.

One trend that we can observe (although tentatively due to the small size of the data set) is that the runtime for a test case seems to grow linearly with the number of rules fired. This is what we expected and hoped to see—it just tells us that Jess isn't doing a bunch of work behind that scenes. We still have much more to understand about the performance of Varia. In particular, it would be nice to know what the most important factor is in a CFG—is it the number of nodes, the number of instructions, the number of join points, whether our system will be able to find the sort of facts it's looking for, etc? These are questions we hope to be able to answer in the future. It will be extremely helpful to integrate Varia with a real compiler so that we can get larger test cases.

## 7.1 Optimizations of the system

During our implementation of Varia, we observed a direct correlation between the number of facts the system deduces and execution time. We attempted some optimizations to trim the number of facts deduced while still producing the correct results. Obviously, we looked for an optimization that would reduce the number of facts with speculative assumptions in the system. We found one considerable opti-

Figure 2: The control flow graph, before and after optimization, for example “dep1”.

Figure 3: The control flow graph, before and after optimization, for example “irr”.

mization. Consider the two facts:

$$\{\langle x = 3 \text{ on edge } e_1\rangle, \langle x = 7 \text{ on edge } e_2\rangle\} \vdash_\ell \langle x = 3 \text{ on edge } e_3\rangle$$

and

$$\langle x = 3 \text{ on edge } e_1\rangle \vdash_\ell \langle x = 3 \text{ on edge } e_3\rangle$$

Since every fact in the database is sound given the speculations, the latter rule implies the former. If we already have the latter fact in the database when Varia asserts the first fact, our meta-logic simply omits adding it to the database.

The dual to this optimization, however, does not seem to be sound. We attempted to add a rule to the meta-logic such that if the first fact was in the database and we assert the second, then Varia retracts the first and asserts the second in its place. We are looking into whether this is a manifestation of Jess implementation or something inherent to the system.

### 7.2 Salience

Another way to improve the performance of the system by modifying the salience, or precedence, of the inference rules. For our current implementation of Varia, we do not modify the salience (so that we guarantee that our implementation is independent of precedence), but in the future we hope to experiment with this feature further. Obvious places to raise the salience of inference rules would be anywhere that produces facts without speculation, speculation elimination rules, and transformations. This would make Varia slightly more focused on finding facts without speculation and only use speculation to find a fixpoint when necessary.

## 8 Speculation Logic

Our logic system encodes speculative facts on top of the logic system of Jess. As we briefly discussed in section 6, this requires a few additional rules to always properly reduce speculation sets.

The first reduction is fixed point elimination, the idea that the fact

$$\langle x = 3 \text{ on edge } e\rangle \vdash_\ell \langle x = 3 \text{ on edge } e\rangle$$

given the speculation mechanism described in sections 5 and 4 reduces to:

$$\vdash_\ell \langle x = 3 \text{ on edge } e\rangle.$$

However, as we noted in section 5.5, using lattice based inference rules, this elimination is subsumed.

Other reductions that are presently encoded as separate rules is transitivity. If we have the two facts

$$\langle x = 3 \text{ on edge } e_1\rangle, \langle x = 7 \text{ on edge } e_2\rangle \vdash_\ell \langle x = 3 \text{ on edge } e_3\rangle$$

and

$$\langle x = 3 \text{ on edge } e_1\rangle \vdash_\ell \langle x = 7 \text{ on edge } e_2\rangle$$

$\langle x = 3 \text{ on edge } e_1\rangle$ is sufficient to deduce the second speculative condition $\langle x = 7 \text{ on edge } e_2\rangle$, which in turn allows us to deduce that $\langle x = 3 \text{ on edge } e_3\rangle$. Ideally, if our meta-logic system (with speculation) was the execution logic, the system itself would handle these derivations.

## 9 Related Work

The Rhodium project [4, 8, 9, 10] is focused on automatically composing optimizations and automatically proving the soundness of optimizations. The goal of composing optimizations is to produce better code, which they demonstrated is possible. However, to prove soundness of composed transformations Rhodium is only able to combine analyses in a single direction. Rhodium includes a domain specific language of rules for data-flow facts and transformations. The optimizer runs by taking all of the rules

that describe phases and running them "in lock-step" on the CFG. Each time an analysis is invoked, it can produce either a data-flow fact (denoting propagation of data-flow analysis but no suggested optimization) or a new CFG subgraph replacement. If an analysis suggests changing the CFG, then it is rerun with the modified CFG. In order to prove soundness in the presence of loops, the CFG modifications are not committed until a fix-point is reached; if no fix-point is possible, then the changes are abandoned because they were not sufficiently conservative.

Varia is similar to the Rhodium project in several respects. Our optimizations are also coded in a logic programming style and the execution engine is designed for mutual interaction. Like the Rhodium framework, it is possible to take the trace of rules invoked to optimize a program and use it to prove the correctness of the transformation[6]. However, our inference system does not require the data-flow to be run in lock step, nor does it require reducible control flow.

## 10 Future Work

We have shown that our design is plausible for implementing simple data-flow analyses and allowing mutual interaction between different data-flow analyses and transformations. The next step of this work is to fully formalize the system and prove that analyses that follow appropriate rules are sound and terminate. During our exploration of Varia, we noticed a possible correlation with modal logic which we hope will help structure the formal system. After we formalize the system, there are several directions we hope to take this work:

1. Formalizing the types of analyses that produce mutual dependencies in the presence of back-edges - our intuition is that this is simply a combination of control flow analysis and control-dependent data-flow analysis.

2. Explore systems for deducing the correct conditions for applying a transformation vs postponing a transformation. In Varia, this would replace our current conditions (which are *any conditions*) with some derivation that the transformation is optimizing (produces better code according to some heuristic).

3. Explore the ability for Varia to discharge proofs of soundness for the transformations performed as well as proofs of optimization if the above direction pans out.

In order to properly formalize our system, and guarantee that the logic rules are executed correctly, we will be continuing our research by implementing our own rules engine. The next revision of the engine and the inference rules to do optimization will be targeted for Microsoft's Phoenix framework [2] and the C– compiler [1].

[6] The inference rules must be proven sound.

## 11 Acknowledgements

This work was funded by NSF grants CCR-0310877, CCF-0429782, ITR-0325460, and CCR-0311482 and a grant from the Microsoft Phoenix project [2].

## References

[1] The C– project. `http://cminusminus.org`.

[2] Microsoft phoenix framework. `http://research.microsoft.com/phoenix`.

[3] Benitez, M. E., and Davidson, J. W. A portable global optimizer and linker. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM Press, pp. 329–338.

[4] Chambers, C., Dean, J., and Grove, D. Frameworks for intra- and interprocedural dataflow analysis. Tech. Rep. 96-11-02, Department of Computer Science and Engineering, University of Washington, November 1996.

[5] Chambers, C., and Ungar, D. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation* (New York, NY, USA, 1989), ACM Press, pp. 146–160.

[6] Click, C., and Cooper, K. D. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst. 17*, 2 (1995), 181–196.

[7] Friedman-Hill, E. *Jess in Action, Java Rule-Based Systems*. Manning Publications Company, 2003.

[8] Lerner, S., Grove, D., and Chambers, C. Composing dataflow analyses and transformations. In *Symposium on Principles of Programming Languages* (2002), pp. 270–282.

[9] Lerner, S., Millstein, T., and Chambers, C. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM Press, pp. 220–231.

[10] Lerner, S., Millstein, T., Rice, E., and Chambers, C. Automated soundness proofs for dataflow analyses and transformations via local rules. *SIGPLAN Not. 40*, 1 (2005), 364–377.

[11] Pioli, A., and Hind, M. Combining interprocedural pointer analysis and conditional constant propagation, 1999.

[12] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *POPL* (1988), pp. 12–27.

[13] Schmidt, D. A. Data flow analysis is model checking of abstract interpretations. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1998), ACM Press, pp. 38–48.

[14] Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. In *POPL* (1985), pp. 291–299.

[15] Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13*, 2 (1991), 181–210.

# Appendix A

We have typeset all of the inferece rules of our system. The notation is similar but not entirely the same as that presented in the paper.

MAIN::INST-PRED-HEAD

$$\frac{\vdash_\ell \mathrm{block}(\mathrm{id} = ?, \mathrm{instrs} = (\mathrm{i}, \ldots), \mathrm{out\text{-}edges} = \ldots)}{\vdash \mathrm{main::ipred\text{-}head}(\mathrm{i})}$$

MAIN::INST-PRED

$$\frac{\vdash_\ell \mathrm{block}(\mathrm{id} = ?, \mathrm{instrs} = (\ldots, \mathrm{i}, \mathrm{j}, \ldots), \mathrm{out\text{-}edges} = \ldots)}{\vdash \mathrm{main::ipred}(\mathrm{i}, \mathrm{j})}$$

Figure 4: Rules for module MAIN

CFOLD::CONST-FOLD-ADD

$$\frac{\mathrm{id} \leftarrow \vdash \mathrm{instr@i} : \mathrm{d} := \mathrm{add}(\mathrm{a}\colon \mathrm{a} \in \mathbb{N}, \mathrm{b}\colon \mathrm{b} \in \mathbb{N})}{\mathrm{id} \leftarrow \mathrm{id}[\mathrm{op\text{-}name} = \mathrm{copy}, \mathrm{srcs} = \mathrm{a} + \mathrm{b}]}$$

CFOLD::CONST-FOLD-SUB

$$\frac{\mathrm{id} \leftarrow \vdash \mathrm{instr@i} : \mathrm{d} := \mathrm{sub}(\mathrm{a}\colon \mathrm{a} \in \mathbb{N}, \mathrm{b}\colon \mathrm{b} \in \mathbb{N})}{\mathrm{id} \leftarrow \mathrm{id}[\mathrm{op\text{-}name} = \mathrm{copy}, \mathrm{srcs} = \mathrm{a} - \mathrm{b}]}$$

Figure 5: Rules for module CFOLD

CP-PATH::CONST-I-1

$$\frac{\mathrm{NS} \vdash_\ell \text{instr@i} : \mathrm{d} := \text{copy}(\mathrm{s}\colon \mathrm{s} \in \mathbb{N})}{\text{cp-path::assert-is-const}(\mathrm{d}, \mathrm{s}, \mathrm{i}, \mathrm{NS})}$$

CP-PATH::CONST-I-2

$$\frac{\begin{array}{c}\mathrm{NS}_1 \vdash_\ell \text{cp-path::is-const}(\text{var} = \mathrm{v}, \text{val} = \mathrm{c}, \text{loc} = \mathrm{i}) \\ \vdash_\ell \text{ipred}(\mathrm{i}, \mathrm{j}) \qquad \mathrm{NS}_2 \vdash_\ell \text{instr@j} : (\mathrm{DS}\colon \mathrm{v} \notin \mathrm{DS}) := \mathrm{n}(\ldots)\end{array}}{\text{cp-path::assert-is-const}(\mathrm{v}, \mathrm{c}, \mathrm{j}, \mathrm{NS}_1 \cup \mathrm{NS}_2)}$$

CP-PATH::CONST-I-3

$$\frac{\begin{array}{c}\mathrm{NSC} \vdash_\ell \text{cp-path::const-in}(\text{var} = \mathrm{v}, \text{val} = \mathrm{c}, \text{id} = \mathrm{b}) \\ \vdash_\ell \text{block}(\text{id} = \mathrm{b}, \text{instrs} = (\mathrm{i}, \ldots)) \qquad \mathrm{NSI} \vdash_\ell \text{instr@i} : (\mathrm{DS}\colon \mathrm{v} \notin \mathrm{DS}) := ?(\ldots)\end{array}}{\text{cp-path::assert-is-const}(\mathrm{v}, \mathrm{c}, \mathrm{i}, \mathrm{NSC} \cup \mathrm{NSI})}$$

CP-PATH::GIVE-UP-SRCS-IPRED

$$\frac{\begin{array}{c}\vdash_\ell \text{instr@loc} : \ldots := ?(\ldots, \mathrm{v}\colon \mathrm{v} \notin \mathbb{N}, \ldots) \\ \vdash_\ell \text{ipred}(\text{pred}, \text{loc}) \qquad \nvdash_\ell \text{cp-path::is-const}(\text{loc} = \text{pred}, \text{var} = \mathrm{v}, \text{val} = (\mathrm{c}\colon \mathrm{c} \neq \top))\end{array}}{\text{cp-path::assert-is-const}(\mathrm{v}, \top, \text{pred}, \{\})}$$

CP-PATH::GIVE-UP-DSTS-IPRED

$$\frac{\begin{array}{c}\vdash_\ell \text{instr@loc} : (\ldots, \mathrm{v}\colon \mathrm{v} \notin \mathbb{N}, \ldots) := \text{op}(\ldots) \\ \vdash_\ell \text{ipred}(\text{pred}, \text{loc}) \qquad \nvdash_\ell \text{cti-op-p}(\text{op}) \qquad \nvdash_\ell \text{cp-path::is-const}(\text{loc} = \text{pred}, \text{var} = \mathrm{v}, \text{val} = (\mathrm{c}\colon \mathrm{c} \neq \top))\end{array}}{\text{cp-path::assert-is-const}(\mathrm{v}, \top, \text{pred}, \{\})}$$

CP-PATH::ELIM-FACT-IS-CONST

$$\frac{\begin{array}{c}(\mathrm{N}_1, \text{nb}, \mathrm{N}_2) \vdash_\ell \text{cp-path::is-const}(\text{var} = \mathrm{v}, \text{val} = \mathrm{c}, \text{loc} = \mathrm{i}) \\ \vdash_\ell \text{cp-path::want-const-edge}(\text{want-id} = \text{nb}, \text{source} = \text{src}, \text{sink} = \text{sink}, \text{var} = \mathrm{v}_2, \text{val} = \mathrm{c}_2) \\ \vdash_\ell \text{cp-path::const-edge}(\text{var} = \mathrm{v}_2, \text{val} = \mathrm{c}_2, \text{source} = \text{src}, \text{sink} = \text{sink})\end{array}}{\text{cp-path::assert-is-const}(\mathrm{v}, \mathrm{c}, \mathrm{i}, \mathrm{N}_1 \cup \mathrm{N}_2)}$$

Figure 6: Rules for is-const in CP-PATH module

CP-PATH::CONST-REPLACE-1

$$\frac{\vdash \text{cp-path::is-const}(\text{var} = \text{v}, \text{val} = (\text{c}\colon \text{c} \neq \top), \text{loc} = \text{i}) \qquad \vdash \text{ipred}(\text{i}, \text{j}) \qquad \text{id} \leftarrow \vdash \text{instr@j} : \text{D} := \text{n}(\text{H}, \text{v}, \text{T})}{\text{id} \leftarrow \text{id}[\text{srcs} = (\text{H}, \text{c}, \text{T})]}$$

CP-PATH::CONST-REPLACE-2

$$\frac{\begin{array}{c}\text{NSC} \vdash_{\ell} \text{cp-path::is-const}(\text{var} = \text{v}, \text{val} = (\text{c}\colon \text{c} \neq \top), \text{loc} = \text{i}) \\ \vdash_{\ell} \text{ipred}(\text{i}, \text{j}) \qquad (\text{NSI}\colon |\,\text{NSI}\,| + |\,\text{NSC}\,| > 0) \vdash_{\ell} \text{instr@j} : \text{D} := \text{n}(\text{H}, \text{v}, \text{T})\end{array}}{\text{NSC} \cup \text{NSI} \vdash \text{instr@j} : \text{d} := \text{n}(\text{H}, \text{c}, \text{T})}$$

CP-PATH::CONST-REPLACE-3

$$\frac{\begin{array}{c}\vdash \text{cp-path::const-in}(\text{var} = \text{v}, \text{val} = (\text{c}\colon \text{c} \neq \top), \text{id} = \text{b}) \\ \vdash \text{block}(\text{id} = \text{b}, \text{instrs} = (\text{i}, \ldots)) \qquad \text{id} \leftarrow \vdash \text{instr@i} : \text{D} := \text{n}(\text{H}, \text{v}, \text{T})\end{array}}{\text{id} \leftarrow \text{id}[\text{srcs} = (\text{H}, \text{c}, \text{T})]}$$

CP-PATH::CONST-REPLACE-4

$$\frac{\begin{array}{c}\text{NSC} \vdash_{\ell} \text{cp-path::const-in}(\text{var} = \text{v}, \text{val} = (\text{c}\colon \text{c} \neq \top), \text{id} = \text{b}) \\ \vdash_{\ell} \text{block}(\text{id} = \text{b}, \text{instrs} = (\text{i}, \ldots)) \qquad (\text{NSI}\colon |\,\text{NSI}\,| + |\,\text{NSC}\,| > 0) \vdash_{\ell} \text{instr@i} : \text{D} := \text{n}(\text{H}, \text{v}, \text{T})\end{array}}{\text{NSC} \cup \text{NSI} \vdash \text{instr@i} : \text{d} := \text{n}(\text{H}, \text{c}, \text{T})}$$

Figure 7: Rules for constant replacement in module CP-PATH

CP-PATH::CONST-IN-NAC

$$\frac{\mathrm{NSC} \vdash_{\ell} \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}, \mathrm{val} = \top, \mathrm{source} = \mathrm{b}, \mathrm{sink} = \mathrm{succ})}{\text{cp-path::assert-const-in}(\mathrm{v}, \top, \mathrm{succ}, \mathrm{NSC})}$$

CP-PATH::GIVE-UP-SRCS-HEAD

$$\frac{\begin{array}{c}\vdash_{\ell} \mathrm{instr@loc} : \ldots := ?(\ldots, \mathrm{v}\colon \mathrm{v} \notin \mathbb{N}, \ldots) \\ \vdash_{\ell} \mathrm{block}(\mathrm{id} = \mathrm{b}, \mathrm{instrs} = (\mathrm{loc}, \ldots)) \qquad \nvdash_{\ell} \text{cp-path::const-in}(\mathrm{id} = \mathrm{b}, \mathrm{var} = \mathrm{v}, \mathrm{val} = (\mathrm{c}\colon \mathrm{c} \neq \top))\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}, \top, \mathrm{b}, \{\})}$$

CP-PATH::GIVE-UP-DSTS-HEAD

$$\frac{\begin{array}{c}\vdash_{\ell} \mathrm{instr@loc} : (\ldots, \mathrm{v}\colon \mathrm{v} \notin \mathbb{N}, \ldots) := \mathrm{op}(\ldots) \qquad \vdash_{\ell} \mathrm{block}(\mathrm{id} = \mathrm{b}, \mathrm{instrs} = (\mathrm{loc}, \ldots)) \\ \nvdash_{\ell} \text{cti-op-p}(\mathrm{op}) \qquad \nvdash_{\ell} \text{cp-path::const-in}(\mathrm{id} = \mathrm{b}, \mathrm{var} = \mathrm{v}, \mathrm{val} = (\mathrm{c}\colon \mathrm{c} \neq \top))\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}, \top, \mathrm{b}, \{\})}$$

CP-PATH::MEET-NAC

$$\frac{\begin{array}{c}\mathrm{NEEDS} \vdash_{\ell} \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}_1, \mathrm{val} = \mathrm{c}_1, \mathrm{source} = \mathrm{src}_1, \mathrm{sink} = \mathrm{sink}) \\ \mathrm{NEEDS} \vdash_{\ell} \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}_1, \mathrm{val} = (\mathrm{c}_2\colon \mathrm{c}_1 \neq \mathrm{c}_2), \mathrm{source} = (\mathrm{src}_2\colon \mathrm{src}_1 \neq \mathrm{src}_2), \mathrm{sink} = \mathrm{sink})\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}_1, \top, \mathrm{sink}, \mathrm{NEEDS})}$$

CP-PATH::CONST-IN-1

$$\frac{\begin{array}{c}\mathrm{NSC} \vdash \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}, \mathrm{val} = (\mathrm{c}\colon \mathrm{c} \neq \top), \mathrm{source} = \mathrm{b}, \mathrm{sink} = \mathrm{succ}) \\ (\mathrm{NSC}_2\colon \mathrm{NSC}_2 \subseteq \mathrm{NSC}) \vdash \text{reachability::active-edge}(\mathrm{active} = \mathrm{t}, \mathrm{source} = \mathrm{b}, \mathrm{sink} = \mathrm{succ}) \\ \text{succ-preds} \leftarrow (\mathrm{sps} \leftarrow \{\}; \\ \forall \vdash \mathrm{block}(\mathrm{id} = (\mathrm{succpred}\colon \mathrm{succpred} \neq \mathrm{b}), \text{out-edges} = (\ldots, \mathrm{succ}, \ldots)).\ \mathrm{sps} \leftarrow \{\mathrm{succpred}\} \cup \mathrm{sps}; \mathrm{sps})\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}, \mathrm{c}, \mathrm{succ}, \mathrm{NSC} \cup (\text{cp-path::create-needs}, \text{succ-preds}, \mathrm{v}, \mathrm{c}, \mathrm{succ}))}$$

CP-PATH::ELIM-FACT-CONST-IN

$$\frac{\begin{array}{c}(\mathrm{N}_1, \mathrm{nb}, \mathrm{N}_2) \vdash_{\ell} \text{cp-path::const-in}(\mathrm{var} = \mathrm{v}, \mathrm{val} = \mathrm{c}, \mathrm{id} = \mathrm{b}) \\ \vdash_{\ell} \text{cp-path::want-const-edge}(\text{want-id} = \mathrm{nb}, \mathrm{source} = \mathrm{src}, \mathrm{sink} = \mathrm{sink}, \mathrm{var} = \mathrm{v}_2, \mathrm{val} = \mathrm{c}_2) \\ \vdash_{\ell} \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}_2, \mathrm{val} = \mathrm{c}_2, \mathrm{source} = \mathrm{src}, \mathrm{sink} = \mathrm{sink})\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}, \mathrm{c}, \mathrm{b}, \mathrm{N}_1 \cup \mathrm{N}_2)}$$

CP-PATH::ELIM-DEAD-CONST-IN

$$\frac{\begin{array}{c}(\mathrm{NC}_1, \mathrm{nb}, \mathrm{NC}_2) \vdash_{\ell} \text{cp-path::const-in}(\mathrm{var} = \mathrm{v}, \mathrm{val} = \mathrm{c}, \mathrm{id} = \mathrm{snk}) \\ \vdash_{\ell} \text{cp-path::want-const-edge}(\text{want-id} = \mathrm{nb}, \mathrm{source} = \mathrm{src}, \mathrm{sink} = \mathrm{snk}) \\ (\mathrm{N}_3\colon \mathrm{N}_3 \subseteq \mathrm{NC}_1 \cup \{\mathrm{nb}\} \cup \mathrm{NC}_2) \vdash_{\ell} \text{reachability::active-edge}(\mathrm{active} = \mathrm{f}, \mathrm{source} = \mathrm{src}, \mathrm{sink} = \mathrm{snk})\end{array}}{\text{cp-path::assert-const-in}(\mathrm{v}, \mathrm{c}, \mathrm{snk}, \mathrm{NC}_1 \cup \mathrm{NC}_2)}$$

CP-PATH::TRANSITIVE-3

$$\frac{\begin{array}{c}\mathrm{NAE} \vdash_{\ell} \text{cp-path::const-in}(\mathrm{var} = \mathrm{var}, \mathrm{val} = \mathrm{val}, \mathrm{id} = \mathrm{loc}) \\ \vdash_{\ell} \text{cp-path::want-const-edge}(\text{want-id} = (\mathrm{killme}\colon \mathrm{killme} \in \mathrm{NAE}), \mathrm{var} = \mathrm{v}, \mathrm{val} = \mathrm{c}, \mathrm{source} = \mathrm{src}, \mathrm{sink} = (\mathrm{snk}\colon \mathrm{loc} \neq \mathrm{snk})) \\ (\mathrm{NCE}\colon \mathrm{NCE} \subseteq \mathrm{NAE} \wedge \mathrm{killme} \notin \mathrm{NCE}) \vdash_{\ell} \text{cp-path::const-edge}(\mathrm{var} = \mathrm{v}, \mathrm{val} = \mathrm{c}, \mathrm{source} = \mathrm{src}, \mathrm{sink} = \mathrm{snk})\end{array}}{\text{cp-path::assert-const-in}(\mathrm{var}, \mathrm{val}, \mathrm{loc}, \mathrm{NAE} \setminus \{\mathrm{killme}\})}$$

Figure 8: Rules for const-in in module CP-PATH

CP-PATH::CONST-OUT-CONFLICT-2

$$\frac{\begin{array}{c}\mathrm{NSC} \vdash_\ell \text{cp-path::is-const}(\text{var} = \text{v}, \text{val} = (\text{c}_2\colon \text{c}_2 \neq \top), \text{loc} = \text{last}) \\ \vdash_\ell \text{block}(\text{id} = \text{b}, \text{instrs} = (\ldots, \text{last}), \text{out-edges} = (\ldots, \text{out}, \ldots)) \\ \vdash_\ell \text{instr@last} : \ldots := ?(\ldots) \qquad \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{t}, \text{source} = \text{b}, \text{sink} = \text{out}) \\ \vdash \text{cp-path::want-const-edge}(\text{want-id} = (\text{nid}_2\colon \text{nid}_2 \in \mathrm{NSC}), \text{var} = \text{v}, \text{val} = (\text{c}_1\colon \text{c}_1 \neq \text{c}_2), \text{source} = \text{b}, \text{sink} = \text{out}) \\ \begin{array}{l}\text{bad-needs} \leftarrow \\ \quad (\text{bns} \leftarrow \{\}; \\ \quad\ \forall \vdash \text{cp-path::want-const-edge}(\text{want-id} = (\text{nid}\colon \text{nid} \in \mathrm{NSC}), \text{var} = \text{v}, \text{val} = (\text{c}_1\colon \text{c}_1 \neq \text{c}_2), \text{source} = \text{b}, \text{sink} = \text{out}). \\ \qquad \text{bns} \leftarrow \text{bns} \cup \{\text{nid}\}; \\ \quad\ \text{bns})\end{array}\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \top, \text{b}, \text{out}, \mathrm{NSC} \setminus \text{bad-needs})}$$

CP-PATH::CONST-OUT-CLEAN

$$\frac{\begin{array}{c}\mathrm{NSC} \vdash_\ell \text{cp-path::is-const}(\text{var} = \text{v}, \text{val} = \text{c}_1, \text{loc} = \text{last}) \\ \vdash_\ell \text{block}(\text{id} = \text{b}, \text{instrs} = (\ldots, \text{last}), \text{out-edges} = (\ldots, \text{out}, \ldots)) \\ \vdash_\ell \text{instr@last} : \ldots := ?(\ldots) \qquad \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{t}, \text{source} = \text{b}, \text{sink} = \text{out}) \\ \nvdash \text{cp-path::want-const-edge}(\text{want-id} = (\text{nid}\colon \text{nid} \in \mathrm{NSC}), \text{var} = \text{v}, \text{val} = (\text{c}_2\colon \text{c}_2 \neq \text{c}_1), \text{source} = \text{b}, \text{sink} = \text{out})\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \text{c}_1, \text{b}, \text{out}, \mathrm{NSC})}$$

CP-PATH::ELIM-CIRCULAR-CONST-OUT

$$\frac{\begin{array}{c}(\mathrm{N}_1, \text{nb}, \mathrm{N}_2) \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src}, \text{sink} = \text{snk}) \\ \vdash_\ell \text{cp-path::want-const-edge}(\text{want-id} = \text{nb}, \text{source} = \text{src}, \text{sink} = \text{snk}, \text{var} = \text{v}, \text{val} = \text{c})\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \text{c}, \text{src}, \text{snk}, \mathrm{N}_1 \cup \mathrm{N}_2)}$$

CP-PATH::ELIM-FACT-CONST-OUT

$$\frac{\begin{array}{c}(\mathrm{N}_1, \text{nb}, \mathrm{N}_2) \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src}, \text{sink} = \text{sink}) \\ \vdash_\ell \text{cp-path::want-const-edge}(\text{want-id} = \text{nb}, \text{source} = \text{src}_2, \text{sink} = \text{sink}_2, \text{var} = \text{v}_2, \text{val} = \text{c}_2) \\ \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{v}_2, \text{val} = \text{c}_2, \text{source} = \text{src}_2, \text{sink} = \text{sink}_2)\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \text{c}, \text{src}, \text{sink}, \mathrm{N}_1 \cup \mathrm{N}_2)}$$

CP-PATH::TRANSITIVE-2

$$\frac{\begin{array}{c}\mathrm{NAE} \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{var}, \text{val} = \text{val}, \text{source} = \text{src}, \text{sink} = \text{snk}) \\ \vdash_\ell \text{cp-path::want-const-edge}(\text{want-id} = (\text{killme}\colon \text{killme} \in \mathrm{NAE}), \text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src-ce}, \text{sink} = \text{snk-ce}) \\ (\mathrm{NCE}\colon \mathrm{NCE} \subseteq \mathrm{NAE} \wedge \text{killme} \notin \mathrm{NCE}) \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src-ce}, \text{sink} = \text{snk-ce})\end{array}}{\text{cp-path::assert-const-edge}(\text{var}, \text{val}, \text{src}, \text{snk}, \mathrm{NAE} \setminus \{\text{killme}\})}$$

CP-PATH::BRANCHEQ

$$\frac{\begin{array}{c}\mathrm{NI} \vdash \text{instr@i} : (\text{b}_1, \text{b}_2) := \text{brancheq}(\text{v}\colon \text{v} \notin \mathbb{N}, \text{c}\colon \text{c} \in \mathbb{N}) \\ \vdash \text{block}(\text{id} = \text{blkid}, \text{instrs} = (\ldots, \text{i}), \text{out-edges} = \ldots)\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \text{c}, \text{blkid}, \text{b}_1, \mathrm{NI})}$$

CP-PATH::BRANCHNEQ

$$\frac{\begin{array}{c}\mathrm{NI} \vdash \text{instr@i} : (\text{b}_1, \text{b}_2) := \text{branchneq}(\text{v}\colon \text{v} \notin \mathbb{N}, \text{c}\colon \text{c} \in \mathbb{N}) \\ \vdash \text{block}(\text{id} = \text{blkid}, \text{instrs} = (\ldots, \text{i}), \text{out-edges} = \ldots)\end{array}}{\text{cp-path::assert-const-edge}(\text{v}, \text{c}, \text{blkid}, \text{b}_2, \mathrm{NI})}$$

Figure 9: Rules to compute constant edge facts in module CP-PATH

CP-PATH::TRANSITIVE-1

$$\frac{\begin{array}{c}\text{NAE} \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk}) \\ \vdash_\ell \text{cp-path::want-const-edge}(\text{want-id} = (\text{killme: killme} \in \text{NAE}), \text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src-ce}, \text{sink} = \text{snk-ce}) \\ (\text{NCE: NCE} \subseteq \text{NAE} \wedge \text{killme} \notin \text{NCE}) \vdash_\ell \text{cp-path::const-edge}(\text{var} = \text{v}, \text{val} = \text{c}, \text{source} = \text{src-ce}, \text{sink} = \text{snk-ce})\end{array}}{\text{reachability::assert-active-edge}(\text{a}, \text{src}, \text{snk}, \text{NAE} \setminus \{\text{killme}\})}$$

Figure 10: Rules for active-edge in module CP-PATH

REACHABILITY::MARK-OUT-EDGES

$$\frac{\text{NRB} \vdash_\ell \text{reachability::reachable}(\text{block} = \text{b}, \text{is-reachable} = \text{r}) \quad \vdash_\ell \text{block}(\text{id} = \text{b}, \text{out-edges} = (\ldots, \text{succ}, \ldots))}{\text{reachability::assert-active-edge}(\text{r}, \text{b}, \text{succ}, \text{NRB})}$$

REACHABILITY::MEET-UNREACHABLE

$$\frac{\text{NEEDS} \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{f}, \text{sink} = \text{b}) \quad \vdash_\ell \text{block}(\text{id} = \text{b}) \quad \neg\exists(\vdash \text{block}(\text{id} = \text{p}, \text{out-edges} = (\ldots, \text{b}, \ldots))). \ \neg\exists(\text{N}_2\colon \text{N}_2 \subseteq \text{NEEDS}) \vdash \text{reachability::active-edge}(\text{active} = \text{f}, \text{source} = \text{p}, \text{sink} = \text{b})}{\text{reachability::assert-reachable}(\text{b}, \text{f}, \text{NEEDS})}$$

REACHABILITY::MEET-PROPOGATE-REACHING

$$\frac{\text{NAE} \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{t}, \text{source} = \text{p}, \text{sink} = \text{b})}{\text{reachability::assert-reachable}(\text{b}, \text{t}, \text{NAE})}$$

REACHABILITY::MEET-SPECULATE-UNREACHABLE

$$\frac{\text{NEEDS} \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{f}, \text{source} = \text{p}, \text{sink} = \text{b}) \quad \vdash_\ell \text{block}(\text{id} = \text{b}) \quad \text{preds} \leftarrow \ (\text{ps} \leftarrow \{\}; \forall \vdash \text{block}(\text{id} = (\text{pred}\colon \text{pred} \neq \text{p}), \text{instrs} = \ldots, \text{out-edges} = (\ldots, \text{b}, \ldots)).\ \text{ps} \leftarrow \text{ps} \cup \{\text{pred}\}; \text{ps})}{\text{reachability::assert-reachable}(\text{b}, \text{f}, \text{NEEDS} \cup (\text{create-needs}, \text{preds}, \text{b}, \text{f}))}$$

REACHABILITY::ELIM-CIRCULAR

$$\frac{(\text{N}_1, \text{nb}, \text{N}_2) \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk}) \quad \vdash_\ell \text{reachability::need-active-edge}(\text{need-id} = \text{nb}, \text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk})}{\text{reachability::assert-active-edge}(\text{a}, \text{src}, \text{snk}, \text{N}_1 \cup \text{N}_2)}$$

REACHABILITY::ELIM-TRUE

$$\frac{(\text{N}_1, \text{nb}, \text{N}_2) \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{active}, \text{source} = \text{source}, \text{sink} = \text{sink}) \quad \vdash_\ell \text{reachability::need-active-edge}(\text{need-id} = \text{nb}, \text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk}) \quad \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk})}{\text{reachability::assert-active-edge}(\text{active}, \text{source}, \text{sink}, \text{N}_1 \cup \text{N}_2)}$$

REACHABILITY::ELIM-TRUE-2

$$\frac{(\text{N}_1, \text{nb}, \text{N}_2) \vdash_\ell \text{reachability::reachable}(\text{block} = \text{b}, \text{is-reachable} = \text{r}) \quad \vdash_\ell \text{reachability::need-active-edge}(\text{need-id} = \text{nb}, \text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk}) \quad \vdash_\ell \text{reachability::active-edge}(\text{active} = \text{a}, \text{source} = \text{src}, \text{sink} = \text{snk})}{\text{reachability::assert-reachable}(\text{b}, \text{r}, \text{N}_1 \cup \text{N}_2)}$$

Figure 11: Rules for module REACHABILITY