Diss. ETH No. XXXXXXX

Incremental Verification

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich

presented by

Uri Juhasz

Msc in Computer Science,
Tel Aviv University

born on 18.01.1979

citizen of Israel

accepted on the recommendation of

Prof. Peter Muller, examiner
Prof. Arie Gurfinkel, co-examiner
Prof. Clark Barrett, co-examiner
Prof. Martin Vechev, co-examiner

20XX

Contents

1 Introduction **1**
- 1.1 Outline . . . 3
- 1.2 Control Flow . . . 4
- 1.3 Locality and Scope . . . 5
- 1.4 Bounded Fragments . . . 7

2 Preliminaries **9**

3 Unit ground first order logic with equality **34**
- 3.1 Congruence closure graphs . . . 35
  - 3.1.1 Data Structure . . . 35
  - 3.1.2 Notation . . . 41
- 3.2 Congruence closure graphs for verification . . . 45
  - 3.2.1 Complexity . . . 46
  - 3.2.2 Joins . . . 50
- 3.3 Information Propagation . . . 63
  - 3.3.1 A verification algorithm . . . 63
  - 3.3.2 Clause propagation criteria . . . 66
  - 3.3.3 Ground unit equality propagation . . . 74
  - 3.3.4 Propagation using sources . . . 77
  - 3.3.5 The equality propagation algorithm . . . 85
- 3.4 Joins . . . 94
  - 3.4.1 Equality propagation . . . 94
  - 3.4.2 Strong join . . . 99
- 3.5 Related Work . . . 103
  - 3.5.1 Congruence closure . . . 103
  - 3.5.2 Information propagation . . . 104

4 Ground first order logic with equality **106**
- 4.1 The fragment of GFOLE . . . 106
  - 4.1.1 Basic clause propagation . . . 111
  - 4.1.2 Redundancy elimination . . . 120
  - 4.1.3 Improvements to the base algorithm . . . 127
- 4.2 Completeness . . . 131
- 4.3 Joins . . . 132
- 4.4 Term, literal and clause ordering . . . 144
- 4.5 Related work . . . 145
  - 4.5.1 Theorem proving and CFGs . . . 145

4.5.2 Clause join . . . 146

5 Scoping and Interpolation 147
5.1 Basics . . . 147
5.1.1 Notation . . . 148
5.2 CFG Node Scope . . . 151
5.3 Ground Unit Interpolation . . . 157
5.3.1 The problem . . . 157
5.3.2 Basic graph-based algorithm . . . 157
5.3.3 Selecting equations to communicate . . . 161
5.3.4 Relevance calculations . . . 165
5.3.5 Scoping in the CFG . . . 170
5.3.6 Related work . . . 178
5.4 Ground Clause Interpolation . . . 181
5.4.1 Basics . . . 181
5.4.2 Approximation of the bottom model . . . 183
5.4.3 Interpolation ordering . . . 186
5.4.4 Interpolation superposition calculus . . . 186
5.4.5 Ordering for Interpolation . . . 188
5.4.6 Related and future work . . . 190

6 Bounded fragments 195
6.1 Term depth . . . 195
6.1.1 Relative depth . . . 196
6.1.2 Enforcing the depth restriction . . . 201
6.2 Size bounds for clauses . . . 202
6.3 Derivation depth . . . 203
6.4 Summary . . . 204

7 Quantification 205
7.1 Preliminaries . . . 205
7.2 Propagation and requests . . . 211
7.3 Quantifiers in bounded fragments . . . 213
7.4 Instantiation . . . 214
7.5 Scoping for quantified clauses . . . 215
7.5.1 Incompleteness . . . 215
7.5.2 Completeness for some restricted cases . . . 216

8 Implementation and experimental results 217
8.1 Order of evaluation . . . 217
8.2 Implementation . . . 221
8.3 Experimental results . . . 224

9 Conclusion and future work 232
9.1 Main contributions . . . 232
9.2 Experience and directions for future work . . . 233

A Appendix **235**
A.1 Interpolation 235
A.1.1 Completeness proof for GFOLE interpolation 236
A.1.2 Improved justification extraction 249
A.2 EC-graphs and clauses 255
A.2.1 Non-unit fall-back at joins 258

Abstract

Many current verification tools use a logical reasoning engine as a back-end, mostly SMT-solvers. While SMT-solvers are efficient and relatively predictable for quantifier free problems, they are not, in general, complete for problems with quantifiers, and tend to show unpredictable performance in the presence of quantifiers, often requiring expert attention for verification to succeed. More advanced software verification goals require the use of quantifiers in order to model concepts for which no decision procedure is implemented in the solver, such as modeling ownership or permissions on the heap.

Automated theorem provers (ATPs) based on the superposition calculus perform significantly better when reasoning with quantifiers, but are weaker for problems with large propositional parts - which many program verification problems contain - and for problems that include significant theory reasoning, such as linear integer arithmetic. A major difference between ATPs and SMT-solvers is that ATPs perform small steps that guarantee progress (generating new clauses), while SMT-solvers must find a conflict (a contradiction) in order to progress, and the number of steps to reach a contradiction is not bounded for problems with quantifiers - hence progress cannot be guaranteed for any number of steps.

In this thesis we develop a verification technique that embeds theorem proving deeply in the verifier. Our technique maintains a superposition-based theorem prover at each program point, and allows these provers to communicate relevant information in order to build a proof for the whole program. Our technique takes advantage of some properties of programs, such as the control flow structure and lexical scope, to restrict the proof search space. Each prover works incrementally in logical fragments of increasing strength and predictable worst-case performance. We define a hierarchy of logical fragments based on size measures of clauses and the derivation depth of clauses in the calculus. In our technique, each prover handles a much smaller problem than the verification of the entire program, and much of the control flow is handled directly, reducing the propositional part of the problem significantly.

We prove that our technique is complete for FOL without theories, and a weaker completeness result when lexical scoping is enforced. We have implemented our technique and, while we cannot prove many programs because of the lack of arithmetic support, our implementation can be used as a pre-processing step before an SMT-solver to reduce the overall proving time. Our implementation applies successively stronger logical fragments, each with predictable worst case run-time, and intermediate results (such as proven assertions) can be extracted at any time.

Sommario

Molti degli attuali software di verifica utilizzano uno strumento di ragionamento logico come back-end, per lo pi dei risolutori SMT. Sebbene questi siano efficienti e relativamente prevedibili per formule senza quantificatori, in generale, non sono completi per formule con quantificatori e tendono ad avere prestazioni imprevedibili in loro presenza, il che spesso richiede l'attenzione di un esperto perch la verifica di un programma abbia successo.

I dimostratori automatici di teoremi basati sul superposition calculus danno risultati significativamente migliori in presenza di quantificatori, ma risultano pi deboli per problemi che contengono in larga parte formule proposizionali - il che il caso per molti problemi di verifica di programmi - e per problemi che richiedono la risoluzione modulo una teoria, come l'aritmetica lineare intera. Un'importante differenza tra i dimostratori automatici di teoremi e i risolutori SMT che i dimostratori automatici di teoremi eseguono piccoli passi che garantiscono di compiere progresso (generando nuove clausole), mentre i risolutori SMT devono trovare una contraddizione al fine di progredire, e il numero di passi per raggiungere una contraddizione illimitato in presenza di quantificatori - non possibile garantire di compiere progresso per qualsiasi numero di passi.

In questa tesi sviluppiamo una tecnica di verifica che incorpora la dimostrazione automatica di teoremi nel software di verifica. La nostra tecnica associa ad ogni punto di controllo di un programma un dimostratore automatico di teoremi basato sul superposition calculus, e permette a questi dimostratori di comunicare informazioni rilevanti al fine di costruire una prova per l'intero programma. La nostra tecnica, per limitare lo spazio di ricerca di una soluzione, si avvale di alcune propriet dei programmi, come la struttura del flusso di controllo e la visibilit delle variabili. Ogni dimostratore opera in modo incrementale in frammenti logici di crescente espressivit e con prevedibile tempo di esecuzione nel caso peggiore. Definiamo una gerarchia di frammenti logici basati sulla misura della dimensione delle clausole e della profondit di derivazione di clausole nel calcolo. Nella nostra tecnica, ciascun dimostratore gestisce un problema molto pi ristretto della verifica dell'intero programma, e gran parte del flusso di controllo viene gestito direttamente, riducendo significativamente la componente proposizionale del problema.

Dimostriamo che la nostra tecnica completa per la logica del prim'ordine senza teorie, e dimostriamo un risultato di completezza pi debole quando la visibilit delle variabili presa in considerazione. Abbiamo implementato la nostra tecnica e, sebbene non possa verificare molti programmi a causa della mancanza di supporto per l'aritmetica lineare, la nostra implementazione pu essere utilizzata in fase di pre-elaborazione prima dell'invocazione un risolutore SMT per ridurre il tempo complessivo necessario per la verifica. La nostra implementazione utilizza frammenti logici sempre pi espressivi, ciascuno con prevedibile tempo di esecuzione nel caso peggiore, e i risultati intermedi (come le asserzioni provate) possono essere estratti in qualsiasi momento.

Chapter 1

Introduction

Recent advances in the technology of automated theorem proving (ATP) and abstract interpretation allow the automated verification of increasingly large and complex programs. Fully automated verification tools can annotate a program with loop and recursion invariants and verify that the annotated program is correct according to a given specification.

In this thesis we are interested in verifying the correctness of (manually or automatically) annotated programs rather than the inference of invariants - this verification problem is interesting in itself when user-given annotations are available and is commonly a sub-problem for more automated tools that try to verify approximations of the program (such as bounded loop unrolling) and approximations of the annotation (such as loop invariants generated by abstract interpretation or by interpolation on the proof of an unrolled program).

Popular program verification techniques (for annotated programs) include variants of verification condition (VC) generation (VCG) and symbolic execution.

In VCG based verification, an annotated program and a specification are encoded in a mathematical formula that holds iff the program satisfies the specification, as pioneered by the work of Dijkstra with weakest preconditions ([Dij75]). This formula can be fed to an automated theorem prover. VCG allows the use of general first order logic (FOL) theorem provers, whether based on SMT solving (e.g. [BCD$^+$11],[dMB08]), completion ([HJL99],[RV99]) instantiation ([Kor08]), or other ([BPT12]), although SMT solvers are, by far, the most common. One of the main disadvantages of VCG is that the translation to FOL loses some explicit information about the program which can be useful for the proof search, such as the control structure of the program and the scope of variables.

Symbolic execution simulates executing the program on each possible path, using symbolic values rather than concrete values for variables. The symbolic execution engine evaluates the feasibility of a program path by using a constraint solver (often a SAT or SMT solver) to check the satisfiability of the branch conditions on a path with the calculated symbolic values - hence, essentially, several small VCs are sent to the solver, each for a specific path in the program. Symbolic execution tools are sensitive to the problem of path explosion - the number of feasible paths in a program can be exponential in the size of the program, even though the reasoning needed for proving two non-disjoint paths can be very similar.

SMT solvers have made significant progress in recent years, and are quite efficient and reliable in solving problems that involve ground FOL combined with other ground theories, notably linear integer and rational arithmetic and arrays. However, when the VC includes quantified formulae, SMT solvers are, in general, not complete, and the performance and even termination of the solver is very sensitive to the input formula, even to parts that are logically unrelated to the proof (as noted e.g. in [LP16]). Quantified VCs are needed for modeling abstractions such as sets and sequences used in specifications, for encoding some invariants and for modeling used in some verification methodologies, such as permissions or dynamic frames for alias control.

The instability of SMT solvers in the presence of quantifiers makes verifiers based on SMT solvers less accessible even to people that are familiar with program proofs, specification and first order logic - users need understanding of the working of the SMT solver in order to understand why verification fails, especially for a program that previously verified and has been modified only slightly. In addition, as FOL is only semi-decidable, any theoretically complete tool will not be bounded in its run-time. For users, it is sometimes useful to know that if a proof for a property of the program was found using a certain effort of the verifier, and the program has been modified but the proof of the property carries over to the modified program, then the property will be proven if the same effort is spent by the verifier on the modified program, regardless of other properties and unrelated modifications of the program. Classic resource limitations on the prover, such as memory and time, cannot usually satisfy this requirement. In addition, if some of the properties of a program are easy to prove while others require longer time or do not hold, it is useful to show the user of a verification tool intermediate results, such as proven properties, while still searching for proofs for the harder properties. Such intermediate results can also be used to cooperate between verification and analysis tools. While the verifier can be run for each property separately, this is often highly inefficient as the proofs of different properties of the same program often share a large number of lemmas.

ATPs based on superposition (e.g. [RV99]) handle quantifiers efficiently, and, in addition, can be modified to search incrementally for proofs of increasing complexity - for example, the proof depth can be bounded (by blocking inferences of maximal depth). Such ATPs have seen much less use in program verification as superposition is not very efficient in handling ground and propositional formulae, which are often the majority of the VC for program verification. In addition, the extensions suggested for superposition based solvers to support linear arithmetic are not as efficient as those for SMT solvers, and integer arithmetic is often needed for program verification.

Program analysis tools based on abstract interpretation calculate an over-approximation of the set of feasible program states at each program point, by applying an abstraction to the state and transition relation of the program. If the approximation satisfies the specification for that program point, so does the program. For a given abstraction, it is sometimes possible for a programmer to predict which properties of a program will be proved, and the approximation at each program point is not affected by modifications to disjoint parts of the program - hence the results of the tool are more stable and predictable to non-expert users. Abstract interpretation can be applied in abstractions of increasing strength, so that the user can expect that properties proven in a certain abstraction will be proven with the same abstraction if the proof is valid for the modified program. Very few abstract interpreters were suggested for quantified domains (e.g. [GMT08]), and they often have to lose precision at join points as they are not goal sensitive - the analyzer at a join point cannot predict what precision is needed to prove properties later in the program.

Main contribution:
In this thesis we propose a generic verification algorithm that is based on a tighter integration between theorem proving and verification. Our algorithm is based on the idea of having a local theorem prover at each program point rather than one global prover, and allowing these provers to exchange information in order to search for a proof for the entire program. Information is exchanged only on-demand between provers rather than eagerly as often in abstract interpretation. Our algorithm is incremental and applies successively stronger logical fragments in order to prove a program, allowing the report of intermediate results to the user. For a given fragment, the proof of each property is independent from the proofs of other properties, but the proofs can still share lemmas.

We have instantiated our algorithm for the fragment of ground equalities using a form of congruence closure graphs, and for general FOL using superposition. We have also implemented a hierarchy of bounded fragments that restrict the proof-tree shape in various ways, including the size of terms and proof depth.

We show how our algorithm can take advantage of the scoping inherent in many programming languages, so that the vocabulary of each prover is small and local - we show how to preserve completeness under scoping using interpolation for ground FOL.

As we have implemented only very basic support for linear integer arithmetic, our tool cannot prove many VCs on its own. However, as intermediate results are usable at any stage, we can use the tool as an optimizing pre-processor before running an SMT solver or other tool.

1.1 Outline

In the rest of this chapter we give an overview of the main ideas in this thesis. In chapter 2 we discuss the theoretical background that we assume, our notation and the structure of input programs that we can handle. In chapter 3 we present our verification algorithm and instantiate it for unit ground equalities using a form of congruence closure graphs. In chapter 4 we instantiate our algorithm for ground clauses using ground superposition. In chapter 5 we show how our algorithm is adapted to search for local proofs, and the use of interpolation to preserve completeness for local proofs in the ground case. Chapter 6 introduces several restrictions on the shape of proofs that define a hierarchy of decidable logical fragments with predictable complexity, whose limit is the complete fragment of first order logic, thus allowing incremental verification in fragments of increasing strength. We show how the algorithm developed for ground clauses is extended to quantified clauses in chapter 7. In chapter 8 we discuss some implementation issues and present experimental results for an implementation of some of the ideas in this thesis. We conclude in chapter 9 and discuss future work.

1.2 Control Flow

When generating the VC for a given program, some information about the CFG has to be encoded into first order logic. Mainly, for an assertion anywhere in the code, only information from statements that precede the assertion can be used to prove the assertion, and two statements on parallel branches are treated differently than two statements in sequence. The weakest precondition (WP) calculus ([Dij75]) is often used as a basis for converting a program and a specification to a logical formula that is valid iff the program satisfies the specification. A WP calculus suitable for verification with an intermediate verification language is described in [Lei05], which assumes a program constructed of `assert` and `assume` statements. As noted in the paper, applying the original formulation of WP to a program with n sequential non-nested branches produces a VC of size that is exponential in n (essentially a case split on each possible path in the CFG), and hence a more efficient formulation is required for many programs. A common method for encoding a polynomial sized VC (also suggested in [Lei05]) is to use a nullary predicate per basic block of the original program, which represents either that execution has reached that basic block or that, if execution reaches that basic block, then all assertions in the block and its successors hold. In either encoding, some implications are encoded between these nullary predicates which encode the structure of the CFG. However, in any encoding into a logic that does not support graph structures directly, the explicit graph structure is lost and hence some graph properties that are easily exploitable in an analysis that uses the CFG are lost. To prove that an assertion holds, it must be shown that it holds on any path reaching it from the root. For each such path, the prover has to show that after executing the statements on the path the assertion holds. As we only treat passive statements (`assert` and `assume`) for the DSA form of the program, the order of execution of the statements on the path does not matter for the proof of the program - hence for each path we have to show that the set of assumed or asserted formulae on the path implies the formula in the assertion. We call these sets of formulae the relevant sets for proving the assertion - for example, in the program in figure 1.1, for `assert e3`, the set {{`S1,S3`},{`S2,S3`}} is the set of all relevant subsets of the program statements relevant for the assertion. Each such subset represents a path reaching the assertion from the root.

Relevance

The first property that is made implicit in the encoding of the VC is relevance - for example, assume that the above program is encoded into a VC and sent to an SMT solver, and, during the proof search, the SMT solver decides on a literal that encodes the fact that the trace passes through `S1` (that is, the **`then`** branch of the first conditional is taken), or, depending on the exact VC encoding, that an assertion does not hold on a path that passes through `S1`. There is nothing preventing the SMT solver from deciding on a literal that occurs only in `S2`, although this is not necessary in order to prove any assertion - a model (trace of a failing assertion) that encodes a trace that passes through

```
if (b1)
   S1;
   assert e1;
else
   S2;
   assert e2;
j1:
if (b2)
   S3;
   assert e3;
else
   S4;
   assert e4;
```

Figure 1.1: Example for the information contained in the CFG

`S1` is fully defined by the interpretation for symbols that occur on the path of the trace - which may not include some of the symbols that occur `S2`. Some of the simplifications performed by SMT solvers (unit propagation and pure literal elimination) can prevent some of these cases, but for complicated CFGs the proportion of the cases eliminated is limited. Many of the more successful SMT solvers use incremental or lazy CNF conversion (e.g. [BDS02a],[DNS05]) which can prevent much of the interference when the VC is encoded carefully. The parallel of the above for a superposition based prover is that if `S1` and `S2` are each encoded into a set of clauses, there is no need to perform any inference between a clause that encodes `S1` and a clause that encodes `S2`, as this inference will not participate in the proof that the assertion holds on any path of the program.

Joins

A second property of the CFG that is less exploitable on a monolithic VC is that of sharing lemmas on joins. Consider the case where some lemma C implied separately by `S1` and `S2` is sufficient, together with the encoding of `S3`, to prove that the assertion `assert e3` holds. For propositional logic, lemmas do not include new literals (although some proofs can be shortened by introducing new literals - e.g. the pigeonhole principle in extended resolution), for ground first order logic with equality (GFOLE - called QF_UF in the SMT community) and more so for FOLE, the introduction of new literals, even if constructed only from the VC vocabulary, can sometimes allow a significantly shorter shortest proof. Abstract interpretation tools search explicitly for such lemmas in a fixed logical fragment. CDCL based SMT solvers ([SS99]), in general, can learn some of these lemmas (we discuss this issue in more detail later - see also [BDdM08]). Superposition based provers can also sometimes generate join lemmas.

1.3 Locality and Scope

Many current programming languages support scoping for program variables, where a variable can be accessed only in a certain area of the program. For example, a loop counter may be in scope only within the loop body. An encoding of a program VC most often represents program variables in some form of dynamic single assignment (DSA) form - this form ensures that each program variable is assigned at most once on each program path (usually a program to program transformation replaces each occurrence of a a program variable with some indexed version of that variable to ensure this property. SSA is a specific case of DSA.). Also, a VC for the unrolling of a program (e.g. as in [AGC12b]) often uses some form of DSA. Note that the original WP encoding of Dijkstra does not introduce any new symbols, DSA or otherwise, and instead represents, for each post-condition and each path in the program, the final value of each program variable that occurs in the post-condition expressed as an expression over the initial value of the program variables. As the number of such paths can be exponential in the program size, additional symbols must be introduced to keep the VC in size polynomial in the input size.

Intuitively, the state of an execution of the program is defined by the current program point and the values of all program variables. For heap manipulating programs, the heap must be modeled in some way so that the value of the heap at different program points is representable as a FOL term, or we face the same problem of exponential sized WP. For example, the Boogie encoding of a program models the heap as an update-able map, for which Boogie emits axioms to the prover - the update-able map behaves as other program variables and hence has several DSA versions. For assertions that refer to earlier versions of variables - specifically, post-conditions that refer to both the initial and final value of a variable - each earlier version of a variable that is later referenced has to be added to the state.

With this intuition in mind, we can expect that program annotation in the style of Floyd, where each program point is annotated by a formula that describes the set of possible states at that point, will include only the *current* DSA version of each variable. In terms of FOLE, this means that the only constants that participate in the program annotation at a given program point are the constants that represent the latest DSA versions of each program variable (more care is needed with join points, where we must ensure that each program variable has the same current DSA version on all the predecessors of the join).

We call program annotations that only mention the current DSA version scoped annotation. The search space for a scoped annotation depends on the number of variables in the source program, while the search space for a non-scoped annotation depends on the number of DSA versions times the number of source program variables - thus looking for only scoped proofs can reduce the proof search-space significantly in some cases. This reduction is especially significant with some techniques for handling quantifiers, where the number of scoped ground instances of a quantified axiom is much smaller than the number of global ground instances.

Remember that the axioms defining the semantics of statements (whether by Floyd, Hoare or others) always correlate the state before and after the execution of the statement, and hence each axiom instance can relate to more than one program point (generally, assignment and `skip` statement axioms refer to two program locations while a binary branch or join axiom refers to three). In addition, each sub-formula of the axiom refers to one specific program location - for example, in the Hoare axiom $\mathrm{Q}[\mathrm{x} \mapsto \mathrm{v}]\{\mathtt{x:=v}\}\mathrm{Q}$, the sub-formula $\mathrm{Q}[\mathrm{x} \mapsto \mathrm{v}]$ refers to the program point before the statement and Q refers to the program

```
method m(n : Integer,b:Boolean)
   requires n>0

   //new array initialized to all false
   a := new Array[Boolean](n)
   a[0]:=true
   if (b)
      a[0] := false
      j := random(0,n)
      a[j] := true
n₁:
   assert ∃i · 0 ≤ i <length(a)∧a[i]=true
```

Figure 1.2: Example for the incompleteness of scoped annotation in universal CNF.
The `random` function is specified as `a≤random(a,b)<b`
A possible scoped annotation at n_1 that is sufficient to prove the assertion is
$\exists i \cdot (0 \leq i <$ `length(a)`$\wedge$`a[`i`]=`**`true`**$)$
However, this annotation is not in the fragment of universal CNF which is used by many provers, and into which there is a validty preserving conversion from FOLE.
A possible non-scoped universal CNF annotation is
$($`b`$\Rightarrow (0 \leq$`j<length(a)`$\wedge$`a[j]=`**`true`**$))\wedge(\neg$`b`$\Rightarrow$`a[0]=`**`true`**$)$
There is no scoped annotation for universal CNF

point after the statement. For a scoped proof, this means that we must only consider axiom instances where each sub-formula only contains the DSA versions of variables relevant for the program point it refers to.

The technique we develop in this work can be used to search for a scoped proof, and we mention some logical fragments where this is complete. In other cases we can prioritize the search for a scoped proof over a non-scoped proof or limit the scope in a less strict way while still preserving completeness.

Scoped proofs do not exist for all logical fragments, for example, consider the code in figure 1.2 - it is easy to see that any scoped annotation at n_1 must include an existential quantifier, and hence there is no scoped proof in universal CNF (most calculi used by automated theorem provers do not generate existential conclusions).

We discuss scoped proofs in chapter 5 and also their relation to interpolation. We do not always look for scoped proofs because some logical fragments do not admit a scoped annotation, and sometimes the size of a minimal scoped annotation is significantly larger that of a minimal non-scoped annotation.

1.4 Bounded Fragments

As FOLE is only semi-decidable, and combined with some theories becomes not even recursively enumerable (e.g. linear integer arithmetic with uninterpreted functions and quantifiers as shown in [KV07]), it is common for program reasoning tools to select decidable fragments of the logic with lower complexity in order to ensure the predictability of the tool. For example, compilers often approximate the possible set of values of variables at each program point using analyses that are guaranteed to terminate quickly, such as constant propagation and definite assignment, for optimization and for reporting warnings and errors to the programmer.

Often, the result of simple and efficient analyses can be used to simplify a program VC, which sometimes shortens verification time. We take advantage of several such simple analyses and apply them exhaustively after each application of stronger and more expensive fragments - rather than just as a pre-process.

We verify the program by defining a hierarchy of logical fragments, each of which has a predictable polynomial complexity, and apply these fragments in succession until the program is proven or a user-chosen limit is reached. Our approach differs from refinement methods such as [CGJ$^+$03] in that the user can select exactly which fragments are applied (rather than depending on a counter-example whose choice is less predictable), and so the performance should be more predictable. The intuition is that, while the proof for an entire program VC may be very deep, the actual part of the proof performed at each program point (in e.g. Hoare logic) is often small. We define the hierarchy of fragments in a way that allows us to search first for small lemmas at program points which combine together to form a proof for the whole program.

Bounded terms: The intuition for limiting the size of terms comes from axiomatizations of recursive abstract predicates used in specification (e.g. for modeling recursive data structures as in [HKMS13]) and generally from proofs of heap manipulating programs. Often, the recursive definition of a predicate is given as an axiom which, when instantiated, allows to *unfold* the definition of the predicate once - e.g. for a predicate defining the validity of a recursive search tree, instantiating the defining axiom for a node produces the instances of the predicate for the node's children. Such an axiomatization is not complete in FOLE, but is sufficient in many cases. In our experience, proofs for such structures do not often require an arbitrary depth of unfolding of the predicate, and hence limiting instantiation to instances with terms that are not very distant (in terms of number of function applications) from input VC terms, should allow us to look first for proofs that do not look in the heap much deeper than the actual program does. We define a measure of term depth that is relative to the set of original VC terms, and also takes into account any equalities derived for the terms. Using this measure, we define a hierarchy of logical fragments where each fragment extends the limit on the relative size of terms.

Bounded derivation depth: We try to prioritize the search for simpler proofs over the search for complicated proofs, and hence we limit the shape of the proof DAG by classifying inferences according to a cost measure, and limiting the number of inferences of each class in each path in the proof DAG. Inferences that strictly reduce the VC size and are cheap to perform (such as unit propagation), are not limited. Other more expensive inferences (such as ground superposition) are limited for each fragment by the maximal number allowed on each proof DAG path. The most expensive inferences, such as superposition of non-unit clauses with two non-ground premises where the conclusion has many free variables, are restricted more. Using the maximal number in a proof DAG path is a compositional measure in the sense that it is easy to calculate the measure for each node of a proof DAG from the measure of its immediate children.

We also use a bound on the number of literals allowed in a clause, in order to prevent some cases of combinatorial explosion.

Chapter 2

Preliminaries

In this chapter we present the formalisms with which we work.
Most of the chapter is a repetition of standard definitions, except for section 2 which discusses our assumptions about the input program, and transformations we perform on it before our algorithm begins.

Sets, multisets and sequences

A multiset m over a set S is a function $m : S \to \mathbb{N}_0$.
We use 0 for the empty multiset.
Singleton multisets are defined as:
$[x \mapsto n](y) \triangleq \textbf{if } (x \equiv y) \textbf{ then } n \textbf{ else } 0$.
Multiset union is defined as:
$(m \cup n)(x) \triangleq \lambda x \cdot m(x) + n(x)$.
Sub-multiset relation is defined as follows:
$m \subseteq n \overset{\triangle}{\Leftrightarrow} \forall i \in \mathbb{N}_0 \cdot m(i) \leq n(i)$
An equivalence relation $\approx$ over a set S is a subset of S^2 s.t.
$\forall (x,y) \in\approx \cdot (y,x) \in\approx$
$\forall (x,y),(y,z) \in\approx \cdot (x,z) \in\approx$
$\forall x \in S \cdot (x,x) \in\approx$
For an equivalence relation $\approx$ over S, and a term $x \in S$ we denote by $[x]_\approx$ the equivalence class of $x \in S$ with respect to $\approx$:
$[x]_\approx \triangleq \{y \in S \mid x \approx y\}$
A partition of a set S - $P \subseteq \mathcal{P}(S)$ - satisfies:
$\forall S_1, S_2 \in P \cdot S_1 \cap S_2 = \emptyset$
$\cup P = S$
The quotient set of a set S for the equivalence relation $\approx$, $S/\approx$ is the set:
$\left\{[x]_\approx \mid x \in S\right\}$
and is a partition of S, and similarly a partition defines an equivalence relation.

A (finite) sequence of length n of elements of a set S is a function from $0..n-1$ to S. For a sequence s we use s_i for the ith element of the sequence - s(i). $|s|$ is the length of the sequence ($|\textbf{dom}(s)|$). We sometimes use the notation $[i \mapsto e(i)]$ to denote sequences, where e(i) is an expression that defines the ith element, and $< s >$ for a singleton sequence. We use this notation where the domain is unambiguous. The concatenation of the sequences s,t is denoted by s.t. We use sequences to denote paths in the program CFG (as a sequence of CFG nodes) and we extend the notation for concatenation to include single CFG-nodes - so, for example, if n,p are nodes and P,Q are sequences of nodes, r.P.p.Q is the sequence P. $< p >$ .Q. $< n >$.

Logic

Syntax and notation

A language is defined formally as follows:
A function or predicate symbol (denoted $\mathrm{f}, \mathrm{g}, \mathrm{h}$) has a fixed arity ($\mathbf{arity}(\mathrm{f}) \geq 0$).
A signature $\Sigma = \mathbf{F}_\Sigma \cup \mathbf{P}_\Sigma \cup \mathbf{X}_\Sigma$ is a set of function symbols $\mathbf{F}_\mathbf{\Sigma}$, predicate symbols $\mathbf{P}_\mathbf{\Sigma}$ and variables $\mathbf{X}_\mathbf{\Sigma}$.
We only handle finite programs and sets of axioms, so that the sets $\mathbf{F}_\Sigma, \mathbf{P}_\Sigma$ used in any VC are finite. However, in order to be able to handle theories with countable signatures, such as integer arithmetic, we allow both these sets to be countable. The set of constants is $\mathbf{C}_\mathbf{\Sigma} = \{\mathrm{f} \in \mathbf{F}_\mathbf{\Sigma} \mid \mathbf{arity}(\mathrm{f}) = 0\}$, we assume $\|\mathbf{C}_\mathbf{\Sigma}\| > 0$ (otherwise the ground fragment is trivial).
We denote a (possibly empty) tuple by an overline - detailed in the following.
We use a standard definition for the language:

function	f,g,h			$\in \mathbf{F}_\mathbf{\Sigma}$	functions
predicate	P,Q,R			$\in \mathbf{P}_\mathbf{\Sigma}$	predicates
variable	x,y,z			$\in \mathbf{X}_\mathbf{\Sigma}$	variables
term	t,s,u,v	::=	$\mathrm{f}(\bar{\mathrm{t}}) \mid \mathrm{x}$	$\in \mathbf{T}(\mathbf{\Sigma})$	free term algebra
atom	a	::=	$\mathrm{t} = \mathrm{t} \mid \mathrm{P}(\bar{\mathrm{t}})$	$\in \mathbf{A}(\mathbf{\Sigma})$	the atoms over Σ
literal	A,B	::=	$\mathrm{a} \mid \neg\mathrm{a}$	$\in \mathbf{L}(\mathbf{\Sigma})$	the literals over Σ
clause	C,D	::=	$\Box \mid \mathrm{A} \mid \mathrm{A} \vee \mathrm{C}$	$\in \mathbf{C}(\mathbf{\Sigma})$	the clauses over Σ

Figure 2.1: language

We use $\mathrm{s}, \mathrm{t}, \mathrm{u}, \mathrm{v}$ for terms, $\bar{\mathrm{s}}, \bar{\mathrm{t}}, \bar{\mathrm{u}}, \bar{\mathrm{v}}$ for term tuples, we also construct tuples from terms using parenthesis - e.g. (t, s).
We occasionally treat an n-tuple as a sequence of n ground terms.
t_i is the i-th element of the tuple $\bar{\mathrm{t}}$ and $|\bar{\mathrm{t}}|$ is the number of terms (size or length) in a tuple.
We treat an equality atom as an unordered set and so $(\mathrm{s} = \mathrm{t}) \equiv (\mathrm{t} = \mathrm{s})$.
We use $\mathrm{s} \neq \mathrm{t}$ to denote $\neg \mathrm{s} = \mathrm{t}$.
As we do not manipulate negations syntactically, we consider $\neg\neg\mathrm{a} \equiv \mathrm{a}$.
We treat clauses as sets of literals whose semantics is the disjunction of these literals.
We denote the empty clause by $\Box$.
We use $\bowtie$ to denote either $=$ or $\neq$.
We define the set of terms $\mathbf{terms}(\mathrm{S})$ of a set of clauses as follows:

$$
\begin{array}{lcl}
\mathbf{terms}(S) & \triangleq & \bigcup\limits_{C \in S} \mathbf{terms}(C) \\
\mathbf{terms}(C) & \triangleq & \bigcup\limits_{l \in C} \mathbf{terms}(l) \\
\mathbf{terms}(s \bowtie t) & \triangleq & \mathbf{terms}(s) \cup \mathbf{terms}(t) \\
\mathbf{terms}(P(\bar{s})) & \triangleq & \mathbf{terms}(\bar{s}) \\
\mathbf{terms}(\bar{s}) & \triangleq & \bigcup\limits_{i} \mathbf{terms}(s_i) \\
\mathbf{terms}(f(\bar{s})) & \triangleq & \{f(\bar{s})\} \cup \mathbf{terms}(\bar{s})
\end{array}
$$

For a set S we denote by $F_n(S) \triangleq S^n \to S$ the set of all functions of arity n over S and $F(S) \triangleq \bigcup\limits_{n \in \mathbb{N}_0} F_n(S)$ the set of all functions over S.

Similarly, we define relations over S as $R_n(S) \triangleq P(S^n)$ and $R(S) \triangleq \bigcup\limits_{n \in \mathbb{N}_0} R_n(S)$.

For the semantics we use functions from terms to a domain D, $f : \mathbf{T}(\mathbf{\Sigma}) \to D$. When applying such a function to a tuple $\bar{t}$ we mean the point-wise application of the function that returns a tuple in $D^{|\bar{t}|}$ - so $f(\bar{t})_i = f(t_i)$.

A structure $\mathbf{S} = (D_S, F_S, P_S)$ for a signature $\mathbf{\Sigma}$ includes:

- A domain D_S which is a non empty set
- An interpretation for function symbols F_S which maps each function of $\mathbf{\Sigma}$ to a function of the corresponding arity over D_S - namely $\mathbf{F_S} \in \mathbf{F_\Sigma} \to F(D_S)$ such that $\forall f \in \mathbf{\Sigma} \cdot F_S(f) \in F_{\mathbf{arity}(f)}(S)$.
- An interpretation $\mathbf{P_S}$ for predicate symbols that maps each predicate symbol of arity n to a relation of arity n over D_S - namely $\mathbf{P_S} \in \mathbf{P_\Sigma} \to R(D_S)$ such that $\forall P \in \mathbf{\Sigma} \cdot P_S(P) \in R_{\mathbf{arity}(P)}(S)$.

An interpretation $\mathbf{I} = (S_I, \sigma_I)$ is a structure $\mathbf{S_I} = (D_I, F_I, P_I)$ and a variable assignment $\sigma_I : \mathbf{X_\Sigma} \to D_I$.

Semantics

For a term t and an interpretation $\mathbf{I} = (S_I, \sigma_I)$ we denote by $[\![t]\!]_{\mathbf{I}} \in D_I$ the interpretation of t in $\mathbf{I}$ in the standard way, as defined below:

$$
\begin{array}{lcl}
[\![f(\bar{t})]\!]_{\mathbf{I}} & \triangleq & [\![f]\!]_{\mathbf{I}}([\![\bar{t}]\!]_{\mathbf{I}}) \\
[\![x]\!]_{\mathbf{I}} & \triangleq & \sigma_I(x) \\
[\![t = s]\!]_{\mathbf{I}} & \triangleq & [\![t]\!]_{\mathbf{I}} = [\![s]\!]_{\mathbf{I}} \\
[\![P(\bar{t})]\!]_{\mathbf{I}} & \triangleq & [\![\bar{t}]\!]_{\mathbf{I}} \in [\![P]\!]_{\mathbf{I}} \\
[\![\neg a]\!]_{\mathbf{I}} & \triangleq & \neg [\![a]\!]_{\mathbf{I}} \\
[\![C]\!]_{\mathbf{I}} & \triangleq & \forall I' \cdot S_I = S_{I'} \Rightarrow \bigwedge\limits_{A \in C} [\![A]\!]_{\mathbf{I'}}
\end{array}
$$

We extend $[\![\cdot]\!]_{\mathbf{I}}$ point-wise to tuples, and use $[\![f]\!]_{\mathbf{I}}$ for $F_I(f)$ and $[\![P]\!]_{\mathbf{I}}$ for $P_I(P)$.
For the ground fragment, where $\mathbf{X_\Sigma} = \emptyset$, an interpretation is essentially a structure.
Satisfiability in the ground fragment of this language is decidable and NP-complete:
We can reduce (in linear time and space) a propositional problem to our fragment by replacing each propositional atom A with the GFOLE atom $f_A() = T$ where f_A is a constant function and T is a specially designated (fresh) true symbol. In the other direction we have a polynomial reduction using the Ackermann

transformation - basically, encode every term $f(\bar{t})$ by a fresh variable $v_{f(\bar{t})}$, for any pair of terms $f(\bar{t})$, $f(\bar{s})$ with the same function symbol we add the clause $\bigvee_i v_{t_i} \neq v_{s_i} \vee v_{f(\bar{t})=f(\bar{s})}$ to encode congruence closure.
We are left with a set of CNF clauses over the fresh variables, with no non-constant functions symbols, of at most square size. We replace each atom $v = u$ by a propositional atom $A_{v=u}$, and for each triple of constants a, b, c we add the clause $\neg A_{a=b} \vee \neg A_{b=c} \vee A_{a=c}$ to encode transitivity - we end up with an equi-satisfiable propositional set of clauses. There are more efficient transformations that achieve the same, however we are interested mostly in the fact that the reduction is polynomial, as the best known algorithm for propositional CNF is exponential.

Terms

Substitutions
A substitution on a signature Σ is a total function $\sigma : X \to \mathbf{T}(\mathbf{\Sigma})$ extended to terms as follows:

$$\begin{array}{lcl} x\sigma & \triangleq & \sigma(x) \\ f(\bar{t})\sigma & \triangleq & f(\bar{t}\sigma) \\ \bar{t}\sigma_i & \triangleq & t_i\sigma \end{array}$$

Term positions
We denote by ϵ the empty (integer) sequence and by `i.s` the sequence constructed by prepending the integer `i` before the sequence `s`.
A position is a sequence of integers.
We denote a sub-term of the term u at position p by $u|_p$ - which is (partially) defined recursively as follows:

$$\begin{array}{lcl} u|_\epsilon & \triangleq & u \\ f(\bar{r})|_{i.s} & \triangleq & r_i|_s \end{array}$$

For example, for $t = f(g(a), h(b, g(a)))$,
$f(g(a), h(b, g(a)))|_\epsilon = f(g(a), h(b, g(a)))$,
$f(g(a), h(b, g(a)))|_0 = f(g(a), h(b, g(a)))|_{1.1} = g(a)$,
$f(g(a), h(b, g(a)))|_{0.0} = f(g(a), h(b, g(a)))|_{1.1.1} = a$
etc.
For a term t the set $\mathbf{pos}(t)$ is the set of all positions of t defined as follows:

$$\mathbf{pos}(f(\bar{s})) \triangleq \{\epsilon\} \cup \{i.p \mid p \in \mathbf{pos}(s_i)\}$$

We also use all positions of a term s in a term t:

$$\mathbf{pos}(f(\bar{s}), t) \triangleq \{\epsilon \mid f(\bar{s}) \equiv t\} \cup \{i.p \mid p \in \mathbf{pos}(s_i, t)\}$$

Two positions p, q are disjoint, denoted $p \ \mathbf{disj} \ q$, iff they do not share any common sub-term - formally:
$p \ \mathbf{disj} \ q \overset{\triangle}{\Leftrightarrow} \exists i, j \cdot (p = i.p' \wedge q = j.q' \wedge (i \neq j \vee p' \ \mathbf{disj} \ q'))$
The set $\mathbf{pos}(t, s)$ is pairwise disjoint.
By $u[t]_p$ we denote a replacement of $u|_p$ by t at the position p in term u - formally:

$$\begin{array}{lcl} s[t]_\epsilon & \triangleq & t \\ f(\bar{s})[t]_{i.p} & \triangleq & f\left(\bar{s}\left[i \mapsto (s_i)[t]_p\right]\right) \end{array}$$

We extend this notion to simultaneous replacement on a pairwise disjoint set of positions P:

$$\begin{array}{lcl} s\,[t]_{\emptyset} & \triangleq & s \\ s\,[t]_{\{\epsilon\}} & \triangleq & t \\ f(\bar{s})\,[t]_{P} & \triangleq & f(i \mapsto s_i\,[t]_{\{p \mid i.p \in P\}}) \end{array}$$

Sub-terms

A term s is a proper sub-term of a term t, denoted $s \lhd t$, if:

$s \lhd t \overset{\triangle}{\Leftrightarrow} \exists p \neq \epsilon \cdot s = t|_p$

And a non-proper sub-term if:

$s \unlhd t \overset{\triangle}{\Leftrightarrow} s = t \vee s \lhd t$

We extend the sub-term relation to tuples, literals, clauses and sets of clauses:

$$\begin{array}{lcl} s \lhd \bar{t} & \overset{\triangle}{\Leftrightarrow} & \exists i \mid s \lhd t_i \\ s \lhd u \bowtie v & \overset{\triangle}{\Leftrightarrow} & s \lhd (u, v) \\ s \lhd P(\bar{t}) & \overset{\triangle}{\Leftrightarrow} & s \lhd \bar{t} \\ s \lhd C & \overset{\triangle}{\Leftrightarrow} & \exists l \in C \cdot s \lhd l \end{array}$$

The substitution $[x \mapsto t]$ is defined as

$[x \mapsto t](y) \triangleq$ **if** $(x \equiv y)$ **then** t **else** y

A composition of substitutions, denoted $\sigma_1\sigma_2$, is defined as:

$(\sigma_1\sigma_2)(x) \triangleq \sigma_1(\sigma_2(x))$

We denote the set of substitutions for a signature Σ as $\mathbf{subs}(\Sigma)$.

Orders

For a set S, a strict partial order $\succ \in S^2$ on S is a binary relation on S satisfying:

- Irreflexive: $\forall x \in S \cdot x \not\succ x$
- Transitivity: $\forall x, y, z \in S \cdot x \succ y \wedge y \succ z \Rightarrow x \succ z$
- Asymmetric: $\forall x, y \in S \cdot x \succ y \Rightarrow y \not\succ x$

For any strict partial order $\succ$, the corresponding reflexive closure $\succeq$ is defined as:

$\forall x, y \in S \cdot x \succeq y \Leftrightarrow (x = y \vee x \succ y)$

A strict total order is a strict partial order where

$\forall x, y \in S \cdot x = y \vee x \succ y \vee y \succ x$

and correspondingly a total reflexive closure satisfies:

$\forall x, y \in S \cdot x \succeq y \vee y \succeq x$

A well founded strict partial order $\succ$ on S has no infinite descending chains - formally:

$\neg\exists f : \mathbb{N} \to S \cdot \forall i \in \mathbb{N} \cdot f(i) \succ f(i+1)$

An equivalent definition is that each subset has a minimum:

$\forall S' \subseteq S \cdot S' \neq \emptyset \Rightarrow \exists t \in S' \cdot \forall s \in S' \cdot s \succeq t$

Term Orderings

A simplification ordering $\succ$ on a term algebra $\mathbf{T}(\boldsymbol{\Sigma})$ is a strict partial order on $\mathbf{T}(\boldsymbol{\Sigma})$ that satisfies:

- Compatible with contexts (monotonic):
  $\forall s, t, c \in \mathbf{T}(\boldsymbol{\Sigma}), p \cdot s \succ t \Rightarrow c\,[s]_p \succ c\,[t]_p$

- Stable under substitution:
  $\forall s, t \in \mathbf{T}(\mathbf{\Sigma}), \sigma \in \mathbf{subs}(\Sigma) \cdot s \succ t \Rightarrow s\sigma \succ t\sigma$

- Sub-term compatible:
  $\rhd \subseteq \succ$

A reduction ordering is a well founded simplification ordering.
The lexicographic extension of an ordering on S to an ordering on S^n for $n > 1$ is defined as follows:
$\bar{s} \succ \bar{t} \overset{\triangle}{\Leftrightarrow} \exists i \geq 0 \cdot s_i \succ t_i \wedge (\forall j < i \cdot s_j = t_j)$
For an ordering $\succ$ on S, we use the multiset extension of $\succ$, for a pair of finite multisets on S, m, n:
$m \succ n \overset{\triangle}{\Leftrightarrow} \forall x \in S \cdot m(x) > n(x) \vee \exists y \succ x \cdot m(y) \succ n(y)$.
For a multiset m and an element $x \in S$ we use:
$x \succ m \overset{\triangle}{\Leftrightarrow} \forall y \in S \mid m(y) = 0 \vee x \succ y$
We use a form of *transfinite Knuth Bendix order* ([WZM12],[KMV11]).
We use $\mathcal{O}$ for the set of ordinal numbers and $\bigoplus, \bigotimes$ for natural addition and multiplication on ordinals, respectively.
The Transfinite Knuth Bendix term ordering, $\succ_{\mathrm{tkbo}}$ has two parameters:
A strict partial (potentially total) ordering $\succ$ on the signature F_Σ (sometimes called a *precedence*).
A weight function, $w : F_\Sigma \cup X_\Sigma \to \mathcal{O}$ that satisfies:
$\forall f \in F_\Sigma \cdot w(f) > 0$ and $\forall x \in X_\Sigma \cdot w(x) = 1$.
Unless otherwise noted we will use a w that satisfied
$\forall f \cdot \mathbf{arity}(f) > 0 \Rightarrow \exists m \in \mathbb{N} \cdot w(f) = \omega \cdot m + 1$
$\forall f \cdot \mathbf{arity}(f) = 0 \Rightarrow w(f) = 1$
that is, the function maps all non-constants to 1 and all constants only to direct successors of limit ordinals less than ω^ω. Note that it is not required that the precedence on function symbols agrees with the ordinal order on their weights.
The weight of a term, $w(t)$, is defined recursively as:
$w(f(\bar{s})) \triangleq w(f) + \bigoplus_i w(s_i)$
For literals:
$w(s = t) \triangleq w(s) \oplus w(t)$
$w(P(\bar{t})) \triangleq w(P) \oplus w(\bar{t})$
We define the multiset of variables of a term t, $FV(t)$, recursively as follows:

$$\begin{array}{lcl} FV(x) & \triangleq & [x \mapsto 1] \\ FV(f(\bar{s})) & \triangleq & \bigcup_i FV(s_i) \end{array}$$

The transfinite Knuth Bendix ordering (tkbo) for terms we use is defined as follows:
$s \succ t$ iff $FV(s) \supseteq FV(t)$ and

- $w(s) > w(t)$ or

- $w(s) = w(t), s \equiv f(\bar{s}), t \equiv g(\bar{t})$ and

  $f \succ g$ or

  $f \equiv g$ and $\bar{s} \succ \bar{t}$

In order to extend the definition to literals and clauses, we extend the weight function to predicate symbols, and assume we have a precedence (total order)

$\succ$ also on $P_\sigma \cup \{=\}$ - predicate symbols including the equality symbol.
This extension is total if $\succ$ is.
tkbo for literals is:
$l \succ l'$ iff $FV(l) \supseteq FV(l')$ and

- $w(l) > w(l')$ or
- $w(l) = w(l')$ where $l \equiv [\neg]P(\bar{s}), l' \equiv [\neg]Q(\bar{t})$ and
    - l is negative and $l \equiv \neg l'$ or
    - $P \succ Q$ or
    - $P \equiv Q$ and $\bar{s} \succ \bar{t}$

In the above we used $P(\bar{t})$ to denote also $s = t$ as $= (s, t)$.
We treat each clause as a multiset of literals and then use the multiset extension of $\succ$.
tkbo is total on ground terms.
tkbo also has the desirable property that it is separating for constants - that is, given a constant indexing function $ci(c) : \mathbf{C_\Sigma} \to \mathbb{N}$ we can assign
$\forall c \in \mathbf{const} \cdot w(c) = \omega \cdot \mathbf{ci}(c) + 1$
Where, for any two term s, t, if the maximal constant index of s is greater than that of t then $s \succ t$ regardless of size. This property is important for completeness under scoping (for the ground fragment) as in [KV09], [McM08].

Superposition

We have chosen to use superposition ([BG91]) as the underlying logical calculus. The motivation for this choice is that superposition is a complete semi-decision procedure for FOLE (as opposed to many of the quantifier instantiation schemes used in SMT solvers), and it is known to be efficient in handling equalities in the presence of quantifiers. An additional motivation is that it is possible to define fragments of superposition that, while not complete, have polynomial complexity. Most of our technique is also relevant for some other calculi, as we discuss in the relevant sections. We present here only the ground fragment of superposition, and present the full superposition calculus when we discuss quantification.
The main ideas of superposition can be described as follows: The propositional part of superposition is based on ordered resolution for propositional logic. Roughly, the main idea is to order the literals in each clause (and hence clauses by the multiset extension of the ordering) and for each pair of clauses with opposing maximal literals derive a smaller clause that encodes a case-split on the maximal literal.
Clauses implied by smaller clauses in the set are called redundant. When all case-split clauses have been derived and the empty clause has not been derived, a model of the set of clauses is the set of positive maximal literals of non-redundant clauses.
For example, consider the following clause set (maximal literals are underlined):
$\{C \vee \underline{A}, \underline{\neg A} \vee D\}$
where $A, \neg A$ are maximal in their respective clauses and do not occur in either of C, D.
We assume $D \succ C$ and hence $\neg A \vee D \succ C \vee A$.

The clause $\mathrm{C} \vee \mathrm{D}$ derived by resolution on the maximal literal encodes the case split on A.
The new clause is smaller than both premises by the definition of multiset orderings.
The maximal literal l of the new clause satisfies $\mathrm{A} \succ \mathrm{l}$ and $\mathrm{l} \in \mathrm{D}$ by the ordering.
Hence, $\neg \mathrm{A} \vee \mathrm{D}$ is redundant and the model for the set of clauses is $\{\mathrm{l}, \mathrm{A}\}$.
The reason that we say the clause $\mathrm{C} \vee \mathrm{D}$ encodes the case-split is that if we had another singleton clause l' in the set where $\mathrm{l}' \in \mathrm{C}$ then $\mathrm{C} \vee \mathrm{D}, \mathrm{C} \vee \mathrm{A}$ are redundant and hence the set of positive maximal literals of non-redundant clauses, $\{\mathrm{l}'\}$, is a model.
The complication added by equality is that an equality literal can conflict with a larger literal - for example, $\mathrm{a}=\mathrm{b}$ conflicts with the larger $\mathrm{f(a)} \neq \mathrm{f(b)}$, and the set $\mathrm{a}=\mathrm{b}, \mathrm{b}=\mathrm{c}$ conflicts with the larger $\mathrm{f(a)} \neq \mathrm{f(c)}$.
Superposition uses unfailing Knuth Bendix completion to ensure that in a clause set saturated for superposition the set of maximal positive literals of non-redundant clauses forms a convergent term rewrite system, and that all maximal terms of maximal literals of non-redundant clauses are reduced to their normal form by the convergent term rewrite system defined by the maximal positive literals of all smaller clauses.
Resolution is replaced with term rewriting by maximal positive literals in order to ensure that the set of maximal literals of non-redundant clauses is a model.
The ground superposition calculus is shown in figure 2.2, we discuss the full calculus when we discuss quantification. The full calculus was shown sound and complete in [BG94a].

A simple example of ground superposition (maximal terms are underlined):
For the set $\left\{\mathrm{a}=\underline{\mathrm{b}}, \mathrm{b}=\underline{\mathrm{c}}, \mathrm{f(a)} \neq \underline{\mathrm{f(c)}}\right\}$ with the ordering
$\mathrm{f(c)} \succ \mathrm{f(b)} \succ \mathrm{f(a)} \succ \mathrm{c} \succ \mathrm{b} \succ \mathrm{a}$
superposition allows us to rewrite the term f(c) to a smaller term using the equation $\mathrm{c}=\mathrm{b}$ - to get:
$\mathrm{f(a)} \neq \underline{\mathrm{f(b)}}$
And then, rewriting with the first clause:
$\mathrm{f(a)} \neq \mathrm{f(a)}$
And then
$\square$

We use the notation $\mathrm{S} \vdash_{\mathrm{X}} \mathrm{C}$ to denote that the clause C is derivable in the calculus X from the set of clauses S. When the calculus is clear from the context we use $\mathrm{S} \vdash \mathrm{C}$. In this section we only refer to the ground superposition calculus and hence we shorten $\vdash_{\mathbf{SP_g}}$ to $\vdash$.

Redundancy elimination

The superposition calculus is complete even when redundant clauses are eliminated according to a certain redundancy criterion.
The full superposition calculus was shown complete under the following redundancy criterion (here only a variant for ground clauses):
For a finite set of clauses S and clause D, if $\mathrm{S}, \mathrm{D} \vdash \square$ and for some $\mathrm{S}' \subseteq \mathrm{S}$, $\mathrm{S}' \models \mathrm{D}$ and $\mathrm{D} \succ \mathrm{S}'$ (D is greater than all members of S') then $\mathrm{S} \vdash \square$ - that is, D is redundant.

$$\mathrm{res}_{=} \quad \frac{C \vee \underline{s \neq s}}{C}$$

$$\mathrm{sup}_{=} \quad \frac{C \vee \underline{l} = r \quad \underline{s} = t \vee D}{C \vee s\,[r]_p = t \vee D} \qquad \begin{array}{l} \text{(i)}l = s|_p \\ \text{(ii)}l \succ r, \text{(iii)}l = r \succ C \\ \text{(iv)}s \succ t, \text{(v)}s = t \succ D \\ \text{(vi)}s = t \succ l = r \end{array}$$

$$\mathrm{sup}_{\neq} \quad \frac{C \vee \underline{l} = r \quad \underline{s} \neq t \vee D}{C \vee s\,[r]_p \neq t \vee D} \qquad \begin{array}{l} \text{(i)}l = s|_p \\ \text{(ii)}l \succ r, \text{(iii)}l = r \succ C \\ \text{(iv)}s \succ t, \text{(v)}s = t \succ D \end{array}$$

$$\mathrm{fact} \quad \frac{C \vee l = t \vee \underline{l} = r}{C \vee t \neq r \vee l = r} \qquad \begin{array}{l} \text{(i)}l \succ r, \text{(ii)}r \succ t \\ \text{(iii)}l = r \succ C \end{array}$$

Figure 2.2: The ground superposition calculus $\mathbf{SP_g}$
$\succ$ is a reduction ordering.
The numbered conditions on the right are the side conditions of each inference rule.
The calculus combines ordered resolution with unfailing Knuth Bendix completion.
Equality resolution ($\mathrm{res}_{=}$) allows the elimination of maximal false literals.
Positive superposition ($\mathrm{sup}_{=}$) ensures that the set of maximal positive literals of non-redundant clauses is a convergent rewrite system.
Negative superposition ($\mathrm{sup}_{\neq}$) allows rewriting maximal dis-equalities by the term-rewrite system defined by maximal positive literals, and together with equality resolution is a generalization of ordered resolution for equality.
Equality factoring (fact) is a version of ordered factorting.

unit $$\frac{\neg A \quad \cancel{C \vee A}}{C}$$

taut $$\frac{\cancel{C \vee A \vee \neg A}}{}$$

$\text{taut}_{=}$ $$\frac{\cancel{C \vee s = s}}{}$$

sub $$\frac{C \quad \cancel{C \vee D}}{}$$

simp_{res} $$\frac{\cancel{C \vee A} \quad \cancel{C \vee \neg A}}{C}$$

$\text{simp}_{\text{res2}}$ $$\frac{C \vee A \quad \cancel{C \vee D \vee \neg A}}{C \vee D}$$

$\text{simp}_{=}$ $$\frac{l = r \quad \cancel{C}}{C\,[r]_p} \qquad \begin{array}{l} l = C|_p \\ l \succ r \\ C \succ l = r \end{array}$$

Figure 2.3: simplification rules
$\cancel{C}$ denotes that the premise C is redundant after the addition of the conclusion to the clause-set and hence can be removed.

For the ground superposition calculus we use the simplifying inference rules shown in figure 2.3, all of which satisfy the above criterion. Most of the simplification rules are standard, and $\text{simp}_{\text{res}}, \text{simp}_{\text{res2}}$ are chosen in order to handle the clauses that occur at join points in the program - we discuss these later.

Congruence closure

While superposition can decide the ground equality fragment, some techniques based on congruence closure are more efficient, and specifically efficient join algorithms have been developed for congruence closure.
We use two variants of the transitive reflexive congruence closure calculus **CC** for unit ground (dis)equalities.
The reason we mention two variants is that the first describes the operation of the graph structure we use and the second follows directly the definition of congruence closure, and hence is used for completeness proofs.

The first version **CC** is described in figure 2.4. This calculus has a standard transitivity axiom and a version of equality resolution, but the less standard part is the congruence closure rule. This rule only allows instances of the general congruence closure rule if one of the terms in the conclusion already occurs in some clause. The reason we use this version is that it describes the operation of a congruence closure (CC) graph (in the sense that the graph represents a set of clauses saturated w.r.t. the calculus) - performing congruence closure in a CC graph does not introduce new equivalence classes, although it may introduce new terms.
For a set of unit ground equality clauses S, **CC**(S) is the closure of S w.r.t. **CC** - we use a dedicated data structure to represent **CC**(S), described later.

$$\text{tra}_{\bowtie} \quad \frac{s = u, u \bowtie t}{s \bowtie t}$$

$$\text{res} \quad \frac{s = t, s \neq t}{\square}$$

$$\text{con} \quad \frac{C \quad \overline{s = t}}{f(\bar{s}) = f(\bar{t})} \qquad f(\bar{s}) \lhd C$$

Figure 2.4: The **CC** calculus
We denote by $\overline{s = t}$ for two tuples $\bar{s}, \bar{t}$ of the same arity the set of non-trivial equalities between corresponding elements of the tuples - formally:
$\overline{s = t}$ is the set $\{s_i = t_i \mid i \in 0..|s| - 1 \wedge s_i \not\equiv t_i\}$.
The rule $\text{tra}_{\bowtie}$ is the transitivity rule.
The rule res is similar to equality resolution.
The rule con encodes standard congruence closure, except that the side condition $f(\bar{s}) \lhd C$, where C is any (dis)equality, ensures that no new equivalence classes are introduced in any derivation.

$$\text{con}_I \quad \frac{\bar{s} = \bar{t}}{f(\bar{s}) = f(\bar{t})}$$

Figure 2.5: The $\mathbf{CC_I}$ calculus
The rule con_I follows the definition of congruence closure.

The second version, $\mathbf{CC_I}$, follows directly the definition of congruence closure. It differs only in the congruence closure rule , as described in figure 2.5. We use this version in completeness proofs - a set of (dis)equalities is inconsistent iff it has a refutation in this calculus.

Programs

We assume as input a program in the Boogie ([BCD+05]) intermediate verification language (or a similar IVL) that has been generated as a verification condition (VC) for some source program (and potentially some annotation). We assume a low level Boogie representation that includes a DAG-shaped CFG (loops and method calls are removed using annotations) and the only statements are `assume` and `assert` (a passified program as described in [Lei05]). CFG-nodes in the input represent basic blocks of the Boogie program, and often correspond to basic blocks of the source program.
We modify this input slightly by splitting each CFG-node at each assertion statement `assert e` that occurs in it and replacing the assertion with an outgoing edge to a new leaf node with the statement `assume ¬e`. Now each CFG-node has only `assume` statements and the order of statements within each CFG-node is unimportant - hence each CFG-node can be treated as a set of FOLE for-

```
n₀:
x:=0
y:=10
while (x<10)
   invariant x>=0 && x<=10 && x+y==10
   n₁:
   x:=x+1
   y:=y-1
n₂:
assert x+y<20
```

Figure 2.6: Example for VC encoding - source program

mulae. We convert this set of formulae per CFG-node (including the negated assertion nodes) to CNF form and now each CFG node is associated with a set of clauses.
For example, the source program in figure 2.6 may be converted to the Boogie-style program 2.7 and is further converted to our representation 2.8.
We refer to the IVL program after our transformations as the program and the source language program as the source program.

Structure

The structure of our program is as follows:
A control flow graph - CFG - which is a directed acyclic graph with one root (the program entry point).
The leaf nodes of the CFG are the goal nodes - introduced per assertion. The goal of verification is to show them infeasible.
Each CFG-node n is associated with a set of clauses - $\mathbf{C}_{\mathrm{n}}$.
The clauses at each non-leaf CFG-node represent an encoding of the transition relation of the original program, or some instrumentation used by the verification condition generator to generate the IVL program.
The clauses at each leaf node represent the negation of an assertion generated for the VC, as described above.
We use the following functions to refer to the CFG structure - for a given CFG-node n:
$\mathbf{succ}(\mathrm{n})$,$\mathbf{succ}+(\mathrm{n})$,$\mathbf{succ}*(\mathrm{n})$ are the direct, transitive and reflexive-transitive successors of n, respectively.
Similarly, $\mathbf{ps}(\mathrm{n})$,$\mathbf{ps}+(\mathrm{n})$,$\mathbf{ps}*(\mathrm{n})$ are the corresponding sets for predecessors.

CFG paths

For a program CFG G, a directed path P in the G is a (possibly empty) sequence of nodes s.t. $\forall 0 \leq \mathrm{i} < |\mathrm{P}| \cdot \mathrm{P}_{\mathrm{i}+1} \in \mathbf{succ}(\mathrm{p}_{\mathrm{i}})$.
$|\mathrm{P}|$ is the length of the path.
We use $\mathbf{paths}(\mathrm{G})$ for the set of all directed paths in G, starting at any node and ending at any transitive successor of the node, including one and zero length paths.
For a node n and transitive predecessor $\mathrm{p} \in \mathbf{ps}*(\mathrm{n})$, $\mathbf{paths}(\mathrm{p},\mathrm{n})$ are all the paths

```
n_0:
assume x_0=0
assume y_0=10
if (*)
   n_1:
   //loop head - assume loop condition
   assume x_1<10
   //assume loop invariant
   assume x_1>=0 && x_1<=10 && x_1+y_1==10
   //loop body
   assume x_2=x_1+1
   assume y_2=y_1-1
   //assert loop invariant (on current DSA versions)
   assert x_2>=0 && x_2<=10 && x_2+y_2==10
   //back edge is removed
   assume false
n_2:
//new DSA versions
//assume negated loop condition
assume !x_3<10
//assume loop invariant
assume x_3>=0 && x_3<=10 && x_3+y_3==10
assert x_3+y_3<20
```

Figure 2.7: Example for VC encoding - Boogie program
All variables have been split to DSA versions.
All assignments are converted to `assume` statementes.
The loop return edge has been cut and the body begins with a fresh version for each variable, an `assume` of the invariant and negation of the loop condition, and ends with an `assert` of the loop invariant on the latest DSA versions.
The code after the loop also uses a fresh DSA version of all variables and `assume`s the invariant. (we did not detail a modifies clause for the loop)

```
assume x₀=0
assume y₀=10
if (*)
   n₁:
   assume x₁<10
   assume x₁>=0
   assume x₁<=10
   assume x₁+y₁==10
   assume x₂=x₁+1
   assume y₂=y₁-1
   if (*)
      n₁ₐ: //introduced assertion node
      assume ¬x₂>=0 ∨ ¬x₂<=10 ∨ ¬x₂+y₂==10
      assert false
   assume false
else
   n₂:
   assume !x₃<10
   assume x₃>=0 && x₃<=10 && x₃+y₃==10
   if (*)
      n₂ₐ: //introduced assertion node
      assume ¬x₃+y₃<20
      assert false
```

Figure 2.8: Example for VC encoding - our encoding
Showing that the post-states of both n_{1a}, n_{2a} is infeasible proves the Boogie program and hence the source program.

in G that start at p and end at n, including the case $n = p$.
$\mathbf{paths}(n)$ is short for $\mathbf{paths}(\mathrm{root}, n)$.
For a path P, the set of all clauses in all nodes on the paths is denoted by $\mathbf{C}_P$ - formally:
$\mathbf{C}_P \triangleq \bigcup_{n \in P} \mathbf{C}_n$.
For a set of clauses S we use $S_=$ for the subset S that is unit equalities, disequalities and the empty clause - formally:
$S_= \triangleq (S \cap \{\square\}) \cup \{u \bowtie v \mid u \bowtie v \in S\}$

Semantics

Traces

A trace is a pair (P, M) where P is a path from the root-node to some node P_{end} and M is a model for the signature of the clauses of the program, s.t. for each node on the path, the clauses at the node are satisfied - formally:
$\forall n \in P \cdot M \models \mathbf{C}_n$. M encodes the values of program variables.

Validity

For a given path P from the CFG-root to a node n and a clause C, we say that C holds at n on P - $n \models_P C$ - if C is entailed by $\mathbf{C}_P$ - this means that C holds at the post-state of n for any trace that passes through P - formally:
$n \models_P C \overset{\triangle}{\Leftrightarrow} n = P_{\mathrm{end}} \wedge \mathbf{C}_P \models C$.
A clause C holds at a node n - $n \models C$ - if C holds holds on every trace reaching n - formally:
$n \models C \triangleq \forall P \in \mathbf{paths}(\mathrm{root}, n) \cdot \mathbf{C}_P \models C$
A node n is infeasible iff $n \models \square$
A program is valid iff all its assertion nodes are infeasible - denoted $\models P$ for a program P.

Program transformations

Our verification algorithm works by manipulating the set of clauses at each CFG-node, and sometimes the CFG-structure itself, until there are no assertion nodes left. We describe a set of program transformations that include both the manipulation of the set of clauses at nodes and the CFG-structure.
In order for the verification algorithm to be sound, it must only apply invalidity-preserving transformations to the program, we call these sound transformations - formally:
A transformation T of a program P where is sound iff $\models P \;\Leftarrow\; \models T(P)$.
Conversely, a complete transformations preserve validity - intuitively not losing information - formally:
A transformation of a program P to a program P′ is complete iff
$\models P \;\Rightarrow\; \models T(P)$. For example, removing any CFG-node whose clause-set contains the empty clause is a sound and complete transformation, as is adding a clause to a node's set of clauses that is entailed by this set of clauses.
All of our transformations satisfy a stronger property than soundness and completeness:
A transformation of P to P′ is conservative if it is sound and complete and,

for each CFG-node n that occurs in both P and P′ and each clause C in the vocabulary of P (containing only symbols that occur in P), C holds at n in P iff it holds at n in P′.
The reason this property is interesting is that it allows incremental verification for some program and vocabulary extensions. For example, if a node n has exactly two successors p_1, p_2 and $C \in \mathbf{C}_{p_1} \cap \mathbf{C}_{p_2}$, we can modify $\mathbf{C}_n$ by adding C to it, which is both sound and complete, but not conservative, as $n \models C$ in P′ but not necessarily in P.

We use mostly two kinds of transformations:
Inference - this transformation, for a given node n, replaces the set $S \triangleq \mathbf{C}_n$ with a new clause-set S′ s.t. $S \models S'$ and $S' \models S$. The inference transformation is conservative by definition. In all the cases we consider, S′ is the result of applying some inference rules from some logical calculus to S (including simplification rules that remove redundant clauses).
Propagation - this transformation propagates clauses from the direct predecessors of a node to the node. For example, a CFG-node with one predecessor can add any clause in its predecessor's clause set to its own clause set while being conservative. For nodes with more than one predecessors, propagation can only add a clause to the node's clause-set if it is entailed by the clause-sets of *all* its direct predecessors - we discuss such joins for each logical fragment we consider.

Joins

A CFG-node with more than one direct predecessor is called a join node. In general, it is not sound to add a clause to a join node that does not occur in all predecessors, hence we need some mechanism to propagate information at joins. In order to be able to perform propagation in a sound and complete way for join nodes, we modify the Boogie program as follows:
We ensure that each branch and join in the program is binary - any n-ary branch or join is cascaded to a binary tree of binary branches or joins. This cascading of branches and joins is a conservatives program transformation.

Branch conditions

For each binary branch b, we add a branch condition atom P_b which is a fresh nullary predicate symbol.
For a binary branch node b with successors s_1, s_2 s.t. s_1, s_2 have a common transitive successors, we add the clause P_b to s_1 and $\neg P_b$ to s_2.
Note that the transformation that adds branch conditions is conservative as it only `assumes` new fresh literals.

For a binary join node j with two predecessors p_1, p_2, if, for some path condition atom P_b, $P_b \in \mathbf{C}_{p_1}$ and $\neg P_b \in \mathbf{C}_{p_2}$, and also for some clause C, $C \in \mathbf{C}_{p_1}$, it is sound to add the clause $\neg P_b \vee C$ to $\mathbf{C}_j$. We call the clause $\neg P_b \vee C$ a relativized version of C.
As we show later, inference and propagation can form a complete verification procedure if the above condition holds for all join nodes - that is, for each join node some P_b holds at the first predecessor and $\neg P_b$ at the other.

Well-branching programs

We define a class of programs for which completeness by propagation and inference can be shown - the class of well-branching programs.
Intuitively, a program is well-branching if each join joins exactly one branch.
Formally, a good-join is a binary join node j with direct predecessors p_1, p_2 s.t. the set $\mathbf{ps}{+}(p_1) \cap \mathbf{ps}{+}(p_2)$ has a single maximum m w.r.t. the topological order on the CFG, and each path from the root to j passes through m. We call this maximum the corresponding branch of the join.
A well branching program is a program where all joins are good joins.
It is easy to see that a well branching program always satisfies the above condition for joins - namely, the branch condition of the corresponding branch is always at opposite polarities at the predecessors of a join.
When cascading branches and joins, we try to ensure that the resulting program is well-branching if possible.
In our experience, the VC of programs without exceptional control flow is often well-branching.
If a program is not well-branching, the addition of branch conditions still allows completeness of propagation, but the relativization of clauses is less efficient.

Path conditions

For a well branching program, we define the path condition of a CFG-node, $\mathbf{pc}(n)$, as the set (conjunction) of the branch conditions that hold at the node - formally:
$\mathbf{pc}(n) \triangleq \ \mathbf{if}\ n = \mathrm{root}\ \mathbf{then}\ \emptyset\ \mathbf{else}\ \mathbf{lpc}(n) \cup \bigcap\limits_{p \in \mathbf{ps}(n)} \mathbf{pc}(p)$
where $\mathbf{lpc}(n)$ is the local path condition of n, which is P_b for one successor of a branch b, $\neg P_b$ for the other successor, and $\emptyset$ for all other nodes.

When a clause is propagated through several nodes, in some of which it is relativized, it can collect several branch literals on the way - the relative path condition - $\mathbf{rpc}(p, n)$ - is intuitively the set of branch literals added to the clause when it is relativized on the path from p to n - formally:
$\mathbf{rpc}(p, n) \triangleq \mathbf{pc}(p) \setminus \mathbf{pc}(n)$

The path condition and relative path conditions can be defined also for non well-branching programs, but the definition is more complicated.

Equivalence classes

For a given set of clauses S, we overload the meaning of $S_=$ to denote also the congruence relation defined by the reflexive transitive congruence closure of the unit ground equalities in $S_=$.
The set of equivalence classes of terms of a set of clauses is defined as:
$\mathbf{EC}_S \triangleq \mathbf{terms}(S)/S_=$.
For a CFG-node n we use $\mathbf{EC}_n$ for $\mathbf{EC}_{\mathbf{C}_n}$ - the set of equivalence classes of terms that occur in clauses at n according to the congruence relation defined by unit clauses at n.

For a congruence relation R we use the notation $[t]_R$ to denote the equivalence class of t in R. We drop the subscript when it is clear from the context.

A desirable property of the calculus **CC** is that $|\mathbf{EC}_S| \geq |\mathbf{EC}_{CC(S)}|$. In fact, if C is the result of a derivation with premises in S, and $S' = S \cup \{C\}$, then $|\mathbf{EC}_S| \geq |\mathbf{EC}_{S'}|$, so the set of equivalence classes does not grow from applying derivations in the calculus. This property is immediate from the definition of the calculus, as for each rule, for each sub-term of the conclusion, either the sub-term occurs in the premises, or the conclusion equates it to a term that occurs in the premises.

Atomic ECs

Our algorithm annotates each CFG-node with an approximation of an congruence relation, and the approximations at adjacent CFG-nodes are often similar (agree on many pairs of terms). We use the following concepts to describe the approximation and the relation between similar congruence relations

For a given congruence relation on ground terms we define the set of atomic ECs - **AECs** - which are the smallest sets of terms out of which equivalence classes can be constructed, and the smallest unit that is potentially common with stronger congruence relation.
Given a congruence relation R and a set of terms T, an EC-tuple $\bar{s}$ is a tuple of equivalence classes of T in R - $\bar{s} \in (T/R)^{\mathbf{arity}(f)}$.
The atomic EC $f(\bar{s})$ for an EC-tuple $\bar{s}$ is a set of terms defined as:
$\mathbf{terms}(f(\bar{s})) \triangleq \left\{ f(\bar{t}) \mid \bigwedge_i t_i \in s_i \right\}$
The set of such atomic ECs for a congruence R is $\mathbf{AECs}(R)$.
By the definition of congruence closure, all terms of an AEC are in the same EC of R - formally:
$\forall f(\bar{s}) \in \mathbf{AECs}(R), t \in \mathbf{terms}(f(\bar{s})) \cdot f(\bar{s}) \subseteq [t]_R$.
However, an EC of R may include more than one AEC.
For example, in the congruence defined by $S = \{a = b, f(a) = g(c)\}$, the set of ECs of terms of S ($\mathbf{terms}(S)/R$) are:
$\{\{a, b\}, \{c\}, \{f(a), f(b), g(c)\}\}$ while the set of AECs of terms of S is
$\{a(), b(), c(), f(\{a, b\}), g(\{c\})\}$.
This hints also at another property of AECs - they allow us to share some of the representation of two similar congruence relations (that is, relations that agree on some subset of equalities). In our setting this is most often the case of the sets of possible AECs for the congruence relations that hold at two consecutive CFG-nodes - for example:
For the set S above and the set $S' = S \cup \{c = d\}$, the set of AECs of S' is $\{a(), b(), c(), d(), f(\{a, b\}), g(\{c, d\})\}$.
If S is the set of clauses of a node and S' is the set of clauses of a direct successor (in the CFG) of that node, they can share the common AECs
$a(), b(), c(), f(\{a, b\})$ while they can only share the equivalence class $\{a, b\}$.
For a given congruence R, the sets of terms of AECs are disjoint and each equivalence class is a disjoint union of sets of terms of AECs. Our congruence closure calculus **CC** does not generate any new AECs - the only rule that may introduce a new term (con) does not introduce a new AEC.
We will use the number of AECs as the main space complexity measure as our data structure is based on AECs and, for all the other congruence closure algorithms that we are aware of, the space complexity is at least the number of AECs, possibly more (this is similar to measuring the size of a fully reduced set

of equations as in [GTN04]).

Proofs and models

A proof tree for a logical calculus and a set of axioms is a tree with an instance of an inference rule from the calculus at each node, where the conclusions of the children of each node are the premises of the inference rule instance at the node. The leaves of the tree are axiom nodes. A refutation tree is a proof tree where the conclusion of the root is a contradiction - in CNF form this is usually the empty clause. A proof DAG is similar to a proof tree where the difference is that the conclusion of a node can be used as the premise of more than one parent. A non-redundant proof DAG is a proof DAG where no two nodes share the same conclusion.

For a given logical calculus, set of axioms and theorem, the minimal proof depth is the minimum of depth for all proof-trees (and, equivalently, proof-DAGs) of the theorem from the axioms in the calculus. The depth of a tree or DAG is the length of the longest path from the root to a leaf. The size of a proof-DAG is the number of nodes it contains. The minimal proof size for a given theorem, calculus and set of axioms is the minimal size of proof-DAG for the theorem from the axioms, and similarly for depth.

For each of the automated theorem proving techniques, when a refutation is obtained, a proof-DAG (possibly redundant, depending on the ATP technology) for the refutation can be extracted in the calculus used by the ATP. For example, the original DPLL algorithm produces proofs in tree from, where the minimal proof size can be exponentially larger than an equivalent non-redundant DAG-proof. CDCL produces DAG proofs ([HBPG08]). The lower bound on time complexity of an ATP run on a problem is related to the minimal proof size in the ATP's calculus, but also to the size of the proof search space - preventing the prover from considering proofs with a highly redundant proof-DAG often accelerates proof search.

Background

We discuss here briefly some of the main existing technologies for theorem proving for first order logic with equality (FOLE) and some of its sub-fragments, as FOLE has proven to be sufficiently expressive for many verification tasks and there are practical automatic theorem provers that support it. Many provers also support directly extensions of FOLE that are either not directly axiomatizable in FOLE or for which no efficient axiom system has been found - specifically, rational and integer linear arithmetic and also bit-vectors, arrays and strings.

A notable exception to the expressiveness of FOLE is that it cannot express transitive closure and least fixed points useful for describing heap structures - however, transitive closure logics are beyond the scope of this work, and most are even not semi-decidable (see e.g. [GOR99],[IRR$^+$04]). Verification systems for heap manipulating programs often use either recursive data structures or restricted forms of quantification ([MSS16]) in order to utilize the more mature field of FOLE theorem proving.

Automated theorem provers

Most current theorem provers for FOLE or its fragments are based on some form of resolution, either on the DPLL algorithm ([DLL62]) or on superposition ([BG91]) and ultimately resolution with unification ([Rob65]). Virtually all current theorem provers prove the validity of a formula by showing that its negation is unsatisfiable.

SMT solvers

DPLL based SMT solvers (e.g. [BCD$^+$11],[dMB08]) generally work by trying to build a model for a formula. The model is constructed for a propositional formula by *deciding*, at each step, the propositional value of one ground literal that occurs in the original formula (a literal is an atom or its negation, where the definition of an atom depends on the logic used - for FOLE an atom is an equality on terms). Once a literal has been decided, the formula is simplified by replacing all occurrences of the literal with the selected Boolean value and simplifying the formula accordingly (the formula is often kept in CNF form in which case simplification is simply unit propagation and pure literal elimination) - simplification may produce further unit clauses that are also propagated. If a contradiction is reached at any point (for CNF - a contradiction is the empty clause), the algorithm backtracks to the last decision and *learns* the negation of the decision - adding the negated literal to the set of clauses and simplifying clauses accordingly. If learning the negated literal also produces a contradiction the previous decision is reversed. If the negated top decision produces a conflict then the formula is not satisfiable. If all literals have been decided and no contradiction has been found then the set of decisions and simplified clauses define a model for the formula - at any point in the algorithm the set of decided and derived literals is called the candidate model.

Conflict driven clause learning: A major improvement in the DPLL algorithm which has made it practical is conflict driven clause learning (CDCL) ([SS99]). CDCL improves upon DPLL by maintaining a graph that keeps track of the implications between literals that have been decided, and hence allows the algorithm to *learn* a new clause when a conflict is reached that includes only the negation of the decisions that have caused the conflict. The algorithm backtracks to the first decision involved in the conflict, rather than to the last decision as in DPLL. CDCL was shown to be exponentially more efficient than the original DPLL ([HBPG08]).

An important property of the propositional DPLL algorithm is that the space requirement is at most proportional to the number of literals in the input formula (the sequence of decided and derived literals). On the other hand, the algorithm can progress - learn new information - only when a conflict is reached. For propositional logic and GFOLE the depth of the search is bounded by the size of the input, but when quantifiers are involved the depth of the search is not bounded.

Nelson and Oppen have added the efficient handling of equality to the ground DPLL procedure above and the handling of some ground theories for which satisfiability is decidable ([NO79]). The algorithm handles equality by using a congruence closure (CC) data structure which allows deciding and backtracking equality literals. The congruence closure data structure detects contradictions by maintaining dis-equality edges and is, at any stage of the algorithm, the candidate model for equality. Ground theories are decided by sending each theory decision procedure the subset of the candidate model that is relevant, with the non-theory symbols abstracted away and, if no contradiction is found, all equality literals between terms in the original formula are decided until either a conflict is found or a full model is found which is agreed by all theories. Several improvements have been proposed to this algorithm, notably allowing a theory decision procedure, when it reports a set of literals as satisfiable, to also report the set of (dis)equalities that are implied by the set of literals (over the terms that occur in the literals), in a similar fashion to unit propagation.

Shostak theories: An alternative method for combining ground propositional and equality reasoning with a theory was introduced by Shostak ([Sho84]). The Shostak method requires, instead of a just a satisfiability decision procedure, a *canonizer* that rewrites any theory term to a normal form (where two theory terms are equal iff they have the same normal form) and a *solver* that decides satisfiability for a set of literals in normal form, and outputs a *solved form* of the input literals, which intuitively means that it expresses one of the input variables as a function of the rest of the variables (variables for the theory are the top non-theory ground terms that occur in the theory literals). The Shostak method is used in the theorem prover Alt-Ergo ([CCKL08]).

DPLL for first order logic: The original DPLL algorithm applied to quantified FOL formula by repeatedly generating ground instances of the quantified clauses and applying the DPLL algorithm incrementally to the set of ground instances - this extension to quantified FOL has not seen much practical use. Current SMT solvers usually use some heuristics to create ground instances of quantified clauses (e.g. [RTG$^+$13],[GdM09]). One common heuristic uses patterns (also called triggers), where, for each quantified clause, a set of terms over the free variables that occur in the clause is selected - the *triggers*, and a ground instance of the clause is generated only if the ground instance of each trigger is E-unifiable (unifiable modulo the equality theory encoded in the CC data structure) with a term represented by the CC data structure.
Another approach for handling quantifier in SMT solvers is model-based quantifier instantiation (MBQI) ([GdM09]) which determines a set of relevant ground

```
n0: if (a>0)
   ...
   if (b=2*a+1)
      ....
   else
      ....
else
   n2: b:=a-1
n3: assert (a+b)^2>0
```

Figure 2.9: Example for lazy CNF conversion

instances (potentially infinite) for each quantified clause and incrementally tries to find a model for a subset of the relevant instances. If any such subset is unsatisfiable then the original formula is unsatisfiable, otherwise a model is found which can be used to guide further instantiation. Current SMT solvers tend to be very efficient on problems involving propositional and ground first order logic, and the ground linear integer and rational theories. Problems including quantifiers are sometimes not handled as efficiently by SMT solvers, and especially the performance of the prover is very sensitive to small changes in the input.

Incremental or Lazy CNF conversion: Another technique that contributes significantly to the success of SMT solvers is incremental or lazy CNF conversion ([BDS02a],[FJOS03]). The basic algorithms of DPLL and its CDCL variant operate on a formula in CNF form (a set (conjunction) of clauses (disjunctions of literals)). In order to convert an arbitrary propositional logic (PL) formula including arbitrary logical connectives in polynomial time and space, it is sometimes necessary to introduce new literals that represent the (propositional) value of sub-formulae ([BEL$^+$01]), and add clauses to the result that encode the equivalence between the newly introduced literal and the sub-formula - for example:
$A \vee (B \wedge C)$
is converted to
$(A \vee L_1) \wedge (L_1 \Leftrightarrow (B \wedge C))$
which is then converted to CNF as
$\{A \vee L1, L_1 \vee \neg B \vee \neg C, \neg L_1 \vee B, \neg L_1 \vee C\}$.
L_1 is the newly introduced literal and the formula $L_1 \Leftrightarrow (B \wedge C)$ is the definition of L_1.

The original DPLL algorithm converts the negation of a formula to a CNF set of clauses and then performs steps of deciding a literal, simplifying and backtracking when a conflict is detected.
Roughly, the idea of lazy CNF conversion in DPLL is to begin with a set of clauses that includes only the negation of the literal that represents the entire formula, and only add definition clauses to the clause-set when the relevant literal is decided with the relevant polarity. The advantage of this technique is that it prevents interference between clauses that may not be on the same decision branch. Consider, for example, the program in figure 2.9.

Depending on the encoding of the VC and the order of decisions by the SMT

solver, eager CNF conversion may cause the prover to produce the following candidate model (showing only theory relevant literals):
$\{b = 2a + 1, b = a - 1, \neg a > 0, (a + b)^2 > 0$.
The literals $b = 2a + 1, b = a - 1$ should not be part of the same candidate model as they occur on different branches - the choice of whether $b = 2a + 1$ or not is irrelevant for the proof of the outer else path.

For a large program with many branches such redundancy can make a significant performance difference (as reported in [BDS02a],[DNS05]). Even more significant is the fact that when pattern-based quantified instantiation is used, and the above encoding is used for the VC, lazy CNF conversion ensures that quantifiers are instantiated only with sets of terms that occur on some path in the program, and specifically not on opposing branches.

Superposition

Another approach for equational theorem proving is superposition, based ultimately on resolution ([Rob65]) and unfailing Knuth-Bendix completion ([BG94b]). Ordered resolution (for propositional logic) can be seen also as a search with candidate models: given a formula in CNF (a conjunction of disjunctions of literals), we order the clauses according to some total order on literals and its extension for clauses.
A clause is redundant if it is entailed by smaller clauses. The candidate model is the set of maximal literals of non-redundant clauses where the maximal literal is positive.
The candidate is not a model iff some non-redundant clause has a maximal negative literal that occurs positively in the model, in which case we resolve the two clauses on their maximal literals and the conclusion is a smaller (at least from the bigger premise) clause that encodes the case-split on the maximal literal for the two clauses (it entails that one of the premises holds, regardless of the value of the maximal literal). If the empty clause is derived the set is unsatisfiable, otherwise the process terminates when no new resolutions can be performed.
For GFOLE and FOLE equalities and dis-equalities are used instead of propositional literals. The difficulty is that a set of maximal equalities can be inconsistent with a different dis-equality (e.g. $a = b$ is inconsistent with $f(a) \neq f(b)$). To solve this difficulty, resolution is replaced with a restriction of ordered paramodulation (rewriting the maximal term of one clause with the maximal equality of another clause) and additional inference rules, ensuring that the candidate model forms a convergent rewrite system and each maximal term in a maximal literal is rewritten to normal form. For example, the terms $\mathrm{C} \vee \underline{\mathrm{l}} = \mathrm{r}, \underline{\mathrm{f(l)}} = \mathrm{t} \vee \mathrm{D}$ where the maximal terms are underlined, produce $\mathrm{C} \vee \mathrm{D} \vee \mathrm{f(r)} = \mathrm{t}$ with a superposition inference - this clause encodes a case-split, either C holds or $\mathrm{l} = \mathrm{r}$ holds and then we can rewrite the right premise to $\mathrm{D} \vee \mathrm{f(r)} = \mathrm{t}$. For quantified clauses, unification is used to restrict the number of possible inferences and the candidate model is constructed from the (often countable) set of ground instances of clauses.
Saturation based provers such as these based on superposition often work by maintaining two sets of clauses, the first set is clauses that are inter-saturated (so all valid inferences with all premises in this set have been performed) and the second set of not yet saturated clauses.

The strength of superposition based provers is in handling quantifiers while the main weaknesses are handling large Boolean problems and handling theories for which no efficient axiomatization has been found. A key difference between saturation based provers (that derive new clauses from existing clauses, such as superposition) and DPLL based provers is that DPLL based provers must find a conflict in order to derive new (learned) clauses, and hence progress, and the complexity of finding one conflict is not bounded when quantifiers are involved, while saturation based provers can generate new clauses (and hence progress) at a known complexity bound (e.g. perform all valid inferences between a not-yet saturated clause and the saturated set and add the clause to the set - with complexity dependent on the size of the saturated set).

Instantiation based theorem proving

Another approach for theorem proving in instantiation based ([Kor08], with equality [KS10]). The approach uses unification, as in resolution based provers, in order to find substitutions, but, instead of performing resolution, the instantiated versions of the premises are added to the clause set - for example:
For the premises $Q(x) \vee P(x,b)$, $\neg P(a,y) \vee R(y)$, resolution produces $\{Q(a) \vee R(b)\}$ while instantiation produces $\{Q(f(a)) \vee P(a,b), \neg P(a,b) \vee R(b)\}$. The attraction of the approach is that it avoids producing large clauses as in superposition, but it requires a different solution for the propositional part of the proof.
After some steps of instantiation, the prover temporarily substitutes a new constant (not part of the vocabulary), for all free variables in all clauses and uses a SAT solver (or SMT solver with equality) to check for satisfiability. Instantiation for FOL with equality is implemented in the iProver theorem prover ([KS10]).

Stålmarck's method

A somewhat different approach for propositional satisfiability, that also handles ground equalities, is Stålmarck's method ([SS98]). This method has seen somewhat less research as it is protected by patent laws.
The interesting aspect of the method, in our context, is the approach for handling case-splits. The algorithm maintains the formula for which satisfiability is to be checked in a specific form, somewhat similar to lazy CNF, and saturates the formula w.r.t. a set of simplification rules (a somewhat similar concept to unit propagation in DPLL). The main difference in formula representation and simplification is that literal equivalence can be represented natively - for example, the simplification rules can learn that $A \Leftrightarrow B$ without determining the actual truth values for A and B. When simplification is insufficient for refutation, the algorithm performs a case split on some literal or atomic formula, and simplifies instances of the formula for both polarities of the literals split on. If neither branch finds a contradiction, a form of join is performed which adds any fact learned separately on both branches to the pre-split formula, the two formula instances used for the case-split branches are discarded. The algorithm proceeds by case-splitting on all relevant atomic formulae - which includes equality between two literals and between a literal and a Boolean constant. If no contradiction is found, the algorithm proceeds by performing case splits of depth 2 - for each relevant atomic formula f, first choose a value V, simplify an

instance of the formula, and then recursively perform the depth-1 case-splitting, and similarly for $f = \neg V$. We find this method interesting because it shares with DPLL the property that the space needed is approximately proportional to the depth of the decision tree, but, as opposed to DPLL and CDCL, it ensures progress regardless of finding a conflict, and has a well defined hierarchy of fragments (by the case-split depth) that can be applied incrementally. An extension for first order logic is given in [Bjö09]. We have not tried to adapt this technique to our setting, but the kinds of joins used, both in the PL and FOLE versions, are related to the joins we perform in join nodes in the CFG.

Model evolution

An algorithm for theorem proving related to DPLL is that of *model evolution* ([BPT12]) implemented in the Darwin theorem prover ([Fuc04]). Model evolution behaves like DPLL but decides also non-ground literals where, essentially, unification is used to select substituted versions of quantified literals to decide upon. The model evolution calculus has the advantage that quantifiers are handled directly, and unification is used for directing instantiation. However, as with the methods for handling quantifiers in SMT solvers, a conflict must be reached in order to learn new information, and the search depth is not bounded.

Chapter 3

Unit ground first order logic with equality

In this chapter we present an incremental, on-demand algorithm and data-structure for propagating equality information and verifying programs in the fragment of unit ground first order logic with equality - GFOLE. The algorithm is not intended as a stand alone verifier, but rather as a basis for the verification algorithms for stronger fragments. The algorithm maintains a data-structure at each CFG-node that encodes some equality information and is updated on-demand with information from predecessors. Other fragments can query the data structure on whether a ground equality holds at a given CFG-node, and can also add a derived equalities to the data-structure at a given CFG-node. The algorithm ensures that enough equality information is propagated to answer the queries that are provable in the fragment, including performing joins of congruence relations at join nodes, and the data-structure is updated accordingly. Equality propagation is done incrementally. When another fragment derives an equality at some CFG-node, the algorithm ensures that this equality is propagated wherever it may affect the result of a previously queried equality, under some limitations.

Our data-structure at each CFG-node is similar to an E-graphs but ensures that the represented congruence relation is fully reduced. In addition to a term-graph, the data-structure includes a relation between the graph of a CFG-node and those of its predecessors, used to communicate equality information.

We start by presenting our graph data structure. We continue by discussing the limitations of using EC-graphs (or sets of equalities) to represent post-conditions in the CFG. Next we present the problem of lazy information propagation and present our algorithm for equality propagation for CFGs without joins. In the next section we present our join algorithm as an extension to the join-less algorithm. Lastly we discuss related work.

3.1 Congruence closure graphs

We define a congruence closure graph data structure that forms the basis of our verification algorithm. Each node in the graph represents an equivalence class of terms. The data structure presented here is a variant on the common representation for congruence closure, dating back as far as [DST80] and earlier, but instantiated per CFG-node, and with edges added between instances to communicate equalities. We show our basic version here in order to fix the vocabulary and as a basis for the full version that we show in section 3.3.

Our data structure represents a set of terms and a set of ground equalities. The set of represented terms includes at least all the sub-terms that occur in the ground equalities. The set of terms is also sub-term closed and also, for each term, the entire equivalence class (EC) of the term w.r.t. the congruence defined by the set equalities, is represented.

3.1.1 Data Structure

A basic EC-graph is composed of a set of nodes that represent equivalence classes of terms. We call each such node a GT (ground term equivalence class). Each GT is composed of a set of ground function application equivalence classes - GFAs - where each GFA is composed of a function and a tuple of GTs. A GFA represents an atomic equivalence class for the EC-graph.
The EC-graph includes a set of constants (the roots of the graph) and a map from each GT to to its super-terms - the GTs that contain a GFA with a tuple that contains the GT. The basic data structure is summarized in figure 3.1. The main difference between our representation and the more common representation is that we have an object (GT) that represents an entire EC, while many congruence closure algorithms only have GFAs and use equality edges between GFAs to specify equality, selecting one GFA to be a representative for each EC. The main reason we have an object per EC is that it makes our data structure independent of its construction order, and hence we can share more parts of an EC-graph between CFG-nodes and join operations are easier, in addition, performance is less sensitive to the order in which graph operations are performed.
The set of terms represented by each node in the graph is defined recursively as follows:

$$\mathbf{terms}(\mathrm{GT}(\mathrm{gfas})) \triangleq \bigcup_{\mathrm{gfa} \in \mathrm{gfas}} \mathbf{terms}(\mathrm{gfa})$$

$$\mathbf{terms}(\mathrm{GFA}(\mathrm{f}, \mathrm{tt})) \triangleq \left\{ \mathrm{f}(\overline{\mathrm{t}}) \mid \bigwedge_{\mathrm{i}} \mathrm{t}_{\mathrm{i}} \in \mathbf{terms}(\mathrm{tt}[\mathrm{i}]) \right\}$$

The `ECGraph` object includes a map of constants (the roots of the terms) which maps a constant function symbol to the GT in which it occurs, and a map from each GT to its super-terms.
The set of GTs in a graph is defined using the super-terms map, with the roots being the constants:

$$\mathrm{GTs}(\mathrm{g}) \triangleq \bigcup_{\mathrm{c} \in \mathrm{g.constants.values}} \mathrm{GTs}(\mathrm{g}, \mathrm{c})$$

$$\mathrm{GTs}(\mathrm{g}, \mathrm{t}) \triangleq \{\mathrm{t}\} \cup \bigcup_{\mathrm{s} \in \mathrm{g.superTerms}[\mathrm{t}]} \mathrm{GTs}(\mathrm{g}, \mathrm{s})$$

```
class GT(gfas : Set[GFA])

class GFA(f : Function,tt : Tuple[GT])

class ECGraph
   method makeTerm(f:Function,tt:Tuple[GT]) : GT
   method assumeEqual(gt0,gt1:GT)

   constants:Map[Function,GT]
   superTerms:Map[GT,Set[GT]]
```

Figure 3.1: Basic EC graph data structure
GTs and GFAs are defined in Scala style to simplify the notation.
A GT represents an EC of ground terms.
A GFA represents an AEC (atomic EC) of ground terms.

Invariant

The invariant of the `ECGraph` g is composed of several parts:
The constants map maps a constant function to its term:
$\forall t \in \mathrm{GTs}(g), f \cdot \mathrm{GFA}(f, ()) \in t.\mathrm{gfas} \Leftrightarrow g.\mathrm{constants}[f] = t$
The super-term map is closed:
$\mathbf{ran}(g.\mathrm{constants}) \subseteq \mathbf{dom}(g.\mathrm{superTerms})$
$\forall t \in \mathbf{dom}(g.\mathrm{superTerms}) \cdot g.\mathrm{superTerms}[t] \subseteq \mathbf{dom}(g.\mathrm{superTerms})$
The graph is congruence and transitive closed:
$\forall s, t \in \mathrm{GTs}(g) \cdot \mathbf{terms}(s) \cap \mathbf{terms}(t) = \emptyset$

Operations

The basic EC graph supports two operations:

`makeTerm(f:Function,tt:Tuple[GT]) : GT` - this operation receives a tuple of graph nodes ($tt \subseteq \mathrm{GTs}(g)$) and a function with the appropriate arity and returns the GT in the graph that includes the GFA $\mathrm{GFA}(f, tt)$. If there is no such GT, a singleton GT - $\mathrm{GT}(\mathrm{SET}(\mathrm{GFA}(f, tt)))$ - is added to graph, updating the data structure accordingly.

`assumeEqual(gt0,gt1:GT)` - this operation merges the GTs gt0 and gt1 and performs congruence closure until the ECGraph invariant holds. Note that the classic E-graph used by SMT solvers uses a union-find data structure (as in e.g. [NO77]) which is generally more efficient, however, in our context our version has some advantages, as described below.

While the operation of `makeTerm` is rather straightforward for the basic EC-Graph, `assumeEqual` requires some more attention, as it forms the basis of all graph updates, hence we give a pseudo-code in figures 3.2,3.3.
The main operation of the `assumeEqual` method maintains a map (`mergeMap`) of merge edges which is maintained as an acyclic directed graph with out-degree one. Each GT is merged with the GT at the end of its chain in `mergeMap`. When

```
method assumeEqual(gt0,gt1:GT)
   requires gt0,gt1 ∈ GTs(this)

   if (gt0=gt1)
      return;

   var mergeQ := new Queue[GT]
   var mergeMap := new Map[GT,GT]
   enqueueMerge(gt0,gt1)

   while (!mergeQ.isEmpty)
      mergeOne(mergeQ.dequeue)

method enqueueMerge(gt0,gt1:GT)
   var sgt = transitiveMerge(gt0)
   var tgt = transitiveMerge(gt1)
   if (sgt!=tgt)
      mergeMap[sgt]:=tgt

method transitiveMerge(gt)
   while gt∈mergeMap.keys
      gt := mergeMap[gt]
```

Figure 3.2: Basic EC graph congruence closure code
The algorithm maintains a queue `mergeQ` of GTs that need to be merged,
and a map `mergeMap` that encodes which GT is merged to which GT, it is maintained essentially as in the union-find algorithm.
The `transitiveMerge` method traverses the map and returns the last element of the chain that starts at its argument.
The `mergeOne` method merges two GTs, updates the relevant maps and enqueues any equalities implied by congruence closure - it is detailed in figure 3.3.

```
method mergeOne(gt:GT)
   var target := transitiveMerge(gt)

   //merge GFAs
   target.gfas.add(gt.gfas)
   foreach (gfa in gt.gfas)
      if (gfa.tt==())
         constants[gfa.f] := target
      else foreach (sgt in gfa.tt)
         superTerms[sgt].remove(gt)
         superTerms[sgt].add(target)

   //update super terms
   foreach (sgt ∈ superTerms[gt])
      foreach (gfa ∈ sgt.gfas)
         if (gt ∈ gfa.tt)
            sgt.gfas.remove(gfa)
            var newGFA := gfa[gt↦target]
            if (∃gt2∈superTerms[target]·newGFA∈gt2.gfas)
               enqueueMerge(target,gt2)
            else
               superTerms[target].add(sgt)
               sgt.gfas.add(gfa)
```

Figure 3.3: Basic EC graph single merge step
The method `mergeOne` merges one pair of GTs and updates the maps and super-terms accordingly.
If a super-term update causes the super-term's EC to overlap another EC, they are enqueued for a merge.
After the `mergeOne` method, `gt` is effectively removed from the graph.
The syntax `gfa[gt↦target]` means replacing each occurrence if `gt` with `target`.

merging two GTs, the sets of GFAs of the GTs are merged, and all the GFAs of the super-terms of the source GT are updated to point to the target GT. If any such updated GFA already occurs in the graph in another GT, we enqueue a merge of the other GT with the target.

Properties

The main property of the basic EC-Graph is that each term added to it (using `makeTerm`) is represented in it, and that it represents a congruence that is exactly the congruence defined by the set of equalities added to it (using `assumeEqual`).
Equations represented by the graph:
There is not necessarily a unique minimal set of equations that represents the congruence represented in the graph - for any congruence relation and any set of equalities that define the congruence, adding the equalities to the graph in any order will result in the same graph - for example, adding $a = b$ and then $f(b) = f(c)$ will give the exact same graph as adding $f(a) = f(c)$ and then $a=b$.
In order to define a canonical set of equations represented in the graph, we assume an arbitrary total order on terms, where we only require that deeper terms are larger. Using this order we can define the set of equations of a graph by defining a representative for each GT:
$\mathrm{rep}(\mathrm{GT}(s)) \triangleq \min_{u \in s} \mathrm{rep}(u)$
$\mathrm{rep}(\mathrm{GFA}(f, tt)) \triangleq f(\mathrm{rep}(\overline{tt}))$
$\mathrm{rep}(\overline{t})_i \triangleq \mathrm{rep}(t_i)$
We also define the representative GFA of a GT by using the order on GFA representatives:
For a pair of GFAs $f(\overline{t}),g(\overline{s})$, $f(\overline{t}) < g(\overline{s})$ iff $\mathbf{rep}(f(\overline{t})) < \mathbf{rep}(g(\overline{s}))$.
Using this ordering we define:
$\mathrm{repgfa}(\mathrm{GT}(s)) \triangleq \min_{u \in s} u$
Because deeper terms are larger, the representatives are well defined.
The canonical set of equations is defined first for one GT:
$\mathbf{eqs}(u) \triangleq \{s = t \mid t = \mathrm{rep}(u) \land \exists v \in u.\mathrm{gfas} \cdot u \neq v \land s = \mathrm{rep}(v)\}$
And for the EC-graph:
$\mathbf{eqs}(g) \triangleq \bigcup\limits_{u \in g} \mathbf{eqs}(u)$

Lazy representation for a congruence

As we propagate equality information lazily in the CFG, we use EC-Graphs to represent an approximation of a congruence - the idea is that, for each pair of terms represented in the graph, the terms are equal in the congruence iff they are represented by the same GT, while for a term not represented in the graph, there may be more equalities in the congruence we are approximating - for example:
In the graph constructed from the following sequence of operations:
```
a:=makeTerm(a,()); b:=makeTerm(b,());
c:=makeTerm(c,()); assumeEqual(a,b)
```
We know that $a = b, f(a) = f(b)$ hold in the congruence we are approximating and we also know that $a = c, b = c$ do not hold, but we do not know whether $f(a) = f(c)$ or $g(b) = g(c)$ or $a = f(a)$ hold.

Given a set of ground equalities and a pair of terms, the EC-graph can be used

to decide whether the set of equalities implies equality between the two terms. This is done using the following auxiliary methods:
This method `makeTerm(t:Term) : GT` simply constructs the term t in the graph bottom up, adding necessary GTs:

```
makeTerm(f($\bar{s}$):Term) : GT
   return makeTerm(f,[$i \mapsto$makeTerm(s$_i$)])
```

The expression $[i \mapsto \text{makeTerm}(s_i)]$ denotes constructing a tuple where the i-th element is the result of the call `makeTerm(s`$_i$`)` (the order of the calls is unimportant in our data structure).
The method `assume(s=t : Equality)` is used to assume equalities on terms rather than on GTs - defined as:

```
assume(s=t:Equality)
   assumeEqual(makeTerm(s),makeTerm(t))
```

Eager use of axioms: A simple way to check whether a set of axioms entails an equality $s = t$ is to use `s0:=makeTerm(s);t0:=makeTerm(t)` to add the two terms to an empty graph, and then use `assume(u=v)` for each equality $u = v$ in the set of axioms, and return `makeTerm(s)==makeTerm(t)`.
Note that the result of makeTerm(s) might change after an `assumeEqual` - if we keetp the `mergeMap` in the data structure rather than just for each `merge` call, we can avoid the second calls for `makeTerm(s),makeTerm(t)` by returning `transitiveMerge(s0)==transitiveMerge(t0)`.

Lazy use of axioms: We can avoid adding all the axioms by only adding axioms where the equated terms are represented in the graph - the algorithm is shown in figure 3.4.
The algorithm repeatedly adds axioms to the graph for all terms represented in the graph, until none are left.
Note that `g.assume(l=r)` might add represented terms to the graph even if l, r are already represented in the graph - for example, if $\mathbf{terms}(g) = \{a, b, f(a)\}$ and we call `g.assume(a=b)`, in the resulting graph g' $\mathbf{terms}(g') = \{a, b, f(a), f(b)\}$.

Correctness: We argue here the algorithm above is correct - that is, that the algorithm returns true iff axioms $\models s = t$.
We use the notation $[t]_g$ for the EC-node (GT) that represents the term t in the EC-graph g, if $t \in \mathbf{terms}(g)$.
We extend this notation to tuples - e.g. $[\bar{s}]_g$ is the tuple of GTs where the i-th element is $[s_i]_g$ and also to GFAs - $[f([\bar{s}]_g)]$ is the GT that contains the GFA $f([\bar{s}]_g)]$ if there is such a GFA in g.
We assume the EC-graph is implemented correctly, so that the graph represents the congruence defined by the set S and the set of terms represented at the graph are all the terms that were added to the graph and their ECs under the congruence in the graph. Specifically, we assume that if the set S of equalities was `assumed` in g, and for the terms s,t, $s, t \in \mathbf{terms}(g)$, then $[s]_g = [t]_g$.

The argument for soundness - that is, **true** is returned only if axioms $\models s = t$ - comes directly from the soundness of the EC-graph operations - we only add to the graph axioms from `axioms`.

The argument for completeness - that is, if axioms $\models s = t$ then **true** is returned - can be shown by strong induction on the depth of a derivation tree

```
checkEntailment(s,t : Term,axioms : Set[Clause]) : Boolean
   var S := new Set[Clause]()
   var g := new EC graph
   g.makeTerm(s)
   g.makeTerm(t)
   do
      variant: |axioms \ S|
      invariant eqs(g) ⇔ S

      var ns: = {l = r ∈ axioms \ S | l ∈ terms(g)}
      foreach l = r ∈ ns//Axiom closure
         g.assume(l=r)
   until ns=∅
   return g.makeTerm(s)==g.makeTerm(t)
```

Figure 3.4: Graph based lazy entailment checking
The algorithm simply selects all equations on terms the graph represents and assumes them in the graph, until no such axioms remain.
We discuss later how to select the new axioms to add (build ns) efficiently.

of $\mathrm{s} = \mathrm{t}$ from axioms in the calculus $\mathbf{CC_I}$.
As $\mathbf{CC_I}$ is complete (it includes all instances of axioms that define a congruence relation, as per the definition), if axioms $\models \mathrm{s} = \mathrm{t}$ then axioms $\vdash_{\mathbf{CC_I}} \mathrm{s} = \mathrm{t}$.
We proceed by strong induction on the derivation tree in $\mathbf{CC_I}$ of $\mathrm{s} = \mathrm{t}$ from axioms.
The induction hypothesis is that for a derivation depth k, if $\mathrm{s}' \in \mathbf{terms}(\mathrm{g})$ and $\mathrm{s}' = \mathrm{t}'$ is derivable from axioms in $\mathbf{CC_I}$ in a derivation of at most k steps, then $\mathrm{t}' \in \mathbf{terms}(\mathrm{g})$ and $[\mathrm{s}']_\mathrm{g} = [\mathrm{t}']_\mathrm{g}$.
We perform case analysis on the root of the derivation tree:
Axioms: If $\mathrm{s} \in \mathbf{terms}(\mathrm{g})$ then the assignment to ns adds the axiom $\mathrm{s} = \mathrm{t}$ to ns and hence to S, hence S$\models \mathrm{s} = \mathrm{t}$ and so $[\mathrm{s}]_\mathrm{g} = [\mathrm{t}]_\mathrm{g}$.
Reflexivity: Immediate - $[\mathrm{s}]_\mathrm{g} = [\mathrm{s}]_\mathrm{g}$.
Transitivity: By definition, there is some u s.t. axioms $\models \mathrm{s} = \mathrm{u}, \mathrm{u} = \mathrm{t}$ and their derivation is of depth at-most k-1. By i.h. $\mathrm{u} \in \mathbf{terms}(\mathrm{g})$ and hence again by i.h. $[\mathrm{s}]_\mathrm{g} = [\mathrm{u}]_\mathrm{g} = [\mathrm{t}]_\mathrm{g}$
Congruence Closure: If $\mathrm{s} = \mathrm{f}(\overline{\mathrm{u}}), \mathrm{t} = \mathrm{f}(\overline{\mathrm{v}})$, then for each i we have a derivation of $\mathrm{u_i} = \mathrm{v_i}$ of depth at most k-1. Because the EC-graph is sub-term closed, for each i, $\mathrm{u_i} \in \mathbf{terms}(\mathrm{g})$ and hence by i.h. also $\mathrm{v_i} \in \mathbf{terms}(\mathrm{g})$ and $[\mathrm{u_i}]_\mathrm{g} = [\mathrm{v_i}]_\mathrm{g}$.
As the EC-graph is congruence closed and contains, for a represented terms, all terms in its EC under the congruence defined by the graph, we get that $\mathrm{f}(\overline{\mathrm{v}}) \in \mathbf{terms}([\mathrm{f}(\overline{\mathrm{u}})]_\mathrm{g}) \subseteq \mathbf{terms}(\mathrm{g})$ and hence $[\mathrm{f}(\overline{\mathrm{v}})]_\mathrm{g} = [\mathrm{f}(\overline{\mathrm{u}})]_\mathrm{g}$
■

3.1.2 Notation

We use a graphical notation for presenting some examples with EC-graphs.
The notation diverges somewhat from the presentation of the algorithm as we sometimes represent tuples of equivalence classes in the notation - this is done to improve clarity. A legend for the notation is given in figure 3.5.
In figures 3.6 we show some graphs with binary tuples. For an EC-graph g, and

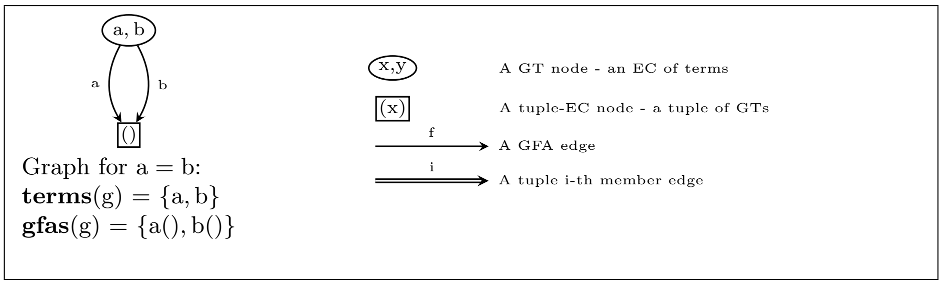
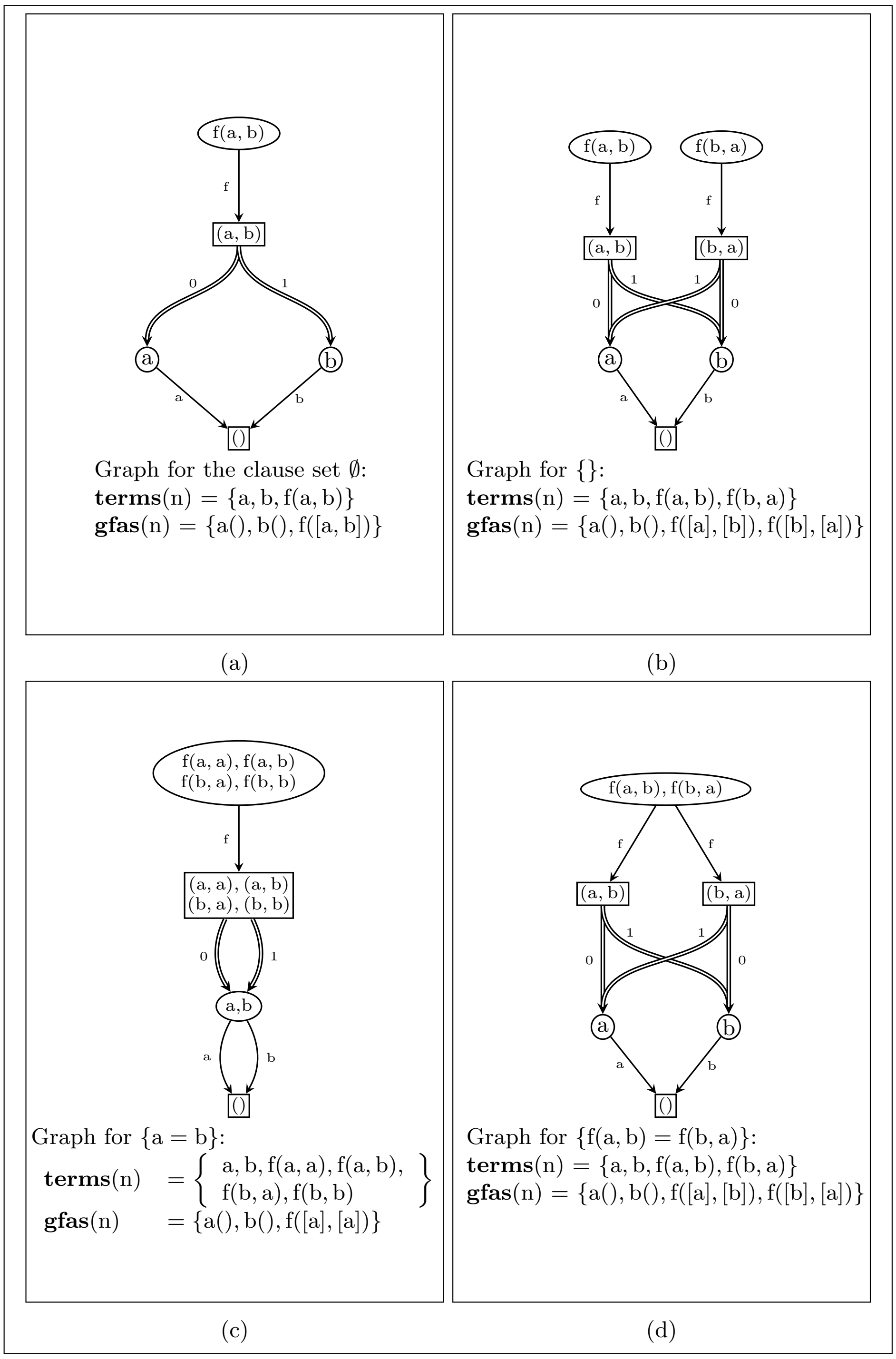


Figure 3.5: Equivalence class graph notation
GTs are represented using circles and GT tuples are represented using rectangles
Inside the circles and rectangles we enumerate some of the members (terms) of each EC for clarity, but as some ECs are infinite, this list is not always complete.
Single arrows represent function application edges while indexed double arrows represent membership in the i-th position of a tuple.
We use the empty tuple as the base of each graph.

a term $t \in \mathbf{terms}(g)$, we use $[t]_g$ for the GT (EC-node) that represents t in g - it is unique by the graph invariant. When an EC-node represents more than one term, we sometimes list more than one term for emphasis - for example if, in a graph g, $a = b$, then the EC for a can be written as $[a]_g, [b]_g, [a, b]_g$ all of which are equivalent. When the graph is clear from the context we drop the g subscript. When we have numbered CFG-nodes, e.g. p_0, p_1, we sometimes use e.g. $[a]_0, [b]_1$ to denote the GT representing a in g_{p_0}, b in g_{p_1}, resp, and also g_0 for g_{p_0}.
The graph in 3.6a is constructed by
`makeTerm(a,());makeTerm(b,());makeTerm(f,([a],[b]))` or
`makeTerm(b,());makeTerm(a,());makeTerm(f,([a],[b]))`
the results are identical.
3.6b is obtained from 3.6a by `makeTerm(f,([b],[a]))`. we can see how GT nodes can be shared - the nodes $[a], [b]$ participate in two tuples.
3.6c is obtained from either 3.6a or 3.6b by `assumeEqual([a],[b])` - the results are identical. We see one GT - [a,b] - which occurs twice in a tuple - hence the tuple-EC contains four tuples. 3.6d is obtained from 3.6b by `assumeEqual([f(a)],[f(b)])` - it cannot be obtained from 3.6c as all of our operations are monotonic in the set of represented equalities.
In figures 3.7 we show some graphs with cycles - note that some ECs are infinite.
The graph in 3.7a is obtained by:
```
makeTerm(a,()); makeTerm(f,([a])); assumeEqual([a],[f(a)])
```
The graph in 3.7b is obtained by:
```
makeTerm(a,()); makeTerm(b,())
makeTerm(f,([a],[b])); makeTerm(g,([a],[b]))
assumeEqual([f(a,b)],[a]); assumeEqual([g(a,b)],b)
```
Note that we can share the representation of tuple-ECs (we use shared tuple-ECs in our implementation).
The graph in 3.7c is obtained by:
```
makeTerm(a,()); makeTerm(b,())
makeTerm(f,([a],[b])); makeTerm(g,([b],[a]))
assumeEqual([f(a,b)],[a]); assumeEqual([g(b,a)],b)
```



(a)

Graph for the clause set $\emptyset$:
$\textbf{terms}(n) = \{a, b, f(a, b)\}$
$\textbf{gfas}(n) = \{a(), b(), f([a, b])\}$

(b)

Graph for $\{\}$:
$\textbf{terms}(n) = \{a, b, f(a, b), f(b, a)\}$
$\textbf{gfas}(n) = \{a(), b(), f([a], [b]), f([b], [a])\}$

(c)

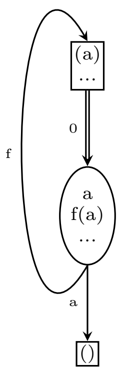
Graph for $\{a = b\}$:

$$\begin{aligned} \textbf{terms}(n) &= \left\{ \begin{array}{l} a, b, f(a, a), f(a, b), \\ f(b, a), f(b, b) \end{array} \right\} \\ \textbf{gfas}(n) &= \{a(), b(), f([a], [a])\} \end{aligned}$$

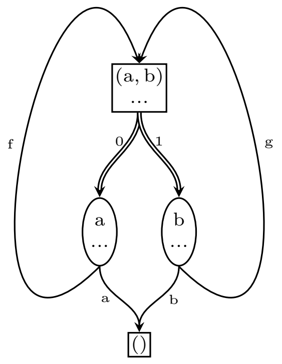
(d)

Graph for $\{f(a, b) = f(b, a)\}$:
$\textbf{terms}(n) = \{a, b, f(a, b), f(b, a)\}$
$\textbf{gfas}(n) = \{a(), b(), f([a], [b]), f([b], [a])\}$

Figure 3.6: Binary tuples



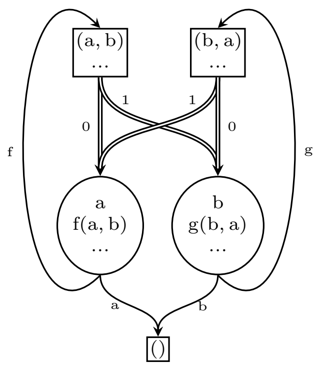
Graph for$\{a = f(a)\}$:
$\textbf{terms}(n) = \{a, f^n(a)\}$
$\textbf{gfas}(n) = \{a(), f([a])\}$



Graph for$\{a = f(a, b), b = g(a, b)\}$:

$$\begin{array}{lcl} \textbf{terms}(n) & = & \left\{ \begin{array}{l} a, b, f(a, b), \\ g(a, b), f(f(a, b), a), \\ g(f(a, b), b), ... \end{array} \right\} \\ \textbf{gfas}(n) & = & \{a(), b(), f([a], [b]), \\ & & g([a], [b])\} \end{array}$$

(a) (b)



Graph for$\{a = f(a, b), b = g(b, a)\}$:

$$\begin{array}{lcl} \textbf{terms}(n) & = & \left\{ \begin{array}{l} a, b, f(a, b), \\ g(a, b), f(f(a, b), a), \\ g(f(a, b), b), ... \end{array} \right\} \\ \textbf{gfas}(n) & = & \{a(), b(), f([a], [b]), \\ & & g([b], [a])\} \end{array}$$

(c)

Figure 3.7: Cyclic graphs

```
method verify(p: CFG)
   notVerifiedAssertions = ∅
   s := new Stack[ECGraph]
   g := new ECGraph

   traverseInPreOrder(p, entryVisitor, exitVisitor)

method entryVisitor(n : CFGNode)
   s.push(g)
   foreach ($s = t \in \mathbf{C}_n$)
      g.assumel(s=t)

   if (n.isLeaf)
      foreach ($s \neq t \in \mathbf{C}_n$)
         if (g.makeTerm(s)==g.makeTerm(t))
            return
      notVerifiedAssertions.add(n)

method exitVisitor
   g := s.pop
```

Figure 3.8: Basic equality verification algorithm - DFS
The method `traverseInPreOrder(p, inv, outv)` traverses the CFG in pre-order - it is standard but we detail it in 3.9 in order to prevent ambiguities when applying the method to DAGs. The method calls the callback inv(n) when first evaluating a CFG-node and outv(n) before back-tracking from the node.
The `entryVisitor` pushes the current EC-graph on the stack, `assumes` the axioms from the current CFG-node to the EC-graph, and if the CFG-node is a leaf node, it checks whether any dis-equality does not hold - which implies that the assertion at that leaf node holds (as assertions are negated).

3.2 Congruence closure graphs for verification

In this section we show an algorithm that uses our congruence closure graph data structure to verify a program VC where all clauses are unit ground equality clauses. This is not intended as a practical algorithm, but we use it to highlight the challenges posed by the CFG structure and how our data structures are built to answer these challenges. The algorithm mimics, in a sense, the operation of a DPLL prover on a certain class of VCs generated from such a program (including only unit ground equalities and dis-equalities).
For simplicity, we assume dis-equalities occur only in leaf nodes. The algorithm is presented in figure 3.8. The verification algorithm stems almost directly from the definition of the validity of a program - namely, that each assertion (leaf) node must be infeasible on all paths reaching it. The algorithm checks validity by enumerating the paths and, for each path, constructing an EC-graph with all the equalities on the path. It then checks if any of the dis-equalities is inconsistent with the EC-graph.
The only optimization in the algorithm is that we keep a stack of EC-graphs so that paths that share a prefix (prefix path in the CFG from the root) do not duplicate the work of constructing the EC-graph for the axioms on the CFG-nodes of the prefix.

```
method traverseInPreOrder(p, inv, outv)
   ts := new Stack[(CFGNode,Int)]
   n := p.root
   k := 0
   do
      if (k==0) //first visit of the node
         inv(n)
      if (k<|n.successors|)
         ts.push(n,k+1)
         (n,k) := (n.successors[k],0)
      else
         outv(n)
         if (!ts.isEmpty)
            (n,k) := ts.pop
   while (!ts.isEmpty)
```

Figure 3.9: Pre-order traversal of the CFG
This method performs standard pre-order traversal of a CFG, calling a visitor `inv` on entry to the node. In addition, the algorithm also calls a visitor `outv` when back-tracking from the node. We detail it here to make be clear about the behaviour in DAG traversal - namely, that each DAG-node is traversed as many times as there are paths leading to it.

This algorithm mimics the behaviour of DPLL with lazy CNF conversion for VC formulae of a specific form. The form (used by Boogie) encodes nested **let** expressions that encode the control flow - for each non assertion CFG-node n, the let expression used is
let $\mathrm{n_{ok}} = \mathrm{Ax_n} \Rightarrow \wedge_{\mathrm{s} \in \mathbf{succ}(\mathrm{n})} \mathrm{s_{ok}}$.
For assertion nodes the expression is
let $\mathrm{n_{ok}} = \mathrm{Ax_n}$.
$\mathrm{Ax_n}$ is the formula that represents the conjunction of the axioms in $\mathbf{C}_\mathrm{n}$.
the **let** expressions are nested in reverse topological order, and the main formula is $\mathrm{root_{ok}}$.
The formula is valid iff the program is correct. This encoding is efficient for lazy CNF conversion SMT solvers as it ensures that, at any point during the DPLL run, only the axioms from one path in the program are available to the prover. Note that SMT solvers often use a slightly different E-graph that supports removing equalities, and hence do not need to maintain a stack of graphs.

3.2.1 Complexity

The complexity of the above algorithm depends first on the complexity of operations on the EC-graph, so we discuss these first. For a set of equalities on ground terms, the complexity measure is the number of function symbol occurrences in the set. Different congruence closure algorithms are compared in [BT00]. They all result in a data structure that can answer efficiently the question of whether two given terms are equivalent under the conjunction of equalities. The efficiency of the above operations depends on the specific algorithm. Some algorithms perform congruence and transitive closure eagerly, while others try to minimize the time it takes to add assumptions, and only perform closure on queries relying on caching (e.g. path compression) to reduce the amortized time for the whole algorithm.

In all of the above algorithms the resulting data structure *represents* the set of all terms that were added to it (using `makeTerm`) and all of their equivalence class according to the congruence defined by all equalities for which we performed `assumEqual` - as in our implementation.

For most implementations, the way to check equality between two terms is to check whether they have the same representative. The time it takes to map a term t to its representative (our `makeTerm(t)` method) depends on the laziness of the algorithm - eager algorithms such as ours answer it in $O\left(|\mathrm{t}|\, log(n)\right)$ (n is the number of vertices in the graph) as the term needs to be mapped bottom up from constants, and each step up requires a lookup in the equivalent of our `superTerms` index.

All of the above algorithms compute the congruence closure only from positive equalities, and then it is possible to check for each dis-equality whether both sides of the dis-equality are in the same equivalence class, which signals a contradiction. For a dis-equality $\mathrm{s} \neq \mathrm{t}$ this check can be done by adding the terms s, t to the congruence closure data structure and then checking whether they map to the same equivalence class - so at most $O\left((|\mathrm{s}| + |\mathrm{t}|)\lg(n)\right)$. It is common to add dis-equalities to a congruence closure graph using edges between the nodes, and a graph is inconsistent if there is an edge between two nodes that are in the same EC. In our EC-graph inconsistency is represented by a dis-equality self edge.
We use dis-equality edges in our implementation, but we do not discuss them here as lazy propagation for dis-equalities is is not complete if we just propagate all dis-equality edges - we use the non-unit clause mechanism described in chapter 4 for dis-equalities and add the edges to the EC-graph for easier representation only - hence we do not discuss these edges in this chapter.

Complexity for a single set of (dis)equalities

Worst case time: The best known time complexity for deciding a set (conjunction) of unit ground equalities and dis-equalities is $O\left(n \lg(n)\right)$ - several congruence closure decision procedures have been shown sound and complete - for example [DST80],[NO03] and (the congruence closure part of) [Sho84] and [NO80], a survey with comparisons is found in [BT00]. These papers include also complexity analysis.
All of the above-mentioned algorithms essentially build a graph with terms as vertices, and perform congruence and transitive closure with differing levels of laziness. Roughly, the reason for the $O\left(\lg(n)\right)$ complexity is that the graph need never have more nodes and edges than there are terms in the original problem, and each time a new equality is inserted into the graph and two nodes are merged, we can remap the super-terms of the term with *less* super-terms, so each term is remapped at most $O\left(\lg_2(n)\right))$ times - this is proven in [DST80]. The number of equivalence classes is reduced by at least one for each non-redundant equality (potentially more in the case of congruence closure), so in total at most n such reductions can happen. An index is used to find the direct super-terms of each term for congruence closure and to update the graph (equivalent to our `superTerms` field).
The `superTerms` index can be implemented as an a-dimensional array for a

complexity $O(a)$ for insert, lookup and remap (on merge) where a is the largest function arity, but space $O(n^a)$ - for a total time complexity of $O(n\lg(n))$. A more space efficient option is a self balancing binary search tree for log(n) for the above operations but space $O(n)$ - giving an overall complexity of $O\left(n\lg^2(n)\right)$. Another practical solution is to use a hash table for average $O(1)$ operations and $O(n)$ space - giving the optimal average complexity but worst case $O\left(n^2\right)$ complexity - see [DST80] for a discussion. We compared the usage of search trees and hash tables in an earlier version of the implementation and hash tables performed consistently and significantly better.
Our algorithm is presented in a slightly simplified form, in the actual implementation we use a map `superTerms` that maps to super GFAs, rather than super GTs, hence saving the search for affected GFAs (this is more in line with the other algorithms), and we maintain a map from a GFA to its GT. The update of the `gfas` field can be postponed until all node merging is done (they are used for e.g. E-matching and superposition) and hence we can get a similar complexity to the common algorithms. However, in our setting, the dominant complexity factor is related to the CFG size rather than just the number of terms, as we detail below, and so preventing repeated traversals of the CFG, and preventing several CFG-nodes from performing the same congruence closure operation, is more important to complexity than each individual congruence closure operation.

Worst case space: Equality is described by either linking two vertices with an equality edge or merging the vertices. For transitive closure all the above algorithms do not produce all transitive edges but rather select one representative for each equivalence class and, lazily or eagerly, the path from each vertex to its representative is compressed to one edge, so at most one additional edge is added per vertex. Space complexity is, hence, $O(n)$, as we have at most one vertex and one edge per input symbol occurrence - a vertex will have at most as many incoming edges as it has occurrences - if f(a) and g(a) occur in the input, the graph will have just one vertex to represent a, but an incoming edge into a each for f(a), g(a). As detailed above, unless the array solution is used we get a space complexity of O(n). This linear space property explains, to some degree, the appeal of DPLL based solvers - while the search space is large, the size of any candidate model is proportional to the size of the problem (in SMT the CC graph is the representative model for uninterpreted functions).

Complexity for a tree-shaped CFG

We look first at tree-shaped CFGs in order to highlight the effect of CFG shape on verification complexity.

For a CFG we use two complexity measures - n measures the total number of function symbol occurrences in all clauses of all CFG-nodes and e measures the number of CFG edges - as we assume a CFG is of maximal out-degree two, e is proportional to the number of CFG-nodes.

For a CFG with no branches or joins, we can collect all clauses from all CFG-nodes into one set and then the complexity for deciding the validity of the program is as above practically $O\left(n\lg^2(n)\right)$. We assume here that there are no CFG-nodes that are neither a branch nor a join and that have an empty set of clauses.

For a binary-tree-shaped CFG, the maximal number of paths is proportional to e. If we simply collect all clauses for each path and apply any of the above-mentioned congruence closure graph construction algorithms for a set of clauses, the complexity is $O\left(n^2\ log^2(n)\right)$ as we have n paths - this bound is exact as we can have all positive clauses at the root and the only other clauses at the leaves. Using our algorithm from figure 3.8, the worst case time complexity is still $O\left(n^2\ log^2(n)\right)$ - consider the case where the root node includes the equations $\mathrm{c} = \mathrm{f}^{\mathrm{k}}(\mathrm{a}), \mathrm{f}^{\mathrm{k}}(\mathrm{b}) = \mathrm{d}$ and each leaf node includes $\mathrm{a} = \mathrm{b}, \mathrm{c} \neq \mathrm{d}$ where $\mathrm{k} = \mathrm{e}$ - each leaf node has to perform k congruence closure operations and hence we get quadratic complexity. However, practically, if equations are distributed more or less evenly between CFG-nodes, our algorithm, as it ensures no equation occurrence is evaluated more than once, can be expected to be more efficient.We are not aware of a better complexity bound for this problem.

In our context, we are interested in two improvements that can improve efficiency in practical cases. The intuition is that, in many cases, only a small proportion of the axioms are needed in order to prove each assertion. Instead of each CFG-node simply copying the EC-graph of a predecessor and adding its own axioms, each CFG-node starts with an EC-graph including only its own axioms, and adds axioms from predecessor on-demand - this is the main subject of section 3.3.
The second improvement is that, as our CFGs are constructed from programs in dynamic single assignment (DSA) form, we should be able to prove an assertion even if the EC-graph of each CFG-node only contains terms representing at most three DSA versions of each Boogie program variable - the DSA versions for the node and for its direct predecessors - hence we expect smaller EC-graphs. We explore this idea in the chapter 5.

Complexity for a DAG-shaped CFG

Even for a loop-free procedure with joins, the complexity of a DFS based algorithm rises to exponential because of join points which cannot be represented as conjunctions.

Our algorithm from figure 3.8 works for a DAG shaped program as well, in fact being a variation of the DPLL procedure: when backtracking from a node after having explored all its outgoing children, we practically forget that we have explored that node and the next time we reach it (on another path) we again explore all outgoing edges. The worst case complexity is dominated by the potentially exponential number of paths - for example, consider the program in figure 3.10 - the program size is proportional to n, but the number of paths is exponential in n.
Modern DPLL based solvers have several advantages over our naive procedure - notably:

- Clause learning - DPLL(CDCL) can learn a clause that generalizes part of the proof of the infeasibility of one path of exploration, which can help it prune some other paths from being explored
- Order of evaluation - DPLL can decide (depending on the encoding of the program VC as a formula and whether we use lazy CNF conversion) to explore the decision tree in a different order than our algorithm does - for example, it could `decide` on a literal that occurs at a join node before `deciding` the literals that determine which path leads to that join node -

```
$b_0$:
    assume $a_0 = d$
    if (*)
        assume $b_1 = a_0$
        assume $a_1 = b_1$
    else
        assume $c_1 = a_0$
        assume $a_1 = c_1$
$j_1$:

...
$j_{n-1}$:
    if (*)
        assume $b_n = a_{n-1}$
        assume $a_n = b_n$
    else
        assume $c_n = a_{n-1}$
        assume $a_n = c_n$
$j_n$:
    assert $a_n = d$ //negated $a_n \neq d$
```

Figure 3.10: linear join proof
The program (from [BDdM08]) can be proven in almost linear time using joins, but is exponential for DPLL and CDCL based provers

in that case the decision tree size might be less than exponential for some programs

We cannot expect better than exponential worst case complexity for DAGs as we can encode CNF-SAT to this problem in linear time - each clause with n literals is encoded into an n-branch with the nth literal on the nth direct successor and the branch is immediately joind, and each literal is encoded as described in section 2. The branch-join structures are organized in sequence and the CNF-SAT problem is satisfiable iff the last node (the only leaf node) is reachable - if we have not found a path leading to it in which we have found no contradiction.

However, we can easily construct a sequence of programs that each have a Hoare proof linear in the size of the program, but their encoding into SMT would take exponential time for current DPLL-CDCL solvers, the example at 3.10 is taken from [BDdM08]. This program takes exponential time for current CDCL solvers without provisions for join as the literals $a_i = a_{i+1}$ and $a_{i+1} = d$ do not exist in the original problem and hence cannot be learned by CDCL (we have tried both Z3 and CVC4 - both timed out after several hours for 50 diamonds, which were verified in less than a second using the algorithm presented later in this chapter).

3.2.2 Joins

A join for a pair of congruences is the intersection of the congruences and is also a congruence. We use the symbol $\sqcup$ for the join of two sets of clauses.
A congruence can be represented by a finite EC-graph iff it is representable

by a finite set of ground equations. Similarly, a join for two EC-graphs is an EC-graph for the intersection of the congruences represented by the EC-graphs.

For the program in figure 3.10, the join at each node $\mathrm{j_k}$ includes the equality $\mathrm{a_k = a_{k-1}}$, as it is in the join (intersection) of the congruences defined by the sets of equalities in both direct predecessors - written as:
$\mathrm{a_k = a_{k-1} \in \{a_{k-1} = b_k, a_k = b_k\} \sqcup \{a_{k-1} = c_k, a_k = c_k\}}$.
If we modify our verification algorithm to traverse the CFG in topological order (essentially breadth first) rather than in pre-order (essentially depth first), and if we had a way for calculating such joins efficiently, we could prove the above program efficiently as adding the joined equalities to the EC-graph at each join allows a very short proof - essentially, we only need to employ transitive closure for the set of clauses $\mathrm{a_0 = d, a_1 = a_0, ..., a_n = a_{n-1}, a_n \neq d}$. In the rest of this section we show the challenges in the approach that traverses *each CFG-node* in the CFG in topological order and calculates a join at each join node, and uses these joins for verifying the program rather than traversing each *CFG-path* in the CFG. We show in this section that a single forward pass of the CFG cannot calculate all necessary joins, and in the next section we show how we overcome this problem. We follow this idea in the rest of this chapter and extend it to richer fragments in later chapters.

Properties of joins

When discussing joins we use the term *joinees* for the two direct predecessor nodes of a join node, and refer to these nodes as $\mathrm{p_0}$ and $\mathrm{p_1}$ (in the order in which they are introduced in the text unless otherwise noted), and n for the join node.

We make an important distinction between calculating the actual join of two congruences at a join point in the program (which may not be finitely presentable as a set of equations), and between calculating a subset of the equalities in the intersection of congruences (an approximation of the join), which is sufficient to prove all assertions in transitive successors of the join point. A sufficient join is related to an interpolant between the congruence defined by its transitive predecessors and the congruence and dis-equalities defined by its transitive successors - we discuss interpolants in detail in chapter 5.

A fragment interpolant for the whole program for a logical fragment is an annotation ϕ_n for each node s.t. ϕ_n is in the fragment (in our fragment - a conjunction of unit equalities) and, for each CFG-node n, $\mathbf{C}_\mathrm{n} \wedge \bigwedge\limits_{\mathrm{p} \in \mathbf{ps}(\mathrm{n})} \phi_\mathrm{p} \models \phi_\mathrm{n}$, and, for each assertion node na, $\phi_\mathrm{na} = \mathbf{false}$. Note that we do not restrict the vocabulary of ϕ_n - we discuss signature restrictions in chapter 5, in this chapter we are only interested in the shape of ϕ_n.

Only valid programs have a fragment interpolant, in fact, a fragment interpolant is a form of validity proof for the program. While the definition of a fragment interpolant is similar to the one for abstract domains in abstract interpretation, our definition is weaker as it does not require a join operator - as we show in the next examples, there are programs from which the join at a node cannot be calculated effectively as a function of the predecessors of the join.

A join (disjunction or intersection) for two congruence relations (and similarly, two EC-graphs) cannot always be represented as the conjunction of a

```
if (*)
    p0 :
    assume c=a
else
    p1 :
    assume c=b
n :
if (*)
pt :
    assume a=b
    pta :
        assert a=c //negated a ≠ c
else
    ...
```

Figure 3.11: Non-unit join
No set of equations at n implied by all its predecessors is sufficient to prove the assertion.

finite set of unit ground equalities (or, equivalently by an EC-graph), or even by an infinite set - for example, the disjunction of the congruences defined by $\{a = b\}$ and $\{a = c\}$ cannot be represented by a set of ground unit equalities.

Consider the example in figure 3.11. Although each path from the root to the assertion in the program can be proven using only unit (dis)equalities, there is no set of ground unit (dis)equalities that holds at n and is sufficient to prove the assertion. As shown in [GTN04], even when the disjunction is representable as a set of unit equalities, this set is not necessarily finite. In light of this limitation, our objective in calculating a join is only to find small joins, and fall-back to non-unit ground equality clauses in other cases.
Worst case space complexity:
As stated above, a join is not always finitely representable, and a join as a conjunction of equalities is not always sufficient for proving all assertions. Hence we are interested only in the size of fragment interpolants when they exist - approximations of the join that are sufficient for proving all assertions in transitive successors. We discuss in this section the worst-case minimal size for such a fragment interpolant, if it exists.

[GN07] shows an exponential lower bound on the size of fragment interpolants - the authors show a program of size n (total function symbols) with only unit equalities (in fact only variable assignments - an even smaller class that does not create cyclic EC-graphs) that ends in a $\sqrt{n}$ sided join node for which the minimal representation of the equivalence class of a certain variable as a set of equalities is of size $\theta(2^{\sqrt{n}})$ (even in a representation where all shared sub-expression are represented only once - as in our EC-graph).
A rough description of the example is that, at the join point, the equivalence class of a variable includes just one more element, which is essentially a full binary tree of depth n of applications of a single binary function f, where at each leaf there is a full binary tree of depth $\lg n$ of f. Each such $\lg n$ depth tree has n leaves, each of which is either the term 0 or 1, but each tree has a different sequence of n 0s and 1s - so in total there are exponentially many distinct AECs. This is achieved at the join by using only a $\sqrt{n}$ sized term at each joinee.

```
b:
   P(n,x,a,b)
j0:
   P(n,y,c,d)
j1:
...
n:
   assume a=c
   assume b=d
   assert x=y
```

Figure 3.12: Exponential sized join proof

Designating this program of size n with the variable x and $\{a, b\}$ standing for $\{0, 1\}$ as $P(n, x, a, b)$, so that at the single leaf node of the program the variable x is equal to a term of size 2^n with only the constants a, b and the binary function symbolf f, we can construct the program shown in figure 3.12.

This program can be proven by annontating each CFG-node with a single EC-graph which is the join of the EC-graphs of its predecessors, but the EC-graph at j_1 must include at least two terms of exponential size - once with $\{a, b\}$ at the leaves and once with $\{c, d\}$, which are only equated at n.
Hence this program can be proven using the hypothetical algorithm with joins, but the minimal size of the EC-graph at the join is $\theta(2^n)$, while DPLL could simply explore each of the $\sqrt{n} \times \sqrt{n} = n$ paths of the program and produce a proof of size n for each path - so in total an $O\left(n^2\right)$ sized proof. This means that even for programs within our class we can have exponential sized minimal interpolant.

The complexity of verifying this class of programs (programs including only ground unit (dis)equalities as clauses) is NP-complete, as we have seen there are polynomial encodings to CNF. As the example above shows there are programs with worst case (single) exponential interpolated proof size as represented using EC-graphs (in our fragment), but we are not aware of a tighter lower or upper bound for the complexity of deciding whether such an interpolated proof exists. Our verification algorithm for general programs limits the size of interpolants, and hence, in programs as above, will fall-back to non-unit clauses at joins and rely on resolution and superposition at j_1 in order to derive a contradiction.

Calculating a join

We now discuss how a fragment interpolant can be calculated at the join, Which mainly centers around determining which terms need to be represented at the join.

In [GTN04], an $O\left(n^2\right)$ algorithm is given for joining two sets of unit equalities, which guarantees that any term that is represented on both sides (that is, for which there is a node in the graph that represents its equivalence class), is also represented at the join, and the join contains the full equivalence class for that term. The basic reason for this complexity is that any equivalence class at the join must correspond to a pair of equivalence classes, one from each joinee, while not necessarily all such pairs are needed in order to represent all the

```
if (*)
   $p_0$ :
   assume a=b
else
   $p_1$ :
   assume f(a)=f(b)
n :
if (*)
   assert g(f(a))=g(f(b)) //negated g(f(a)) $\neq$ g(f(b))
else
      ...
```

Figure 3.13: Join indirect congruence closure
Adding $f(a), f(b)$ to the EC-graph of p_0 allows us to prove the program with joins.

common terms (that is - terms represented on both sides of the join) at the join.

To illustrate: consider joining two identical congruence closure graphs - the join is exactly the size of each joinee, and the only pairs are two copies of the same equivalence class. On the other hand, consider joining $\{a = f^m(a)\}$ and $\{a = f^n(a)\}$ where m and n are co-prime - here the join would be $\{a = f^{mn}(a)\}$, so of a quadratic size representation. For both joinees and for the join point, the set of represented terms is $\left\{f^i(a) \mid i \in \mathbb{N}\right\}$. Here we are using all the pairs where each pair $([f^x(a)]_0, [f^y(a))]_1$ represents the equivalence class $\left\{f^{((xn[n^{-1}]_m+ym[m^{-1}]_n)\ \mathbf{mod}\ mn)+imn}(a) \mid i \in \mathbb{N}\right\}$, where $[n^{-1}]_m$ is the multiplicative inverse of n in the ring Z_m, by the Chinese remainder theorem.

For any equality $s = t$, we can determine whether it holds at a join by adding both s and t to both joinees, and performing a join as above. Adding a term t to a congruence closure graph is of complexity $|t| \lg |t|$ and can add at most $|t|$ edges to the graph. For the equality $s = t$ the time is then proportional to $(m + |t| + |s|) \times (n + |t| + |s|))$ multiplied by some log factor (depending on the way the lookup maps are constructed).

However, it is not immediately clear which terms should be added at which join - consider the example in figure 3.13. Here it is easy to see that adding $\{f(a), f(b)\}$ at p_0 is sufficient to prove the assertion.

On the other hand, consider figure 3.14, which could be generated as the VC of a program using the DSA form. In this case, we need to translate the terms $g(f(a_1)), g(f(b_1))$ through $a_1 = a, b_1 = b$ in order to derive the set of terms which would have to be added to the joinees of n (e.g. $\{f(a), f(b)\}$ would suffice) in order to be able to prove the assertion.

The example in figure 3.15 shows how deeper DSA chains affect the calculation of the set of terms required at a join. Here we want to propagate $f(a_i) = f(b_i)$ for each i, but not $g(a_i) = g(b_i)$. We need the information from all the nodes $p_1..p_n$ in order to determine which terms are needed at p_j. If k, n are of the same order of magnitude, propagating only $\{f(a_i), f(b_i)\}$ at each p_i would result in an interpolant of size $O(k + n)$, while an eager interpolant (that tries to include in the join each term that occurs on *either* side, as opposed to those that occur on both sides), that includes $\left\{g^k(a_i), g^k(b_i)\right\}$ for each p_i, is of the size $O(k \times n)$ - practically quadratic vs. linear size interpolant.

Even if we know the minimal size of a sufficient interpolant, such a minimal

```
if (⋆)
    $p_0$ :
    assume a=b
else
    $p_1$ :
    assume f(a)=f(b)
n :
    assume $a_1$ = a
    assume $b_1$ = b
    if (⋆)
        $p_t$ :
        assert $g(f(a_1)) = g(f(b_1))$ //negated $g(f(a_1)) \neq g(f(b_1))$
    else
        ...
```

Figure 3.14: Join indirect congruence closure DSA
This program includes the variables a, b in two DSA versions

```
$p_b$ :
if ($c_1$)
    $p_t$ :
    assume $a = b$
else
    $p_e$ :
    assume $f(a) = f(b)$
    assume $g^k(a) = g^k(b)$
$p_j$ :
...
$p_1$ :
    assume $a_1 = a$
    assume $b_1 = b$
$p_2$ :
    assume $a_2 = a_1$
    assume $b_2 = b_1$
    ...
$p_n$ :
    assume $a_n = a_{n-1}$
    assume $b_n = b_{n-1}$
    $p_{na}$ :
    assert $f(a_n) = f(b_n)$ //negated $f(a_n) \neq f(b_n)$
```

Figure 3.15: Join congruence closure DSA chain
The program variables a, b have several DSA versions, and each CFG-node only refers to up to two versions of each variable. In order to determine which terms need to occur at the join, we need knowledge of the assertion and the entire DSA chain.

```
if (*)
    p0 :
    assume a = b
    assume c = d
    assume f(a, a) = e
    assume f(a, d) = e
    assume f(d, a) = e
    assume f(d, d) = e
else
    p1 :
    assume a = c
    assume b = d
    assume f(c, c) = e
    assume f(c, b) = e
    assume f(b, c) = e
    assume f(b, b) = e
n :
if (*)
    pt :
    assume a = b = c = d = x
    assert f(x, x) = e //negated f(x, x) ≠ e
else
    ...
```

Figure 3.16: congruence closure source quadratic

interpolant is not unique - consider, for example, the program in figure 3.16. Here, any of the 16 options $\{f(x, y) = e \mid x, y \in \{a, b, c, d\}\}$ would suffice as an interpolant, and there is no specific reason to choose any of the interpolants - they are symmetric.

In selecting an interpolant, there are also incremental considerations. Our general verification algorithm interleaves the verification steps of different logical fragments. For the purpose of the discussion in this chapter, this is important because the algorithm might apply a step of the unit ground equality fragment (that is, try and verify the program with the current set of unit equalities), then apply some steps from another fragment which might produce new equalities at some CFG-nodes, and then again try to verify the program with unit equalities. In such a scenario, we want to ensure that all verification steps are performed incrementally, and specifically that join approximations (fragment interpolants) are not recalculated from scratch every time we apply the ground equalities fragment, but rather that the fragment is applied incrementally.

Consider the example in figure 3.16, where the **else** branch has no ground equalities. Assume we have selected one interplant from the sixteen possible ones and proven our assertion in the **then** branch.
Now the verification algorithm applies one step of another fragment which produces some new unit clauses at the **else** branch, as shown in figure 3.17. In the new program, there is exactly one specific minimal interpolant - $f(a, d) = e$. If we have chosen another interplant in the previous stage, we cannot reuse it and have to recalculate the join.

The example in figure 3.18, adapted from [GTN04], shows that, for some

```
...
n :
if (*)
    p_t :
    assume a = b = c = d = x
    assert f(x,x) = e //negated f(x,x) ≠ e
else
    p_e :
    assert f(a,d) = e //negated f(a,d) ≠ e
```

Figure 3.17: Congruence closure - incremental interpolant
This is a modification of the program in figure 3.16.
We assume the dis-equality on the **else** branch is added after the join has been calculated.

joins, any set of equalities that represents the congruence at the join is infinite (the congruence is not finitely presented) for finite sets of clauses at the joined nodes. However, there are several interpolants for the program at the join - including:
$g(f^2(a)) = g(f^2(b)) \wedge g(f^3(a)) = g(f^3(b))$ and
$g(f^6(a)) = g(f^6(b))$.

In the example in figure 3.19, the minimal interpolant is of quadratic size (m,n co-prime). In this case our algorithm will fall back to non-unit guarded equalities (in the fragment of ground superposition), to get at n (p is the branch condition joined at n):
$\neg p \vee a = f^m(a)$
$\neg p \vee f^{2m}(a) = f^m(a)$
$\neg p \vee g(f^m(a)) = d$
$p \vee a = f^n(a)$
$p \vee f^{2n}(a) = f^n(a)$
$p \vee g(f^n(a)) = d$
Which is an interpolant linear in the size of the input ($O(m+n)$) - if represented using shared common sub-expressions (as we do in our EC-graph) we only use $O(max(\mathrm{m},\mathrm{n}))$.
At p_t, the interpolant is reduced by congruence closure to:
$a = f(a)$
$\neg p \vee a = a$
$\neg p \vee a = a$
$\neg p \vee g(a) = d$
$p \vee a = a$
$p \vee a = a$
$p \vee g(a) = d$
Now tautology elimination ($\mathrm{taut}_=$) would leave:
$a = f(a)$
$\neg p \vee g(a) = d$
$p \vee g(a) = d$
And then reducing resolution ($\mathrm{simp}_{\mathrm{res}}$):
$a = f(a)$
$g(a) = d$

```
$p_b$ :
if ($c_1$)
    $p_0$ :
    assume $a = f(a)$
    assume $b = f(b)$
    assume $g(a) = g(b)$
else
    $p_1$ :
    assume $a = b$
n :
// Here $\forall i \cdot g(f^i(a)) = g(f^i(b))$ holds
if (*)
    $p_{j_t}$ :
    assume $a = f(a)$
    assume $f^2(b) = c$
    assume $f^2(c) = c$
    assert $g(a) = g(c)$
else
    $p_{j_e}$ :
    assume $f^3(a) = c$
    assume $f^3(c) = c$
    assume $b = f(b)$
    assert $g(c) = g(b)$
```

Figure 3.18: Infinite join for two congruences
For each i, $g(f^i(a)) = g(f^i(b))$ at the join n - this congruence is not finitely representable as a set of equalities.

```
if (*)
    $p_0$ :
    assume $b = f^m(a)$
    assume $b = f^m(b)$
    assume $g(b) = d$
else
    $p_1$ :
    assume $c = f^n(a)$
    assume $c = f^n(c)$
    assume $g(c) = d$
n :
    //Here $f^{2mn}(a) = f^{mn}(a)$
    //  and $g(f^{mn}(a)) = d$
    if (*)
        $p_t$ :
        assume $a = f(a)$
        assert $a = g(d)$ //negated $a \neq g(d)$
    else
        ...
```

Figure 3.19: congruence closure source quadratic depth
m,n are co-prime.
The minimal interpolant is of quadratic size.

```
    if (c1)
        $p_0$ :
        assume $a = b$
    else
        $p_1$ :
        assume $f(a) = a$
        assume $f(b) = b$
        assume $g(a) = g(b)$
n :
    ...// Here, for any k, $g(f^k(a)) = g(f^k(b))$ holds
$n_i$ :
    assume $f(a) = a$ //Here, for any k, $g(a) = g(f^k(b))$
    assert $g(a) = g(b)$ //negated $g(a) \neq g(b)$
```

Figure 3.20: Join infinite equivalence class (m,n co-prime)

Which is sufficient to prove the assertion.

The next example, in figure 3.20 (based on an example from [GTN04]), shows an infinite equivalence class *after* the join - the set of provable equalities for g(a) at n is infinite - namely $\left\{g(a) = g(f^k(b)) \mid k \in \mathbb{N}\right\}$. If there is a fragment interpolant then only a finite subset of each such infinite equivalence class is needed to prove the program.

The example in figure 3.21 shows that, in a case as in figure 3.20 where an EC at the join is inifinite but an interpolant does exist, the minimal size of any interpolant depends on all equations in successor nodes - hence an algorithm as suggested before that traverses the CFG in topological order and calculates interpolants cannot decide the class of program where interpolants exist. In order to prove the assertion at n_j, the smallest equality we would have to include at n is $g(f^{mn}(a)) = g(f^{mn}(b))$. The parameter m is not known when calculating the interpolant at the node n if we traverse the CFG in topological order, and similarly the paramenter n is not known when traversing the CFG in reverse topological order - this essentially means that abstract interpretation for the domain of finite conjunctions of ground unit equalities cannot define a join operator that decides all programs in this fragment that have an interpolant - the join cannot be determined in one pass of the program.

A variant of the problem in 3.21 is shown in figure 3.22. Here, if $m = n$ and $n > 1$ then the assertion does not hold, but if m,n are co-prime then the assertion holds, with a minimal interpolant of size greater than mn.

The following example, shown in figure 3.23, suggests that, unlike the example in [GN07], the number of joins is not the only source for super quadratic complexity (remember that their example only allows clauses derived from the DSA form of assignments, and hence cannot create cycles in the EC-graph, which are created by `assume` statements): for any given polynomial degree d, this example requires an interpolant of size at least a polynomial of degree d, as follows:

We designate the ith prime number starting at two as p_i. Given a polynomial degree d, we show a sequence of programs where the nth program is of size linear in n but the minimal interpolant for the nth program is of size polynomial in n. For the nth program in the sequence, we select the integers $m_1..m_d$ s.t. the

```
    if (c1)
        $p_0$ :
        assume $a = b$
    else
        $p_1$ :
        assume $f(a) = a$
        assume $f^n(b) = b$
        assume $g(a) = g(b)$
n :
    ...// Here holds, for any k>0, $g(f^{nk}(a)) = g(f^{nk}(b))$
n_i :
    assume $f(a) = a$ //Here, for any k, $g(a) = g(f^{nk}(b))$
    //Smallest sufficient interpolant $g(f^{mn}(a)) = g(f^{mn}(b))$
    ...
p :
n_j :
    assume $f^{2m}(b) = f^m(b)$
    assert $g(a) = g(f^m(b))$ //negated $g(a) \neq g(f^m(b))$
```

Figure 3.21: join infinite equivalence class

```
    if (c1)
        $p_0$ :
        assume $a = b$
    else
        $p_1$ :
        assume $f(a) = a$
        assume $f^n(b) = b$
        assume $g(a) = g(b)$
n :
    ...// Here holds, for any k>0, $g(f^{nk}(a)) = g(f^{nk}(b))$
n_i :
    assume $f(a) = a$ //Here, for any k, $g(a) = g(f^{nk}(b))$
    ...
p :
n_j :
    assume $f^{2m}(b) = f^m(b)$
    assert $g(a) = g(f^{m+1}(b))$ //negated $g(a) \neq g(f^{m+1}(b))$
```

Figure 3.22: Join infinite equivalence class
The assertion holds iff $\exists k, l \cdot k\mathrm{n} = l\mathrm{m} + 1$

```
    if (c1)
        $p_0$ :
        assume $a_1 = a_2$
        ..
        assume $a_1 = a_d$
    else
        $p_1$ :
        assume $f(a_1) = a_1$
        ...
        assume $f(a_d) = a_d$
        assume $g(a_1, a_1...., a_1) = g(a_1, a_2, ..., a_d)$
n :
//Here, for any k, $g(f^k(a_1), f^k(a_1), ...., f^k(a_1)) = g(f^k(a_1), f^k(a_2), ...f^k(a_d))$
$n_a$ :
    assume $f^{2m_1}(a_1) = f^{m_1}(a_1)$
    assume $f^{2m_2}(a_2) = f^{m_2}(a_2)$
    ....
    assume $f^{2m_d}(a_d) = f^{m_d}(a_d)$
    assert $g(f^{m_1}(a_1), f^{m_1}(a_1), ...f^{m_1}(a_1)) = g(f^{m_1}(a_1), f^{m_2}(a_2), ..., f^{m_d}(a_d))$
```
Figure 3.23: One join super-quadratic interpolant

program above is in size linear in n and the minimal interpolant is of minimal size polynomial of degree d in n:
We use $m_i \triangleq p_i^{\lceil \lg_{p_i} n \rceil}$ - this means that m_i is the lowest power of p_i that is greater or equal to n - hence $n \leq m_i < p_i n \leq p_d n$. The size of the program (unique subterms) is at most $4d + 2dnp_d$, as each of m_i is smaller than $p_d n$.
The only equalities that hold at the join are of the form
$g(f^k(a_1), f^k(a_1), ..., f^k(a_1)) = g(f^k(a_1), f^k(a_2), ...f^k(a_d))$ for some k.
Each such k has to satisfy the set of constraints $\{m_i | k \mid 0 < i \leq d\}$. As m_i are pairwise co-prime, this entails that k is at least $\prod_i m_i$, which satisfies $\prod_i m_i \geq n^d$.
We believe our example suggests that the worst case size complexity for unit interpolants in a CFG might be higher than double exponential, but the exact lower bound for the space complexity of a fragment interpolant for our class of programs remains an open question as far as we are aware. The decision problem of whether a fragment interpolant exists is at least semi-decidable, as we can enumerate all such interpolants and the entailment checks are decidable.

Summary

We have seen that for some programs joins can reduce the verification runtime exponentially, while in other cases a join does not exist as a set of equalities. A fragment interpolant (a sufficient approximation for the join as a set of equalities) for a single binary join for a set of terms is up to quadratic in the number of terms, but the size of the terms needed at each join can be a polynomial of any degree in the input size. A minimal fragment interpolant can also be of quadratic depth and not only size.
The minimal representation of the equivalence class of a term at a node after a single binary join can be of size 1 up to infinite, and the size of the representa-

tion of a member of the class after n consecutive joins can be exponential. Furthermore, there may not be a single minimal interpolant, and even when there is a minimal interpolant, the minimal interpolant for an extension of the program is not necessarily an extension of the minimal interpolant.
Most of the examples we have shown in this section are pathological examples that explore the complexity bounds of the fragment interpolant, and not practical examples. In actual VCs of programs, we have found that the choice of which equalities are propagated using fragment interpolants at joins, and which are propagated using non-unit clauses, has a significant effect on the performance of our algorithm. In the next sections we show how our algorithm selects which part of the join is calculated, and which is relegated to richer fragments.

3.3 Information Propagation

We have seen in the previous section that verification using a DPLL or CDCL based solvers suffers from some inherent problems when joins are concerned, even when all clauses are unit ground clauses. While the example program we have seen (figure 3.10) is a synthetic example that we do not expect to see as a real program, we have encountered the same problem when quantifiers are involved - namely, when a solver explores the search space of a program proof by enumerating program paths, and when quantifier instantiation is involved, if the CDCL part of the prover cannot learn a lemma (clause that holds at the join) that involves an instance of a quantified clause, the number of times a quantifier is instantiated can grow exponentially with the number of sequential joins in the program.

We have run the Boogie program in figure 3.24 for different versions of k using Z3, and counted the number of instantiations of heap axioms (both read-over-write with equal and non-equal indices) - the results are detailed below.
We run this program for different values of k with the following results:

k	ROW=	ROW≠
0	3	2
1	6	8
2	12	24
4	88	352
10	2868	20480
10h	182	172

The first column shows the number of consecutive branch-join pairs - the meaning of the last row is described below.
The second and third columns show the number of instances of the following axioms (heap axioms):
$\mathrm{ROW}{=}{:}\forall \mathrm{h}, \mathrm{x}, \mathrm{v} \cdot \mathrm{h}[\mathrm{x} := \mathrm{v}][\mathrm{x}] = \mathrm{v}$
$\mathrm{ROW}{\neq}{:}\forall \mathrm{h}, \mathrm{x}, \mathrm{y}, \mathrm{v} \cdot \mathrm{x} \neq \mathrm{y} \Rightarrow \mathrm{h}[\mathrm{y} := \mathrm{v}][\mathrm{x}] = \mathrm{h}[\mathrm{x}]$
The last row is a version of the program where k=10 but we have added a copy of the assertion after each join as a hint for the prover - this hint adds a literal to the input that the prover can use in learned lemmas, hence the significant reduction in axiom instances.
The results show that heap manipulating programs with many joins can benefit from calculating joins, at least for the prover Z3. We have experienced similar performance with other axioms as well in programs with complicated CFGs. We have tried the example below also with the SMT solver CVC4. For CVC4 we did not see such an increase in quantifier instantiations for low values of k, but with k = 20 the solver timed out after two minutes, while it took less than a second to prove the case of k = 10 (similar to the example in figure 3.10). The SMTlib input includes no other quantified axioms but the above two. Our verification algorithm presented in this chapter, together with the heap fragment presented in section 8.1 solve this problem for k = 40 in less than a second.

3.3.1 A verification algorithm

We have seen in the last section and above that enumerating program paths has some inherent weaknesses when join nodes are present in the program, not all of which are mitigated by current CDCL technology. In addition, we have

```
assume r0≠r1
assume r0≠r2
assume r1≠r2

heap[r0] := 0;
heap[r1] := 1;
heap[r2] := 2;

heap[r1] := 1;
if (*)
   heap[r1] := 1;
else
   heap[r2] := 1;

....

heap[r1] := k;
if (*)
   heap[r1] := k;
else
   heap[r2] := k;
assert heap[r0] == 0;
```

Figure 3.24: Number of read-over-write axiom instantiations with branch-join pairs.

seen that for a set of ground equalities, we have a method to select a sub-set that is sufficient to decide whether a given equation is entailed, as shown in the algorithm in figure 3.4.

We have also seen in some examples for unit equalities, such as 3.21, that in order to calculate an approximation of the join that is sufficient to prove the assertions following the join (a fragment interpolant), we need information both from the transitive successors and transitive predecessors of the join, hence the join cannot be calculated in a single backward or forward pass of the program.

In this section we propose a verification algorithm that aims to avoid the above-mentioned shortcomings. The algorithm is based on a traversal of the CFG in topological order, but it behaves lazily in propagating information forward in the CFG. While our algorithm of figure 3.8 which mimics, to a degree, the behavior of DPLL on a certain encoding of VCs, carries the EC-graph of each CFG-node to each of its successors, we propose to propagate this information lazily - initially each CFG-node constructs an EC-graph that contains only its own clauses, and later, when needed, it requests more clauses from its predecessors, with a mechanism similar to the one in figure 3.4.

The algorithm is presented for verifying programs with arbitrary sets of clauses, and we show how it is specialized to work with EC-graphs. In this section we discuss the mechanism for propagating clauses.

The outline of the algorithm is shown in figure 3.25. The algorithm traverses the CFG in topological order and verifies each CFG-node in turn.
For each node, the algorithm maintains two sets of clauses, as is common in saturation provers:
The first set, `todo`, contains clauses that are yet to be processed.

The second set, `done`, contains clauses that are inter-saturated w.r.t. the logical calculus, and also for which all relevant clauses from predecessors have been propagated.
The algorithm begins with only the clauses that belong to the node, and gradually propagates clauses from predecessor nodes as they are needed. For each clause that is selected to be processed, the algorithm first propagates all relevant clauses from predecessors, and then performs all inferences with the already saturated clauses. The process terminates when there are no more non-saturated clauses.

The request mechanism

In figure 3.26 we show the algorithm that imports the relevant clauses for a given clause from all of its transitive predecessors.
The algorithm uses the method `traverseBF` shown in figure 3.27 - this method performs a traversal of the CFG starting at n and going backwards and then forward - we present an implementation simply to disambiguate its semantics, the algorithm itself is mostly standard DAG traversal.
At each CFG-node in the backwards traversal, the method calls a visitor `bVisitor` which returns a set of predecessors relevant for traversal - in our case, each CFG-node may decide not to propagate the request backwards.
After backward traversal, the method traverses forward in reverse order, calling the visitor `fVisitor` - in our case, this visitor propagates clauses matching the request from direct predecessors.

Our backward visitor `importBW` subtracts from a request all cached requests, and if any uncached requests are left it propagates them to its direct predecessor. The cache is updated accordingly.
Our forward visitor `importFW` propagates all relevant clauses from the direct predecessors and adds them to the current CFG-node.

Clause relevance

Essentially, a clause is relevant for another clause in a calculus if there is an inference in the calculus where both clauses are premises. For a simple example, consider propositional ordered resolution - as resolution is allowed only between clauses with maximal literals with opposing polarity but the same atom, a clause $C \lor \underline{A}$ (A maximal) is relevant for a clause $D \lor \underline{B}$ iff $A \equiv \neg B$.
For each logical fragment, we have a `Request` data structure that encodes the information needed to determine the relevant clauses for a clause (or set of clauses) - for example, for ground ordered resolution, we only need the maximal literal of a clause, hence the request structure includes a set of literals.
The relevance criterion for each fragment is implemented in the `isRelevant` method referenced in the code - in this section we describe the criterion for EC-graphs and for unit ground superposition - for ordered resolution, for a request `r` which is a set of literals and a clause $C \lor \underline{A}$ (A maximal), `isRelevant` returns $A \in$ `r`.

The request Cache

The algorithm maintains a cache at each CFG-node, which remembers which requests have already been answered for that node. When a new request arrives at a node, the cache is subtracted from the request (in order not to re-request previously answered requests from predecessors) and if there are any un-answered requests they are sent to the predecessors. The cache is updated accordingly. Generally, the request cache structure is identical to a request structure. For ordered resolution, the request cache is a request - a set of literals, the operations `add` and `subtract` for the cache are simply set union and difference and the `isEmpty` operation for the requests simply checks if the set is empty.

The important property of the cache is that a node that has a certain request cached has already imported all relevant clauses (added to the `done` set) from all its transitive predecessors.

For ordered resolution, if the request cache at a CFG-node n includes the literal A, it means that all clauses in all transitive predecessors of n with a maximal literal $\neg$A have been propagated to `n.done`.
For some fragments, the requests and caches over-approximate the set of relevant clauses (we give an example later in this section) - the intuition behind over-approximating the request is that, for most clauses, most CFG-nodes do not have a relevant clause, and hence if our cached propagation criterion over-approximates the set of clauses that need to be propagated, it can reduce the number of CFG traversals for later requests. In addition, when we apply our algorithm incrementally, a predecessor node can derive a clause that is relevant for a clause already saturated in a previous pass in some successor. The cache allows us to propagate the newly derived clause during the CFG traversal in `CFG.verify`, without having to request it again in the successor - we describe this process for EC-graphs in the next section.

Clause propagation

The algorithm propagates clauses (in the method `importFW`), by selecting the relevant clauses from the predecessor's `done` set, and adding them, relativized as described in 2, to the `done` set. The reason clauses are added to the `done` set is to prevent any inference between two imported clauses - such an inference is performed at an earlier node. (we discuss relativization in chapter 4 - for ground unit clauses, we only propagate clauses we can join, and hence, in practice, no relativization takes place - we only mention it here as otherwise the algorithm is unsound).

In the rest of this section we present the propagation criteria for superposition and for EC-graphs, and compare them. Our algorithm uses the EC-graph criterion for unit equalities and the superposition criterion for the non-unit and non-ground fragments.

3.3.2 Clause propagation criteria

In this section we compare the propagation criteria for ground unit superposition and for EC-graphs.

Superposition based propagation

We look now at the ground superposition calculus, restricted to unit clauses, shown in figure 3.28.

```
CFG.verify() : Set of unverified assertions
   foreach node n in topological order
      n.verify()
      if (!n.isInfeasible and n.isLeaf)
         result.add(n)

Node.verify()
   todo.enqueue( $\mathbf{C}_n$ )
   while !todo.isEmpty
      c = todo.dequeue
      done.add(c)
      importRelevantClauses(c)
      foreach (d ∈ inferences(c,done))
         if (d ∉ done)
            todo.enqueue(d)
```

Figure 3.25: Verification algorithm with lazy propagation

```
Node.importRelevantClauses(c : Clause)
   requestMap := new Map[Node,Request]
   requestMap[this] := makeRequest(c)

   traverseBF(this, importBW, importFW)

importBW(n : CFGNode) : Set[CFGNode]
   var r := requestMap[n]
   r.subtract(n.requestCache)
   if (!r.isEmpty && n!=root)
      foreach (p ∈ $\mathbf{ps}(n)$)
         requestMap[p].add(r)
      return predecessors
   else
      return ∅

importFW(n : CFGNode)
   r := requestMap[n]
   n.requestCache.add(r)
   foreach (p ∈ $\mathbf{ps}(n)$)
      foreach (pc ∈ p.done)
         c := relativize(p,n,pc)
         if (isRelevant(r,c))
            n.done.add(c)
```

Figure 3.26: Basic clause propagation

The implementation of `traverseBF` is shown in figure 3.27.

The clause propagation algorithm first traverses the CFG in reverse topological order, starting at the current node. The traversal stops at any CFG-node that already has all relevant clauses cached.

After the reverse traversal, the algorithm traverses the CFG forward, propagating relativized clauses and updating the cache.

```
traverseBF(n : CFGNode, bVisitor, fVisitor)
   todoBW = new Set[Node]
   todoBW.add(n)
   todoFW = new Stack[Node]
   while (!todoBW.isEmpty)
      var n := todoBW.removeMax (topological order)
      var ps := bVisitor(n)
      if (ps==∅)
         todoFW.push(n)
      else
         todoBW.add(ps)

   while (!todoFW.isEmpty)
      n := todoFW.pop
      fVisitor(n)
```

Figure 3.27: CFG traversal back and forth
The implementation is shown only to clarify any ambiguities.
The algorithm traverses the CFG backwards from `n`, calling `bVisitor` on each node traversed.
`bVisitor` returns the set of predecessors relevant for traversal - if none are returned then traversal does not continue beyond the node (in a DAG a branch node may be relevant for only one successor, in which case it is traversed.)
The algorithm then traverses forward from each node where traversal ended, calling `fVisitor`.

$$\mathrm{res}_= \quad \frac{\mathrm{s} \neq \mathrm{s}}{\Box}$$

$$\mathrm{sup}_= \quad \frac{\underline{\mathrm{l}} = \mathrm{r} \quad \underline{\mathrm{s}} = \mathrm{t}}{\mathrm{s}\left[\mathrm{r}\right]_\mathrm{p} = \mathrm{t}} \quad \begin{array}{l} \text{(i)}\mathrm{l} = \mathrm{s}|_\mathrm{p} \\ \text{(ii)}\mathrm{l} \succ \mathrm{r} \\ \text{(iii)}\mathrm{s} \succ \mathrm{t} \\ \text{(iv)}\mathrm{s} = \mathrm{t} \succ \mathrm{l} = \mathrm{r} \end{array}$$

$$\mathrm{sup}_{\neq} \quad \frac{\underline{\mathrm{l}} = \mathrm{r} \quad \underline{\mathrm{s}} \neq \mathrm{t}}{\mathrm{s}\left[\mathrm{r}\right]_\mathrm{p} \neq \mathrm{t}} \quad \begin{array}{l} \text{(i)}\mathrm{l} = \mathrm{s}|_\mathrm{p} \\ \text{(ii)}\mathrm{l} \succ \mathrm{r} \\ \text{(iii)}\mathrm{s} \succ \mathrm{t} \end{array}$$

Figure 3.28: The unit ground superposition calculus $\mathbf{SP_{gu}}$
Maximal literals are underlined for clarity

It is apparent from the formulation of the ground superposition calculus, that the relevance of a clause for any binary inference depends mostly on its maximal term - similar to ordered resolution where relevance depends on the maximal literal.

For a clause $\underline{\mathrm{l}} = \mathrm{r}$ $(\mathrm{l} \succ \mathrm{r})$, the clauses that can participate in an inference with $\mathrm{l} = \mathrm{r}$ as the left premise are those in which l is a sub-term of the maximal term - formally:
$\underline{\mathrm{s}} \bowtie \mathrm{t}$ where $\mathrm{s} \succ \mathrm{t}$, $\mathrm{l} \trianglelefteq \mathrm{s}$, $\mathrm{s} \bowtie \mathrm{t} \succ \mathrm{l} = \mathrm{r}$ (s is a super-term of l)
The clauses that can participate where $\underline{\mathrm{l}} = \mathrm{r}$ is the right premise are those in which the maximal term is a sub-term of l - formally:
$\underline{\mathrm{s}} = \mathrm{t}$ where $\mathrm{s} \succ \mathrm{t}$, $\mathrm{s} \trianglelefteq \mathrm{l}$, $\mathrm{l} = \mathrm{r} \succ \mathrm{s} = \mathrm{t}$ (s is a sub-term of l)
The clause $\mathrm{s} \neq \mathrm{t}$ can only be a right premise, and then the maximal term of the left premise must be a subterm of s - formally:
$\mathrm{l} = \mathrm{r}$ where $\mathrm{l} \trianglelefteq \mathrm{s}$, $\mathrm{l} \succ \mathrm{r}$ (includes also $\mathrm{s} = \mathrm{t}$)

We can see from the above formulation that for a positive clause $\mathrm{l} = \mathrm{r}$ where l is maximal we want all clauses where the maximal term is a sub- or super-term of l, and for a negative clause $\mathrm{s} \neq \mathrm{t}$ (s maximal) we want all positive clauses where the maximal term is a sub-term of s.
For example - for the clause $\underline{\mathrm{f(b)}} = \mathrm{a}$, the clauses $\underline{\mathrm{f(b)}} = \mathrm{b}, \underline{\mathrm{g(f(b))}} = \mathrm{c}$, $\underline{\mathrm{b}} = \mathrm{a}$ are all relevant, but the clauses $\underline{\mathrm{b}} \neq \mathrm{a}, \underline{\mathrm{f(c)}} = \mathrm{f(b)}, \underline{\mathrm{c}} = \mathrm{b}, \underline{\mathrm{f(c)}} \neq \mathrm{f(b)}$ are not.

The method `isRelevant` for the ground superposition fragment implements the criteria described above.

For ground superposition the requests and cache is implemented as follows: The cache includes two sets of terms, $\mathtt{ts}_{\mathrm{lhs}}$ and $\mathtt{ts}_{\mathrm{rhs}}$ - the first for terms that occur as maximal positive terms and the second for terms that occur as a sub-term of a maximal term in a clause of any polarity. If a term t is in $\mathtt{ts}_{\mathrm{lhs}}$, then any clause with a maximal term a (non-strict) super-term of t is propagated.
For a term t in $\mathtt{ts}_{\mathrm{rhs}}$, any positive clause with a maximal term that is a (non-strict) sub-term of t is propagated.
The `subtract` method is simply set difference on each of the two sets.
When updating the cache in `requestFW`, for a positive clause $\underline{\mathrm{l}} = \mathrm{r}$ we add l to $\mathtt{ts}_{\mathrm{lhs}}$ and the sub-term closure of l to $\mathtt{ts}_{\mathrm{rhs}}$. For a clause $\underline{\mathrm{s}} \neq \mathrm{t}$ we add the sub-term closure of s to $\mathtt{ts}_{\mathrm{rhs}}$.
A clause $\underline{\mathrm{s}} \bowtie \mathrm{t}$ is relevant for a request `r` (`isRelevant(r,C)`) if it is positive and a (non-strict) sub-term of s is in $\mathtt{ts}_{\mathrm{lhs}}$, or if C is of any polarity and a (non-strict) sub-term s is in $\mathtt{ts}_{\mathrm{rhs}}$.

For example, if we request propagation for the clauses $\underline{\mathrm{f(b)}} = \mathrm{a}$ at the CFG-node n, $\mathtt{ts}_{\mathrm{lhs}}$ of the cache will include, after propagation, the term f(b), and $\mathtt{ts}_{\mathrm{rhs}}$ will include $\mathrm{b}, \mathrm{f(b)}$ so that if, at a later stage, any CFG-node requests relevant clauses for the clause $\underline{\mathrm{b}} \neq \mathrm{a}$, the request will not be propagated further.
The advantage of such a formulation for the cache is that, when we limit the set of terms that can occur at a CFG-node (in all clauses at that node), we can bound the number of times this node has to answer requests. We limit the set of terms both by size bounds and by scoping, as detailed in chapters 6 and 5.

Congruence closure based propagation

Even for a large set of equalities without any dis-equality, many superposition derivations can take place, although the set is trivially satisfiable. More generally, even in a set with dis-equalities, we can find a subset of the equalities that is sufficient to show a refutation if there is one - as we have done in the algorithm in figure 3.4.

For EC-graph based propagation, assume we have the EC-graph $\mathrm{g_n}$ at node n and the graph $\mathrm{g_p}$ at its direct predecessor p. Our criterion for propagation from figure 3.4 is that any equality on a term represented in $\mathrm{g_n}$ be propagated - for each GT $\mathrm{u} \in \mathrm{g_n}$ and GT $\mathrm{v} \in \mathrm{g_p}$, the equalities in $\mathbf{eqs}(\mathrm{v})$ are relevant iff $\mathbf{terms}(\mathrm{u}) \cap \mathbf{terms}(\mathrm{v}) \neq \emptyset$.
We can send a request that contains all the GTs of $\mathrm{g_n}$, and return any GT of $\mathrm{g_p}$ that shares a term with any of the requested GTs - $\mathrm{g_n}$ then `assumes` $\mathbf{eqs}(\mathrm{v})$ for each such propagated GT. In the next section we show how to decide this criterion efficiently.

Comparison of the propagation criteria

In this sub-section we motivate our choice for using the EC-graph based propagation mechanism rather than superposition for unit ground clauses. The main reason for this choice is that join calculations, and especially incremental join calculations, are more efficient using EC-graphs than ground unit superposition. A second reason is that the representation of a propagation cache is more efficient with EC-graphs as we need to index one object for an entire EC, while for superposition we need to index each term separately, even if two terms were proven equal in the CFG-nodes where they are cached. We demonstrate these two reasons below.

The main difference between the two approaches is that superposition only considers one side of an equation for propagation but must import both sub-terms and super-terms, while the congruence closure (EC-graph) based approach considers both sides of an equation, but only imports sub-terms.

Consider the case where the predecessor clause set is
$\{\underline{\mathrm{d}} = \mathrm{a}, \underline{\mathrm{f(b)}} = \mathrm{e}, \underline{\mathrm{f(c)}} = \mathrm{a}\}$, and the clause set at n is $\{\underline{\mathrm{b}} = \mathrm{a}, \underline{\mathrm{d}} \neq \mathrm{b}\}$.
Superposition imports $\underline{\mathrm{d}} = \mathrm{a}, \underline{\mathrm{f(b)}} = \mathrm{e}$ while an EC-graph imports
$\mathrm{d} = \mathrm{a}, \mathrm{f(c)} = \mathrm{a}$ - only $\mathrm{d} = \mathrm{a}$ is actually needed, so both approaches imported (different) useless clauses.

Joins

For CFGs with joins, we cannot simply propagate a clause from the predecessor of a join to the join node, as the clause might not hold in the other joinee. Instead, we can only propagate equalities agreed by both joinees or relativized clauses, which are clauses guarded by the branch condition of the corresponding branch for the join. We have discussed the problem of joining two congruences in the previous section, and we show here an example that compares how EC-graphs and superposition differ in handling joins. The main outcome is that superposition requires more CFG-traversals in order to calculate a join, while the congruence closure based relevance criterion suffices for joins, when there is a join in the unit-ground fragment.

Saturating a set of equalities with superposition essentially establishes a rewrite relation for ground terms at every node which has a unique normal form for each ground term. The relation is only partially represented (by equalities)

at the node (similar to the congruence being approximated by an EC-graph), and when a term is requested in the CFG-node, we ensure that the relevant part of the rewrite relation is propagated from predecessors.
The main issue we have encountered is joins - the problem is, given two strongly normalizing rewrite relations which agree on an ordering, and given a term t, find the normal form of t at the intersection of the rewrite relations.
The intersection of the rewrite relations of two strongly normalizing rewrite systems that agree with the same total well founded ordering and are finitely generated is also strongly normalizing, by the simple argument that the normal form of a term t at the intersection is the minimum of the intersection of the equivalence classes of t in each rewrite relation, where the part of the equivalence class of t at the intersection that is smaller or equal to t is finite because that part of the rewrite relation is also finitely generated and agrees with a well founded ordering).

The critical difference between EC-graphs and superposition is that EC-graphs represent a fully-reduced rewrite system - the set of rewrite rules rewrite each term to its normal form in one step - while superposition constructs a left-reduced rewrite system - each term is the left-hand-side of exactly one rewrite rule, but the right hand side is not necessarily reduced (unit ground superposition constructs the left-reduced rewrite system when we can eliminate clauses using the simplification rule $\mathrm{simp}_=$ from figure 2.3 - for our purpose it is only important that the right hand side is not fully reduced) .
For example, the rewrite system $\{\underline{d} = c, \underline{c} = b\}$ is not fully reduced but is left reduced, while the (unique) fully reduced versions is $\{\underline{d} = b, \underline{c} = b\}$. A non-left-reduced system is, for example $\{\underline{d} = c, \underline{d} = b\}$, from which superposition (with simplification) derives $\{\underline{d} = c, \underline{c} = b\}$. With a fully reduced system, we can reach any member of the EC of a term t in two steps (two equations) - one step finds the normal form, and the second step uses any rule with the normal form as a right hand side. When the system is only left reduced, we might need more steps - e.g. for $d = c, c = b, b = a, e = a$ we need four steps to reach d from e. When looking for the normal form of a term at a join, we have to consider all members of the EC of that term on both joinees and select the minimum of the intersection - hence the number of steps needed to find this minimum is important.
Obviously, constructing a fully reduced rewrite system is costlier than a left-reduced one, but, in our experience, it pays off in more efficient join operations.
Consider the example in figure 3.29.
For superposition, we can propagate relativized (non-unit) clauses (assuming the joined branch condition C) as follows:

Initially, no CFG-node has any valid inference, and, except for n, n_a, no clauses to import - we look at the proof process in n.

Initially we request (rhs) $\{f(e), e\}$ and propagate the relativized clauses:
$\left\{\neg C \vee \underline{e} = d, C \vee \underline{f(e)} = a\right\}$

We use these to derive:
$\left\{\neg C \vee \underline{f(d)} = x, C \vee a = x\right\}$

```
b :
if (c1)
    p0 :
    assume $\underline{f(b)} = a$
    assume $\underline{c} = b$
    assume $\underline{d} = c$
    assume $\underline{e} = d$
    p1 :
    ... //unrelated clauses
else
    p2 :
    assume $\underline{f(e)} = a$
    p3 :
    ... //unrelated clauses
n :
assume $\underline{f(e)} = x$
// Here $f(e) = a$ holds
na :
    assert $\underline{a} = x$ //negated $\underline{a} \neq x$
```

Figure 3.29: Propagation condition comparison with joins

Next we request (rhs) $\{d, f(d), a\}$ and propagate:
$\{\neg C \vee \underline{d} = c\}$

Which allows us to derive:
$\left\{\neg C \vee \underline{f(c)} = x\right\}$

Next we request $\{c, f(c)\}$ and propagate:
$\{\neg C \vee \underline{c} = b\}$

Which allows us to derive:
$\left\{\neg C \vee \underline{f(b)} = x\right\}$

Next we request $\{b, f(b)\}$ and propagate:
$\left\{\neg C \vee \underline{f(b)} = a\right\}$

Which allows us to derive:
$\{\neg C \vee \underline{a} = x\}$

Finally, at na:
We request $\{a, x\}$ and propagate:
$\{\neg C \vee \underline{a} = x, C \vee \underline{a} = x\}$
We then derive, together with $a \neq x$:
$\{\neg C \vee x \neq x, C \vee x \neq x\}$ which allow us to find the refutation.

Note that we needed to traverse the CFG several times.
Note also that if we apply our simplification inference simp_{res} from figure 2.3 -

$$\frac{\cancel{\mathrm{C} \vee \mathrm{A}} \quad \cancel{\mathrm{C} \vee \neg \mathrm{A}}}{\mathrm{C}} \,,$$

we can derive:

$$\frac{\cancel{\mathrm{C} \vee \underline{\mathrm{a}} = \mathrm{x}} \quad \cancel{\neg \mathrm{C} \vee \underline{\mathrm{a}} = \mathrm{x}}}{\underline{\mathrm{a}} = \mathrm{x}}$$

And save some derivation steps. This simplification rule is designed exactly for such cases.

For EC-graph, we show in the next section a join algorithm that derive a = f(e) at the join, and can prove the program without any non-unit clauses. In addition, the superposition approach traversed the CFG several times, while our approach requires just one traversal.

Summary:
While both propagation criteria have advantages and disadvantages, and both propagate useless clauses, in our setting we have found the congruence closure based approach to be more suitable in the unit ground equality case. For larger fragments (non ground and/or non-unit) we will use a hybrid method, where the unit fragment will serve as a base. In this chapter we only discuss unit ground equalities using EC-graphs from now on, and in chapter 4 we handle joining non-unit clauses.

3.3.3 Ground unit equality propagation

In this section we describe the data structure and algorithm used to propagate ground unit equalities. This algorithm and data structure form the basis of all the other fragments we consider in this thesis.

We use the verification algorithm described in figures 3.25 and 3.26 with the following changes:

- An EC-graph is used to represent the set `done`
- Requests are represented as sets of GFAs over the EC-graph of the relevant CFG-node
- The request cache is a set of GFAs, which includes all GFAs in the EC-graph and an additional set of GFAs
- Each request is translated before being sent to the predecessors, so that the EC-tuple in the GFA is over GTs of the predecessor's EC-graph
- Instead of propagating relativized clauses from predecessors, each node performs a *meet* of its EC-graph with the join of the EC-graphs of the predecessors, for the requested GFAs (this is just an intuitive description - described in detail later)
- We maintain a links between each GT in the EC-graph of each CFG-node and the corresponding GTs of EC-graphs of predecessor CFG-nodes. Corresponding GTs are those that share a term. We use these links to translate requests quickly, and also to perform the join and meet

We describe first our data-structure and algorithm for sequential nodes (without any join) and in the next section show the needed changes to support joins.

Completeness

Our algorithm aims to propagate equality information in the CFG so that disequalities can be refuted and other logical fragments can use the equality information. Equality information should be propagated on-demand, so we have to define how equality information is requested. For EC-graphs, we request equality information by adding terms to the EC-graph. In general, we assume each CFG-node n has a set of requests $\mathrm{R_n}$ which are clauses (in our case, unit equalities).

An equality $\mathrm{s = t}$ holds at a CFG-node n ($\mathrm{n \models s = t}$) iff it holds on all paths leading to n, and it holds on a path P ($\mathrm{P \models s = t}$) if it is entailed by the clauses on the path ($\mathbf{C}_\mathrm{P}$).

We define soundness and completeness for any algorithm that, given a set of requests $\mathrm{R_n}$ per CFG-node, annotates each CFG-node with a set of clauses ϕ_n - in our algorithm from figure 3.25 this is the set $\mathrm{done_n}$, and for the ground unit equality fragment it is represented using an EC-graph.

An equality $\mathrm{s = t}$ is proved at a CFG-node n if $\mathrm{s = t \in \phi_n}$.

An annotation for a CFG-node n is sound if, for each CFG-node n and equality $\mathrm{s = t \in R_n}$, $\mathrm{s = t \in \phi_n \Rightarrow n \models s = t}$.

An annotation for a CFG-node n is complete if, for each CFG-node n, for each $s = t \in R_n$, $n \models s = t \Rightarrow s = t \in \phi_n$.
For EC-graphs, we use the notation g_n for the EC-graph of a node n and $[t]_n$ for the GT (EC-node in the graph) that represents t in g_n, if $t \in \mathbf{terms}(g_n)$.
We use the notation $g_n \models s = t$ to denote that the equality s=t is entailed by the graph g_n and that s, t are represented in g_n - formally $s, t \in \mathbf{terms}(g_n)$ and $[s]_n = [t]_n$. For a node n, $s = t \in \phi_n$ is defined as $g_n \models s = t$.

Equality information is requested for EC-graphs by adding the terms to the graph - so, by definition, $s = t \in R_n \Rightarrow s, t \in \mathbf{terms}(g_n)$.
Soundness for EC-graph annotations is, hence: $g_n \models s = t \Rightarrow n \models s = t$.
Completeness for EC-graphs is, for each $s = t \in R_n$, $n \models s = t \Rightarrow g_n \models s = t$.
In fact, the completeness definition is too strong for CFGs with joins, we use this definition for CFGs without joins and discuss a weaker condition for DAGs in the next section.
A CFG-node with exactly one predecessor is a sequential node while a CFG-node with two or more predecessors is a join node.

Our data-structure and algorithm are incremental in two ways:
We can run the algorithm with a certain set of requests R_n, and guarantee completeness for the resulting annotation (for join-less CFGs, otherwise a weaker guarantee). At this stage we can add more requests to R_n (that is, add more terms to the EC-graphs of some nodes) and the algorithm manipulates our data structure to ensure that the completeness guarantee is re-established.
The other option is to add more assumptions (invoke `assume(s=t)` for some equations $s = t$ in some of the CFG-nodes) in which case the algorithm again re-establishes the completeness guarantee (as the relation $n \models s = t$ might have changed for some n and $s = t \in R_n$). Both the above operations model the way other fragments communicate with our fragment.

Source links

In order to facilitate efficient equality propagation and efficient incremental updates, we maintain a link between each GT (EC-node) in the EC-graph of each CFG-node and the GTs of the EC-graphs of its direct predecessors that share a term with it. These links allow us to translate requests to predecessors and translate responses from predecessors, and also allow us to determine quickly if there is any relevant equality information in any of the transitive predecessors of a CFG-node.
We also use these links for incremental updates of CFG-nodes, where a stale link (an edge to a GT that has been merged with another GT) allows us to update the EC-graph of a CFG-node incrementally after the EC-graphs of direct predecessors have been updates (using the `mergeMap`).
The sources function is part of the state of an EC-graph which is maintained by our algorithm. The function returns, for a CFG-node n, a direct predecessor p and GT $u \in g_n$ a set of GTs.
We use the notation $\mathbf{sources}_\mathbf{n}^\mathbf{p}(u)$ for the source of $u \in g_n$ in the predecessor p.
When there is no ambiguity, we use $\mathbf{sources}_\mathbf{n}(u)$, $\mathbf{sources}^\mathbf{p}(u)$ or $\mathbf{sources}(u)$.
We extend the sources function to tuples, where the sources of a tuple are the Cartesian product of the tuple of sources. For example,
if $\mathbf{sources}([a, b]) = \{[a], [b]\}$
then $\mathbf{sources}(([a, b], [a, b])) = \{([a], [a]), ([a], [b])([b], [a])([b], [b])\}$.

```
n0 :
assume f(a) = g(a)
    // GFAs : a(), f([a]), g([a])
    // GTs : [a], [f(a), g(a)]
n1 :
assume a = b
    // GFAs : a(), b()
    // GTs : [a, b]
    // sources([a, b]1) = {[a]0}
n2 :
assert f(b) = g(b) //negated f(b) ≠ g(b)
    // GFAs : b()
    // GTs : [b]
    // sources([b]2) = {[a, b]1}
```

Figure 3.30: propagation sources
The state before n_2.`makeTerm(f([b]))`
We list the sources for each EC at each CFG-node.
The source invariant holds for the above example.

The intended invariant of the sources function states that the sources of a GT $u \in g_n$ are the GTs $v \in g_p$ s.t. u and v share a term - formally:
For a CFG-node n and a direct predecessor p,
$\forall u \in g_n \cdot$
$\quad \mathbf{sources}_{\mathbf{n}}^{\mathbf{p}}(u) = \{v \in g_p \mid \mathbf{terms}(u) \cap \mathbf{terms}(v) \neq \emptyset\}$
An example for the sources function is shown in figure 3.30.

We present now a formulation of this invariant that drives the way our algorithm establishes the invariant. The formulation is local in the sense that, for each GT, detecting a violation of the invariant or fixing a violated invariant involves traversing only a constant number of edges (both GFA and source edges). The above formulation does not satisfy locality as calculating the set **terms**(u) requires recursive descent in the structure of u of non-constant depth. In addition, we present the invariant as a conjunction of quantified implications. If the invariant does not hold for some assignment to the quantified variables, it is fixed by changing the right-hand-side of the implication. We present several such conjuncts in this section that together form the invariant established by our algorithm and are presented in a way that suggests how the algorithm establishes them.

The source invariant:
For a CFG-node n and a direct predecessor p.
$\forall u \in g_n, f(\bar{t}) \in u, \bar{s} \in \mathbf{sources}(\bar{t}) \cdot$
$\quad f(\bar{s}) \in \mathbf{gfas}_p \Rightarrow [f(\bar{s})] \in \mathbf{sources}_{\mathbf{n}}^{\mathbf{p}}(u)$

For example, consider the state in figure 3.31.
The figure shows the state of three consecutive CFG-nodes - n_0, n_1, n_2.
Source edges are marked in blue.
Consider the following assignment to the quantified variables:

The CFG-node n is n_1 and the predecessor p is n_0,
The GT $[f(a)]_1$ (assigned to u) contains the GFA $f([a, b])_1)$ (assigned to $f(\bar{t})$) where the tuple $([a]_0)$ (assigned to $\bar{s}$) satisfies $([a]_0) \in \mathbf{sources}_{\mathbf{n}}^{\mathbf{p}}([a, b]_1)$ and also $f([a]_0) \in \mathbf{gfas}_0$ - hence the invariant implies that there is a source edge between the GTs $[f(a)]_1$ and $[f(a)]_0$ ($[f([a]_0)]_0 \in \mathbf{sources}_{\mathbf{n}}^{\mathbf{p}}([f([a])]_1)$).
In our example the invariant holds. The main loop of the algorithm fixes such local inconsistencies (that arise from node GT merging and other operations). In our example, if the condition does not hold during the run of our algorithm (that is, there was no source edge from $[f(a)]_1$ to $[f(a)]_0$), our algorithm would establish the invariant locally by adding that edge.

3.3.4 Propagation using sources

We show now show how the sources function is used for information propagation. In the example in figure 3.30, we have not yet added the terms $f(b), g(b)$ to g_2 and hence we are missing some equality information at n_2 in order to prove the assertion (namely, $n_2 \models f(b) = g(b)$ - but $g_2 \not\models f(b) = g(b)$.

In order to ensure that the information is propagated, we use a local propagation invariant, that works together with the source correctness invariant to ensure that enough information is propagated. The local propagation invariant for sequential nodes (with one predecessor) ensures that, for an CFG-node n and a GT $u \in g_n$, u has all the terms of all its sources - formally:

The sequential propagation invariant, part 1:
For a sequential CFG-node n and a direct predecessor p.
$\forall u \in g_n, v \in \mathbf{sources}(u), f(\bar{s}) \in v \cdot$
$\quad \exists f(\bar{t}) \in u \cdot \bar{s} \in \mathbf{sources}(\bar{t})$

We use the notation $f(\bar{t}) \in u$ to denote that the GFA $f(\bar{t})$ is in the `gfas` field of the GT u.
The invariant states that each GFA in each source of u has a corresponding GFA in u. This invariant is maintained by our algorithm and ensures that the set of terms of each GT is a superset of the union of sets of terms of all its sources. Note that as our EC-graph is kept congruence closed, this invariant implies that no two GTs share the same source (in a sequential node) - otherwise two GTs that share a source would share a GFA.

In our example in figure 3.30, we can see that this invariant does not hold - the EC $[b]_2$ does not contain the GFA a(). Our algorithm fixes the invariant when invoking `makeTerm(b, ())`, so that essentially the equality $a = b$ is propagated eagerly. We detail in the next section how this invariant is established, for now we just assume that whenever a GT is missing some GFAs according to our invariant, the algorithm adds the missing GFAs. Note that the invariant is phrased in terms of the sources function and refers only to GTs that are a constant distance from each other (in terms of GFA and source edges).

We now demonstrate how equalities are actually propagated by our algorithm. Consider the example in figure 3.32a.
We describe how our algorithm performs n_1`.makeTerm(f, (`$[a]_1$`))`, ensuring that $g_1 \models f(a) = g(a)$.

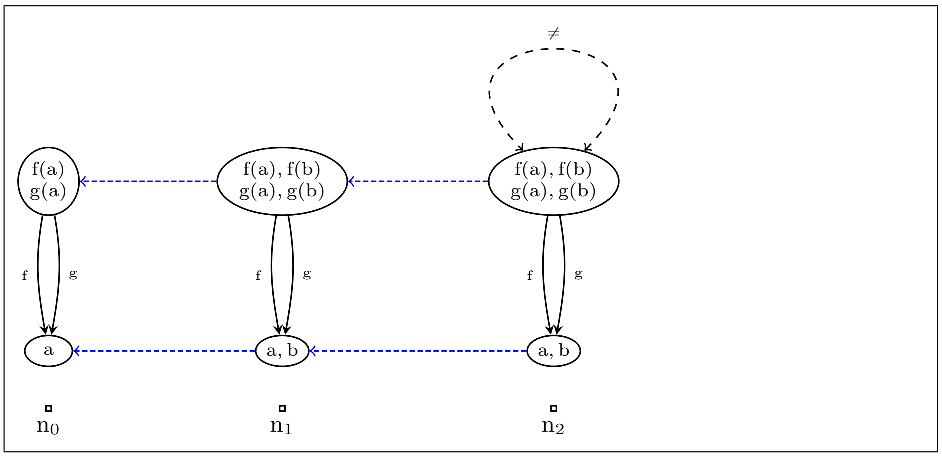
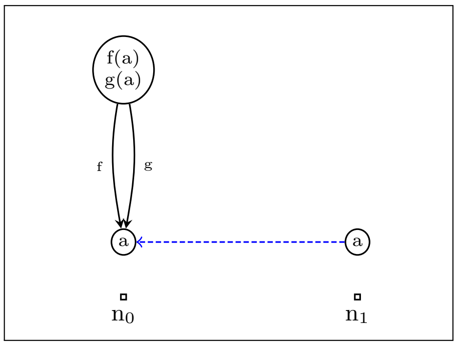


Figure 3.31: Source edges
We omit tuples as we only use unary functions.
Dashed arrows represent dis-equalities.
Blue dashed arrows represent source edges.

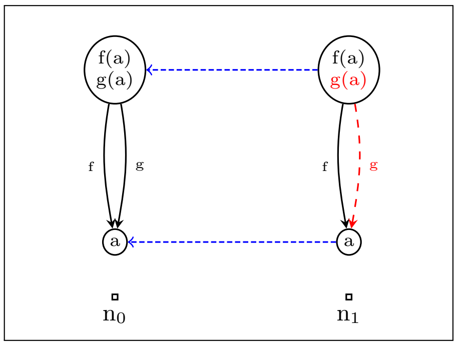
First, the GFA $f([a]_0)$ is searched in the `superTerms` field of n_1, and not found. As the GFA does not yet exist (the term f(a) is not represented in g_1), we create a new GT that includes that GFA and map it in the `superTerms` field as a super-term of $[a]_0$. We also add a source-edge to the new GT, as mandated by the source invariant.
The state is shown in figure 3.32b, we mark in red the missing parts of the EC-graph that are needed in order to propagate the equality $f(a) = g(a)$.
For each source-edge added to a GT, our algorithm looks at each GFA of the new source, and adds the missing ones (in order to establish the propagation invariant).
In our case, the GFA $f([a]_1)$ is present but the GFA $g([a]_1)$ is missing.
We look at the source GFA $g([a]_0)$ and look at the inverse source of the tuple $([a]_0)$ - in this case $([a]_1)$.
We add the GFA $g([a]_1)$ to our new GT and we are done. The result is shown in figure 3.32c.
In the next example, we show what happens when the inverse source of the tuple is empty.

Consider the state in figure 3.33a.
We add the GT for h(a) - shown in figure 3.33b.
We now proceed as before, looking for an inverse source for $[f(a)]_0$ - none is found.
In this case we add a new empty GT, and attach it with a source-edge to $[f(a)]_0$.
The result is shown in figure 3.33c.
This GT has no members yet, but it has a source - so the propagation invariant forces us to look for an inverse source for the GFA $f([a]_0)$ with which propagation is complete - shown in figure 3.33d.

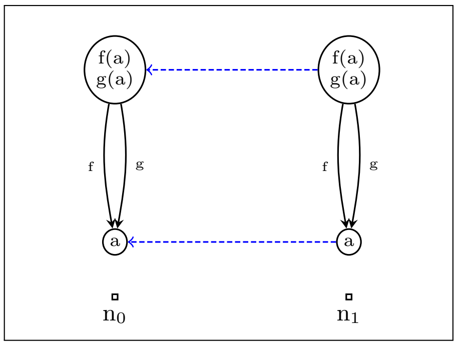
The process of completing GFAs helps us ensure that all relevant equality information is propagated, and only relevant equality information (by the congruence



(a) Before n_1.`makeTerm(f, ([a]_1))`



(b) After adding $f([a]_1)$
$g([a]_1)$ is missing



(c) After adding $g([a]_1)$
Invariant is satisfied

Figure 3.32: Example for the propagation invariant
Missing parts of the graph are in red

closure propagation criterion - we propagate all equality information for sub-terms). This is the implementation of the algorithm in figure 3.4.
The algorithm adds at most one GT per predecessor GT.
We note here that, in some cases, we may create empty GTs as above and they will never become actual GTs (will never represent any term). This can happen in the case of joins, and also for bounded fragments and scoping - we give an example when discussing joins.

In the above description we have implicitly assumed one important property of source-edges - namely, that each GT can have at most one source - formally:

The sequential propagation invariant part 2:
For a sequential node n and a predecessor p.
$\forall \mathrm{u}, \mathrm{v} \in \mathrm{g_n} \cdot$
$\quad \mathbf{sources_n}(\mathrm{u}) \cap \mathbf{sources_n}(\mathrm{v}) \neq \emptyset \Rightarrow \mathrm{u} = \mathrm{v}$

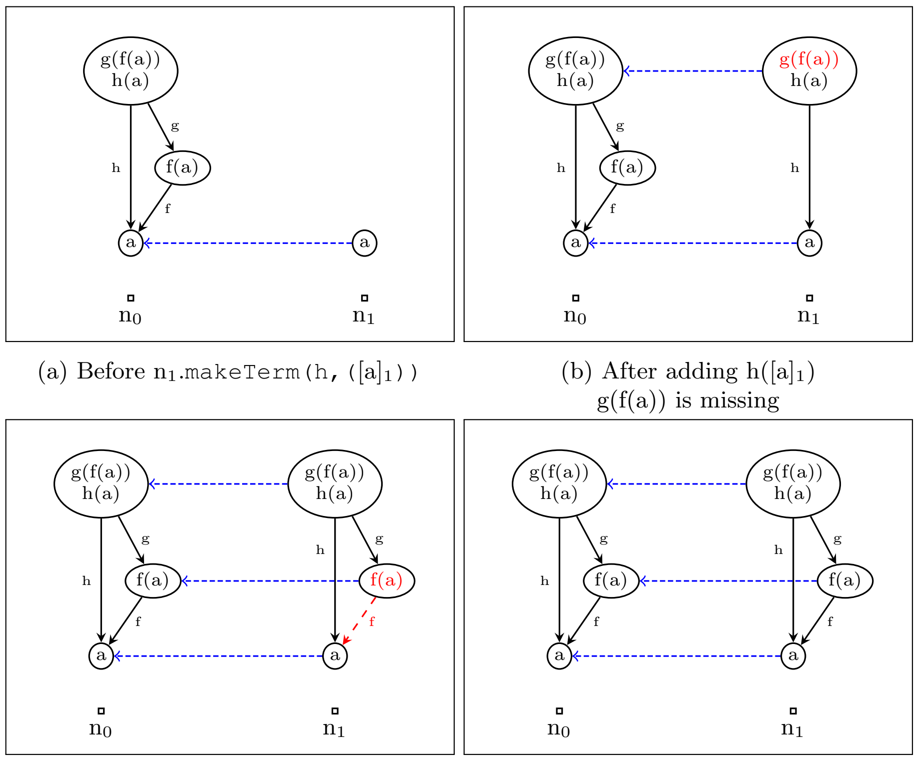
This invariant comes up whenever one of the other invariants forces us to add a source-edge - if that source already has an inverse-source, instead of adding another inverse source we merge the two GTs. Note that the first part of the propagation invariant is insufficient here, as the empty GT has no GFAs to operate on. In the above case that would not be a problem, but if the predecessor had a cycle in the EC-graph - e.g. $\mathrm{a} = \mathrm{h(a)}$ - we could add an unbounded number of empty GTs, while part two of the propagation invariant (when enforced eagerly) prevents that as we can add at most as many GTs as there are in our predecessor graph.

The request cache

In order to reduce the number of traversals of the CFG, we cache at each CFG-node the previous requests, so that no CFG-node propagates the same request twice. The cache for a CFG-node n consists of all the GFAs of $\mathrm{g_n}$ - $\mathbf{gfas}_\mathrm{n}$ - together with another set of GFAs, $\mathbf{rgfas}_\mathrm{n}$. We call a member of $\mathbf{rgfas}_\mathrm{n}$ an RGFA (A rejected GFA). For each GFA in the cache $\mathrm{f}(\bar{\mathrm{t}}) \in \mathbf{gfas}_\mathrm{n} \cup \mathbf{rgfas}_\mathrm{n}$, the members of the EC-tuple $\bar{\mathrm{t}}$ are all in $\mathrm{g_n}$ - $\bar{\mathrm{t}} \in \mathrm{g_n}$ - this means that we can only cache direct super-terms of terms represented in $\mathrm{g_n}$. The idea is that if a request for a term t has returned an empty response, a request for any super-term of t will also be empty.

Consider the example in figure 3.34.
The example shows the state before we invoke `makeTerm(f,([b]))` on $\mathrm{n_2}$ (and before adding $\mathrm{f(b)}, \mathrm{g(b)}$ to $\mathrm{n_3}$).
The initial state (excluding $\mathrm{n_3}$) is shown in figure 3.35a.

We now show the operation of $\mathrm{n_2}$`.makeTerm(f([b]))`:
As in the previous example, $\mathrm{n_2}$ sends a request for $\mathrm{f([b]_2)}$ which gets translated down the line to $\mathrm{f([b]_0)}, \mathrm{f([a]_0)}$.
In this case, $\mathrm{n_0}$ has no information about either of the requested GFAs, and has no predecessors, so it adds the GFAs to the cache by adding an RGFA for each.
For $\mathrm{n_1}$ the situation is the same - the predecessor has no information about the



(a) Before n_1.`makeTerm(h, ([a]`$_1$`))`

(b) After adding h($[a]_1$) g(f(a)) is missing

(c) After adding an inverse source for g(f($[a]_0$))

(d) Final state

Figure 3.33: Example for deep propagation

```
n_0 :
assume b = b
    // GFAs : b()
    // ECs : [b]
    // rgfas : ∅
n_1 :
assume b = b
    // GFAs : b()
    // ECs : [b]
    // sources([b]_1) = {[b]_0}
    // rgfas : ∅
if (*)
    n_2 :
    assert f(b) = b //negated f(b) ≠ b
        // GFAs : ∅
        // ECs : ∅
else
    n_3 :
    assert f(b) = g(b) //negated f(b) ≠ g(b)
        // GFAs : b()
        // ECs : [b]
        // sources([b]_2) = {[b]_1}
```

Figure 3.34: propagation sources
The state before n_2.`makeTerm(f([b]))`
The local source correctness invariant holds for the above example.

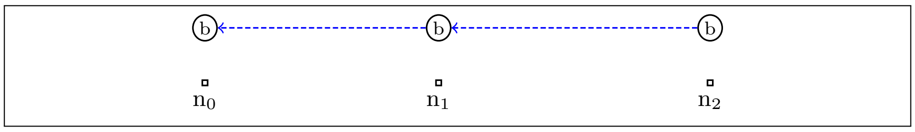
request, so we add it to the cache.
Finally, n_2 adds the GFA as a singleton GT.
The final state, after adding also n_2.`makeTerm(g([b]))`, is shown in figure 3.35b.

After n_2.`makeTerm(f([b]))`, we also invoke
n_2.`makeTerm(g([b]))` adding the corresponding GFAs as in the previous case.
Now we are done with n_2.

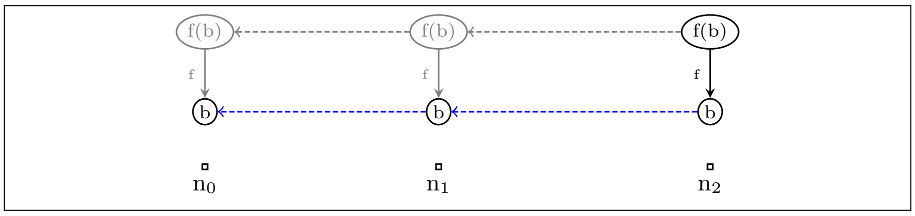
When n_3 performs n_3.`makeTerm(f([b]))`:
The request sent to n_1 is $\{f([b]_3\}$ which is translated to $\{f([b]_1)\}$.
As n_1 has this RGFA, we subtract it from the request and now the request is $\emptyset$.
The request is not propagated further and we are done with n_3.

Incremental updates

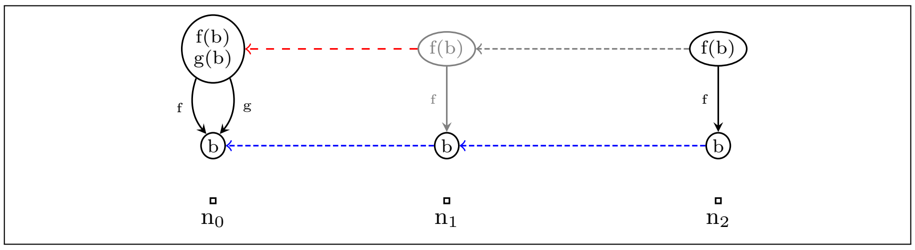
Assume that we have performed the operations for our last example as described, and completed one pass of `CFG.verify()` with the EC-graph fragment. Now another logical fragment has produced the equality $f(b) = g(b)$ at n_0, for example using quantifier instantiation.
The state is depicted in figure 3.35c.
We can see that now there is information about $f(b)$ at n_0 that was not propagated to where it is needed at n_2 (and n_3). We call such a state inconsistent,



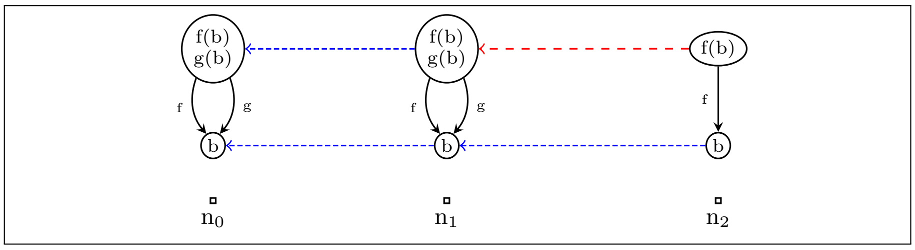
(a) The initial state for figure 3.34



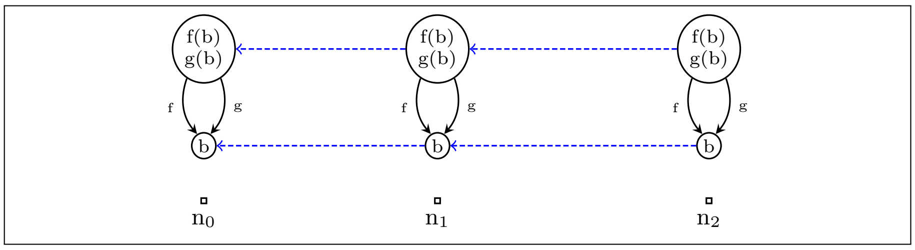
(b) The state after n$_2$.`makeTerm(f,([b]))` in 3.34



(c) The state after n$_0$.`assumeEqual(f(b),g(b))` in 3.34.
Red dashed arrows represent inconsistent sources - RGFA to GFA
(only for illustration, not actually included in data structure)



(d) The state after n$_1$.`update`.
n$_1$ is now consistent - but n$_2$ is not.



(e) The state after n$_2$.`update`.
All nodes are now consistent

Figure 3.35: .
Gray dashed circles represent RGFAs.
Gray dashed arrows represent source edges to RGFAs
(only for illustration, not actually included in data structure)

and we say that locally, n_1 is inconsistent (shown in red) - on the next pass of verification for our fragment (which again traverses the CFG in topological order), our algorithm invokes an `update` on each CFG node before invoking `Node.verify`. The `update` method, when invoked on a CFG-node all of whose predecessors are consistent, ensures that CFG-node is also consistent (and does not break consistency for predecessors, although it might for successors). We do not `update` all successors eagerly on every change as it is extremely inefficient - instead, when traversing the CFG in topological order for any fragment, we `update` each CFG-node before performing any inferences on it.

In order to support an efficient incremental `update` method, each CFG-node keeps some history information that allows it to summarize to successors the changes to its EC-graph since the last time the successor was updated.
The history is kept mainly in two fields:

- The `mergeMap` field that was used for merging, including an inverse map
- A map that assigns a `generation` for each GT. Roughly, each time `assumeEqual` or `update` are called on a CFG-node, a new generation starts. The idea is that if a GT u exists in the EC-graph at some generation, if u is merged into v then the merged node is of a strictly higher generation. Each CFG-node remembers the last generation of its predecessors for which it is up-to-date, and only considers newer GTs when invoking `update`

In our example, $[b]_0$ is of generation 1 while $[f(b)]_0$ is of generation 2. The last predecessor generation of n_1 is 1 (that is, n_1 has all the information of generations up to 1 from predecessors). When n_1 invokes n_1`.update`, it requests a list of changes from n_0 later than generation 1, and receives the list that includes only the GT $[f(b)_0]$. The `update` method collects all changes from predecessors.
For each new or updated predecessor GT, we check if the corresponding GT is in our cache (using the inverse source-edges) - in our case, for the new GT $[f(b)]_0$, the inverse source of the tuple $([b]_0)$ is $([b]_1)$. We check if we have the GFA $f([b]_1)$ in the cache and we find we have it as an RGFA. Hence we replace the RGFA with a GT that contains the GFA $f([b]_1)$, add the source-edge and propagate equality information. The state after n_1`.update` is shown in figure 3.35d.
For n_3, the operation of the `update` method is similar, except that here the updated predecessor GT matches a GT rather than an RGFA - hence we simply add the corresponding source-edge and propagate equalities. The final state is shown in figure 3.35e

We formalize now two parts (conjuncts) of the local CFG-node invariant that ensures that an RGFA exists in CFG-node only if none of the terms it represents occurs in all its predecessors. The first part is the part that is broken in figure 3.35c (marked in red), where the RGFA $[f(b)]_1$ should instead be added to a GT as n_0 has the GT $[f(b)]_0$.

For a CFG node n with one prededessor p.
$\forall \overline{t} \in g_n, \overline{s} \in \mathbf{sources_n}(\overline{t}), f \cdot$
$\quad f(\overline{s}) \in \mathbf{gfas}_p \Rightarrow f(\overline{t}) \notin \mathbf{rgfas}_n$
And
$\forall f(\overline{t}) \in \mathbf{gfas}_n \cup \mathbf{rgfas}_n, \overline{s} \in \mathbf{sources_n}(\overline{t}) \cdot$
$\quad f(\overline{s}) \in \mathbf{gfas}_p \cup \mathbf{rgfas}_p$

The first part of the invariant prevents the situation in figure 3.35c - its violation is fixed by replacing the RGFA $[f(b)]_1$ with the GT that includes the GFA $[f(b)]_1$ (the algorithm never removes an RGFA without replacing it with a GFA - the only exception is in a very specific garbage collection process).
The second part of the invariant ensures that if some term is in the request cache ($\mathbf{gfas}_n \cup \mathbf{rgfas}_n$), then it is also in the request cache of the predecessor. This invariant is enforced by sending a request for the offending GFA, as we have shown above for invocations of `makeTerm`. The second part of the invariant can also be violated when a new source-edge is added or when two GTs are merged.

3.3.5 The equality propagation algorithm

We have seen now the main points of the algorithm and the reasoning behind them, and now we present the algorithm for sequential nodes in more detail. In the next section we present the changes needed in order to support joins.
The high-level view
The main part of the algorithm is shared between `makeTerm`, `assumeEqual` and `update`.
For each operation, we modify or add some GTs in the EC-graph and add them to a queue of modified GTs.
Our main loop maintains two queues - the `mergeQ` we have seen for monolithic EC-graphs for GTs that need to be merged and the `propQ` (propagation queue) that contains all GTs that have been modified, where modifications include the addition of GFAs and source-edges. The main loop of the algorithm simply processes both queues and propagates changes from each GT to its neighbours as needed, until the graph is consistent - all conjuncts of our invariant are satisfied. Changes include adding GFAs or sources to GTs, and merging GTs.

For `makeTerm` and `assumeEqual`, we assume that in the pre-state all parts of the invariant hold, we add information to the graph which might violate the invariant, and then the main loop of the algorithm fixes the broken part.
For `update`, we assume that all predecessors of the node are consistent, and furthermore we assume that the predecessors have changed only monotonically (adding information) since the last time our node was up-to-date.

`makeTerm`
The algorithm for `makeTerm` is presented in figure 3.36, it is an extension of the algorithm for monolithic EC-graphs (which simply adds a singleton GT if it does not already exist).
The algorithm first checks if the term already exists (using `findGT`), in which case it is simply returned. The check is at the cost of a constant number

```
method makeTerm(f:Function, tt : Tuple[GT]) : GT
   t:=addGT(f,tt)
   mainLoop
   return transitiveMerge(t)

method addGT(f:Function, tt: Tuple[GT) : GT
   var t := findGT(f,tt)
   if (t!=null)
      return t
   t := addNewSinlgeton(f,tt)

method findGT(f:Function,tt:Tuple[GT]) : GT
   t := null
   if (tt=())
      if (f∈constants.keys)
         t := constants[f]
   else
      t := superTerms[tt[0]].findOrNull(t ⇒ f(tt)∈t.gfas)
   return t

method addNewSingleton(f,tt)
   t := new GT(new GFA(f,tt))
   if (tt=())
      constants[f] := t
   else
      foreach (ti∈tt)
         superTerms[ti].add(t)
   rgfas.remove(f(tt))
   propQ.enqueue(t)
```

Figure 3.36: The algorithm for makeTerm
The method checks if the GFA already exists, and otherwise creates a new singleton GT and invokes the main loop

of searches in maps (we simplify here the implementation of the superTerms field). The addGT method enqueues the newly added GT in propQ.

assumeEqual
The algorithm for assumeEqual is shown in figure 3.37, it is an extension of the algorithm for monolithic EC-graphs. The method simply enqueues a merge between the two input GTs.

update
The algorithm for update is shown in figure 3.38.
The algorithm receives a set of new GTs in the predecessor (since the last generation in which we have sampled the predecessor), and a list of predecessor GTs that have new information (that is, have added GFAs since the last sampled generation).
The method searches the inverse source of each updated GT and, if found, updates the source edge (using the predecessors mergeMap) and enqueues the relevant GT.

```
method assumeEqual(gt1,gt2:GT)
   if (gt1==gt2)
      return

   enqueueMerge(gt1,gt2)

   mainLoop
```

Figure 3.37: The algorithm for `assumeEqual`

```
method update(updatedGTs : Set[GT], newGTs : Set[GT])

   foreach pgt∈updatedGTs
      if sources$^{-1}$.hasKey(pgt)
         gt := sources$^{-1}$[pgt]
         removeSource(gt,pgt)
         addSource(gt,predecessor.transitiveMerge(pgt))

   foreach pgt∈newGTs
      foreach f(ptt)∈pgt.gfas
         if sources$^{-1}$.hasKey(ptt)
            tt := sources$^{-1}$[ptt]
            if rgfas.conatins(f(tt))
               gt := addSingleton(f,tt)
               addSource(gt,pgt)
            else
               gt := findGT(f,tt)
               if (gt!=null)
                  addSource(gt,pgt)

   mainLoop
```

Figure 3.38: The algorithm for `update`
The inputs are the set of predecessor GTs updated (that is, that have been merged with another GT or have had a GFA added) and the set of new GTs added to g_p.

For new GTs, we only check if there is a new predecessor GT that has an inverse source GFA or RGFA in our graph - in which case we add it as a source (converting an RGFA to a singleton GT) and enqueue the corresponding GT.

The main loop:
The code for the main loop is shown in figure 3.39.
For each GT, we maintain a list of new GFAs and new or updated source edges. The main loop basically merges GTs until there are none left to merge and then propagates information from each merged or modified GT. The method `mergeOne` merges two GTs as we have seen before in figure 3.3. The only difference is that each GFA or source-edge added to the merge target is marked as new, and the merge target is added to `propQ`.
The method `propagateOne` propagates all latest changes to a GT to its adjacent GTs - it updates the source-edges of the GT and propagates any new source

```
method mainLoop
   while (!mergeQueue.isEmpty || !propQ.isEmpty)
      while (!mergeQ.isEmpty)
         mergeOne(mergeQ.dequeue)
      while (!propQ.isEmpty)
         propagateOne(propQ.dequeue)

method propagateOne(gt:GT)
   if (mergeQ.contains(gt))
      return

   while (!gt.newGFAs.isEmpty || !gt.newSources.isEmpty)
      updateForNewGFAs(gt)
      completeDownNewSources(gt)

   propagateUpNewSources(gt)
```

Figure 3.39: The algorithm for `mainLoop`
The algorithm processes both queues until no work is left. Merging is done as shown for the monolithic EC-graph.

information to super-terms (up). Added source may force us to add new GFAs, which may, in turn, force us to add new sources edges - hence the loop.
The method also propagates source information for super-terms from new source edges.

The pseudo-code for the incremental update of a GT is shown in figure 3.40. `updateForNewGFAs` first ensures that all required equality information is propagated to the predecessor (we describe requests below), and then adds all relevant source-edges, for example, in figure 3.33b, the source edge between $[h(a)]_1$ and $[h(a)]_0$ is added after the GFA $h([a]_1)$ is added to the new GT.

The `propagateUpNewSources` method looks at all cached requests that are direct super-terms of the GT with an added source, and ensure first that all relevant equality information for these super-terms is added to the predecessor. Next, we add new source-edges to super-terms, and in the case the super-term is an RGFA, we convert it to a GT - as described in the example in 3.35d.

Requests:
The algorithm for servicing a request is shown in figure 3.42.
The algorithm is basically an extension of the generic algorithm from figure 3.26. The algorithm traverses the CFG in reverse topological order starting at n, where the request (a function symbol and set of tuples of GTs) is filtered through the cache at each CFG-node (in `requestBW` - any GFA that already exists in g_n or $\mathbf{rgfas}_n$ is removed), and then translated through source-edges to the predecessor.
The predecessor is traversed only if the filtered translated request is not empty - this is the key to reducing CFG-traversals.
We then traverse the CFG in topological order (`requestFW`), starting at each CFG-node that sent no request to its predecessor (or the root), and ending at n.
Each CFG-node adds to its cache all the requested GFAs - those for which there are no sources in the predecessor are added to $\mathbf{rgfas}_n$ (ensuring this request is

```
method updateForNewGFAs(gt : GT)
   foreach f(tt)∈gt.newGFAs //only process new GFAs
      var pgts := predecessor.requestGTs(f,sources[tt])
      addSources(gt,pgts)

method completeDownNewSources(gt)
   foreach pgt in gt.newSources //only new sources
      foreach f(ss)∈pgt.gfas
         var tt := makeInverseSource(ss)
            addGFA(gt,f,tt)

method propagateUpNewSources(gt)
   foreach pgt in newSources[gt] //only new sources
      foreach f(tt)∈superGFAs(gt)∪rgfas[gt]
         var pgts := predecessor.requestGTs(f,sources[tt])
         var gt:=findGT(f,tt)
         if (gt!=null)
            addSources(gt,pgts)
         else if (!pgts.isEmpty)
            addSingleton(f,tt)
```

Figure 3.40: The algorithm for propagating changes from a GT
New sources may add new GFAs and vice-versa.
After all GFAs and sources are added,
we propagate source information to super-terms.
The methods `addSource` and `addGFA` are detailed in figure 3.41.

```
method makeInverseSource(pgt:GT) : GT
   var gt:=sources$^{-1}$[pgt]
   if (gt==null)
      gt := new GT()
      addSource(gt,pgt)
   return gt

method addSource(gt:GT,pgt:GT)
   sources[gt].add(pgt)
   sources$^{-1}$[pgt].add(gt)
   propQ.enqueue(gt)

method addGFA(gt,f,tt)
   gt2 := findGT(f,tt)
   if (gt2!=null)
      equeueMerge(gt,gt2)
   else
      gfa := new GFA(f,tt)
      gt.gfas.add(gfa)
      foreach (gt2 in tt)
         superTerms[gt2].add(gt)
      propQ.enqueue(gt)
```

Figure 3.41: Helper functions for the propagation algorithm

```
method requestGTs(f,tts:Set[Tuple[GT]]) : Set[GT]
   var requestMap := new Map[Set[Tuple[GT]]]
   requestMap[this] := (f,tts)

   traverseBF(this,requestBW,requestFW)

   return findGTs(f,tts)

method requestBW(n:CFGNode) : Set[CFGNode]
   (f,tts) := requestMap[n]
   //The request for the predecessor
   var ptts := new Set[Tuple[GT]]
   foreach tt∈tts
      t := findGT(f,tt)
      if (t==null)
         if (f(tt)∉rgfas)
            ptts.add(sources[tt]) //request not in cache

      if (ptts.isEmpty)
         return ∅ //no request to predecessor
      else  //propagate request to predecessor
         requestMap[predecessor].add((f,ptts))
         return Set(predecessor)

method requestFW(n:CFGNode)
   (f,tts) = requestMap[n]
   foreach tt∈tts
      var pgts := predecessor.findGTs(f,sources[tt])
      if (!pgts.isEmpty)
         makeTerm(f,tt)
      else
         rgts.add(f(tt))
```

Figure 3.42: The algorithm for `requestGTs`
The method `requestGTs(f,tts)` propagates equality information for all terms in f(tt) for tt∈tts.

never again propagated from this CFG-node) and the others are added to g_n as new GTs using the `makeTerm` method (this invocation of `makeTerm` will send a new request for the same set of GFAs, but that request is answered immediately as the response is cached in the predecessor - however, it may send further new requests triggered by congruence closure - we discuss this case below).

An example

We describe now an example that shows that the main loop of our algorithm may need several up and down propagation in order to ensure consistency. Consider the state in figure 3.43a.
We now perform n_1`.makeTerm(f,(`$[a]_1$`))`.
The steps are shown in figures 3.43a to 3.43d and 3.44, annotated with the

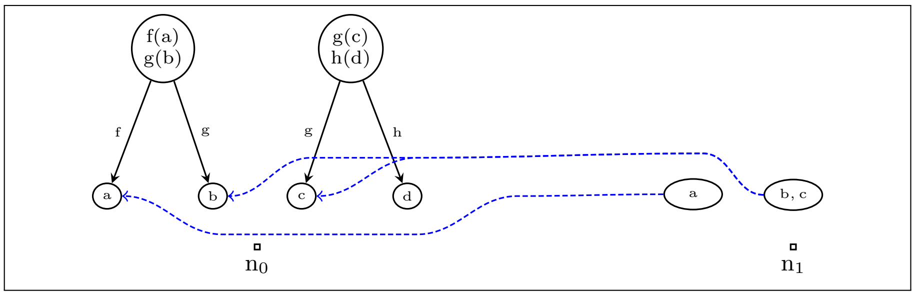
method in the algorithm used for each step. We can see why we have a loop in `propagateOne` - we may need several steps until one GT is consistent.

In the case where n_0 is a transitive predecessor rather than a direct predecessor (assuming no other equlities on intermediate nodes), the process will be similar, except that each time we draw a new source-edge, we send a request to predecessors that propagates all relevant equality information to the direct predecessor. In the case of deeper terms, for example, if we replace, in the example above, $g(b), g(c)$ with $g(f(b)), g(f(c))$ resp. - the process is again similar except that the loop between updating source-edges and completing GFAs happens accross more than one GT and hence more than one call to `propagateOne`.

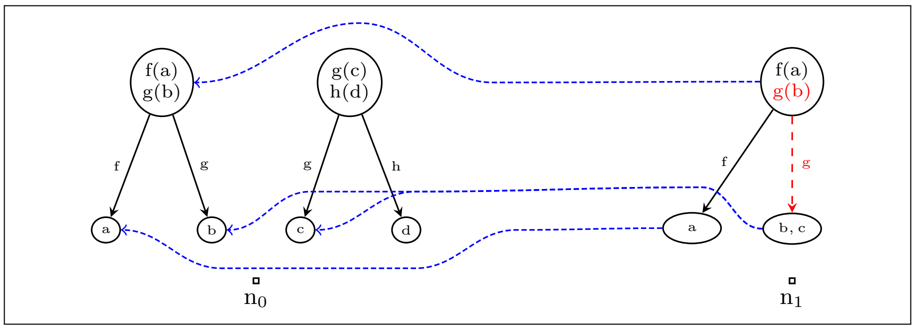
Algorithm properties

We have sketched an algorithm for the incremental, on-demand propagation of equality information in a CFG without joins. The condition the algorithm aims to satisfy is that, for a consistent CFG (that is, where all CFG-nodes are consistent w.r.t. the invariants we have shown above), for any CFG-node, any two terms that are represented in the EC-graph of the node are equal in the EC-graph iff it holds at the node that they are equal -
$\forall n \in cfg, s, t \in \mathbf{terms}(g_n) \cdot n \models s = t \Leftrightarrow [s]_n = [t]_n$.
We have not proved this completeness property formally, our informal argument runs as follows:
For a pair of consecutive CFG-nodes, the property holds by strong induction on the maximal depth d of s, t - the induction hypothesis is that, for all terms smaller than d, both the source edges are correct (that is, there is a source edge iff there is an intersection in the represented terms) and that $n \models u = v \Rightarrow [u]_n = [v]_n$ holds for all u,v of depth up to d. Together with the property that each EC-graph is congruence and transitive closed, and that we merge two GTs that share a source (essentially transitive closure) and together with the invariant conjuncts (ensuring the propagation of all GFAs and the addition of source-edges), we should be able to show that the induction property holds also for depth d.
For a CFG of depth more than two (linear or tree-shaped), we need another property, essentially that if a term is represented at some CFG-node and is also represented by some transitive predecessor, then it is represented on all the path between the node and the transitive predecessor. This property is ensured by our request cache - each term represented at a node is either represented in the direct predecessor, one of its sub-terms is not represented, or it is represented by some RGFA in the predecessor. Because we disallow an RGFA if it has a GT source, strong induction on both the term depth and the length of the source-edge chain should show that there is a chain of source-edges between each pair of nodes with the same term represented, that are on the same CFG-path. A formal proof of the above remains as future work.

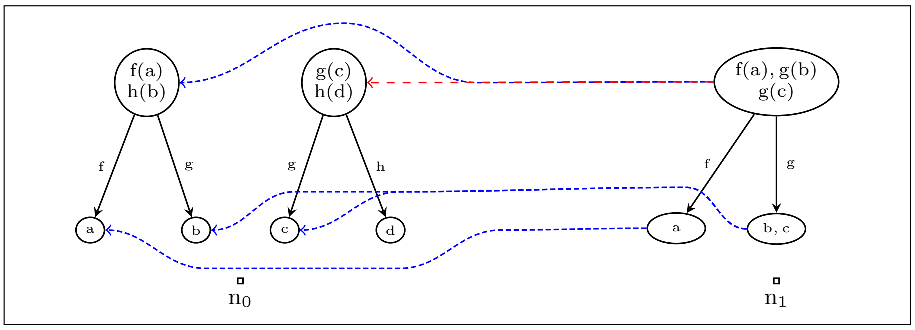
Another important property is incrementallity - no operation (congruence closure, comparison of function symbols) happens twice for the same inputs, unless one of the participants of the operation has changed. We achieve this property by marking any change in the structure and only operating on modified parts of the structure. We also ensure that equality information is propagated eagerly from predecessors, so that the same congruence closure operation is not performed at a CFG-node and any of its successor. We have not performed



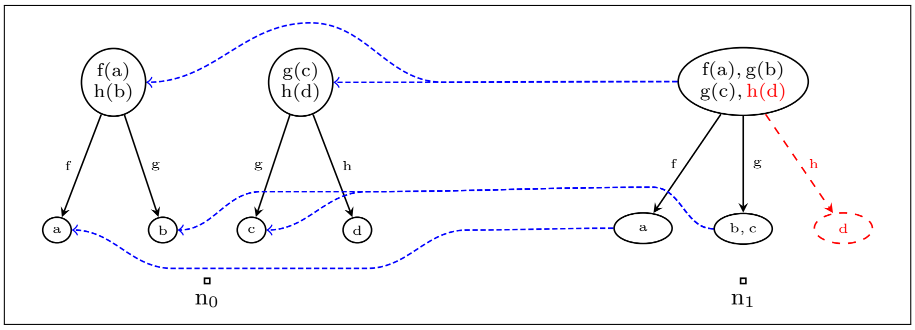
(a) The state before n_1.`makeTerm(f,(a))`



(b) The state after `makeSingleton(f,`$[a]_1$`)`



(c) The state after `completeDownNewSources`



(d) The state after `updateForNewGFAs`

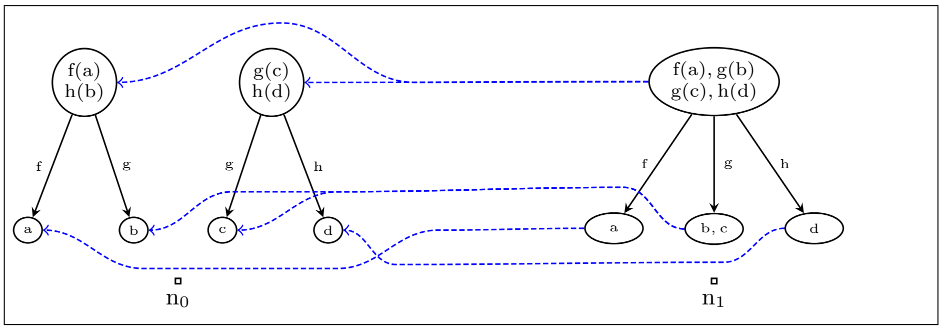


Figure 3.44: The final state

a formal complexity analysis on the algorithm, and some finer detail of the algorithm are not shown above (e.g. how to find the inverse sources of tuples and how to represent requests that are Cartesian products) can affect the worst case complexity.
However, we can see that the total size of each EC-graph is at most the sum of sizes of all the clauses in all its CFG-node's predecessors (no operation in our algorithm adds a GFA that does not come from some input clause) - which suggests a quadratic space complexity and between quadratic and cubic time-complexity (with some logarithmic factor from table lookups).

In the chapters 5 and 6 we show two improvements on the algorithm, namely restricting the set of constants allowed at each CFG-node and restricting the depth of represented term. Both restrictions are enforced at the construction level of the algorithm and so hold at any intermediate state, and allow us to give a strict space bound on the algorithm. Completeness can be preserved in the first case by generating some non-unit clauses (as we show) and in the second case we can incrementally increase the allowed depth to allow completeness.

3.4 Joins

We have seen, in the previous section, our propagation algorithm for sequential CFG-nodes. In this section we show the necessary changes to make the algorithm apply for CFG-nodes with two predecessors, where the semantics is that we can only propagate an equality to the join node if it is implied by both predecessors.

As opposed to the sequential case, for joins we cannot guarantee that any equality that holds at a CFG-node on two represented terms also holds in the EC-graph, simply because not all interpolants are representable as a finite conjunction of equalities - as we have seen e.g. in figure 3.11 - the interpolant between $\{c = a\} \sqcup \{c = b\}$ and $\{a = b \wedge a \neq c\}$ cannot be represented as a finite conjunction of unit equations. Furthermore, we have seen that even in the cases where a unit interpolant exists, the smallest such interpolant may be exponential in the program size, even in the sub-fragment that does not allow cycles in EC-graphs. In light of the above, our algorithm for joins guarantees only a weaker property - we guarantee, for each join, that any pair of terms represented in both joinees and at the join, are equal at the join EC-graph iff they are equal at both joinees. The guarantee for source-edge chains is weakened so that, if a term t is represented at a CFG-node n and at some transitive predecessor p, we can only guarantee that $[t]_n$ is connected by a source-edge chain to $[t]_p$ if there is a cut in the graph between n and the root that includes p (that is, a set of CFG-nodes that pairwise do not share a path, but that every path from the root to n includes one node in the set) and t is represented on each member of the cut - this simply means that, at any join between p and n, t is represented in both joinees.
For equality propagation our aim is to guarantee that if $n \models s = t$ for some CFG-node n, and $s, t \in \mathbf{terms}(g_n)$, then if there is a cut in the CFG between n and the root where each node p in the cut satisfies $[s]_p = [t]_p$ then $[s]_n = [t]_n$. A formal proof remains as future work.

3.4.1 Equality propagation

The main difference with joins is that we do not add the GFAs from all sources of a GT to the GT, but rather only GFAs that have a corresponding source-GFA in *both* joinees.
Consider the example in figure 3.45.
We match the GFAs $f([b]_0)$ and $f([a, b]_1)$ because they share the same function symbol f, and both GFAs are in the sources of the same GT.
Compare the situation to the one in figure 3.46 - here the function labels f,g do not match, hence we do not add a GFA to the GT $[f(a)]_n$ - we do have the matching GFAs $f([a]_0)$ and $f([a, b]_1)$, for which we already have a GFA with the inverse source of the tuple - $[a]_n$. We can see also how the condition for merging with common sources works - while for the sequential case any two GTs that share a source are merged, for the join case we merge GTs only if they share a source in *all* joinees.
Now consider the situation in figure 3.46 - here we have a GT in g_n which represents no terms. We denote such a GT as an infeasible GT, where a GT is feasible if it represents at least one term.
Such infeasible GTs do not arise in the sequential case for consistent EC-graphs,

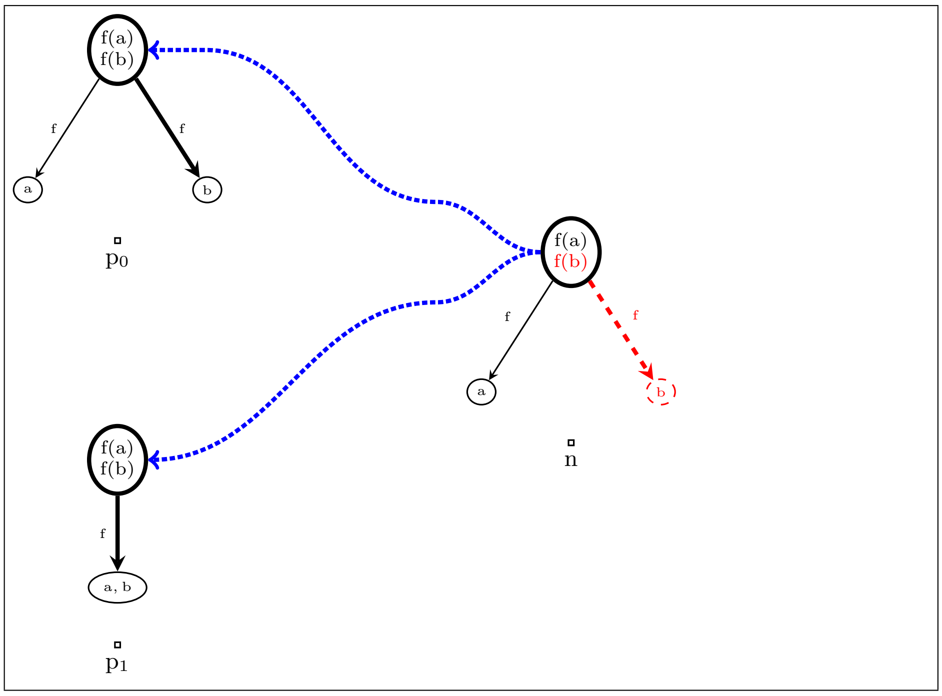


Figure 3.45: Example for GFA completeness for joins
The join is $\{a = b\} \sqcup \{f(a) = f(b)\}$.
The matching function edges are shown in bold.

although they do when scoping or depth restrictions are in place. Our algorithm allows such GTs, but they are only visible to internal methods of the EC-graph. Essentially, they serve as witnesses that the EC-graph is consistent, but they are not visible to users of the EC-graph - that is, the `gfas` and `superTerms` fields visible to users of the EC-graph do not contain these GTs and any GFA whose tuple is infeasible).
For example, if we perform E-matching on the EC-graph of 3.46, matching the (non-ground) term f(x) with the GT $[f(a)]_n$, we get exactly one match (substitution) which is $[x \mapsto [a]_n]$. These infeasible GTs and GFAs that contains them are also not visible to successors of the EC-node - they cannot be the target of source-edges, although they can be the source of source-edges (in fact they must have a source in each joinee), as shown in the example.

We phrase now the version of the local invariant for joins - showing each conjunct.

The conjunct that defines source edges is unchanged. The first conjunct defines when we have to add a GFA to a GT based on its sources - when there are GFAs in both sources that agree on the function symbol:

For a CFG-node n.
$\forall u \in g_n, v_0 \in \mathbf{sources^0}(u), f(\overline{s_0}) \in v_0, v_1 \in \mathbf{sources^1}(u), f(\overline{s_1}) \in v_1 \cdot$
$\quad \exists f(\overline{t}) \in u \cdot \overline{s_0} \in \mathbf{sources^0}(\overline{t}) \wedge \overline{s_1} \in \mathbf{sources^1}(\overline{t})$

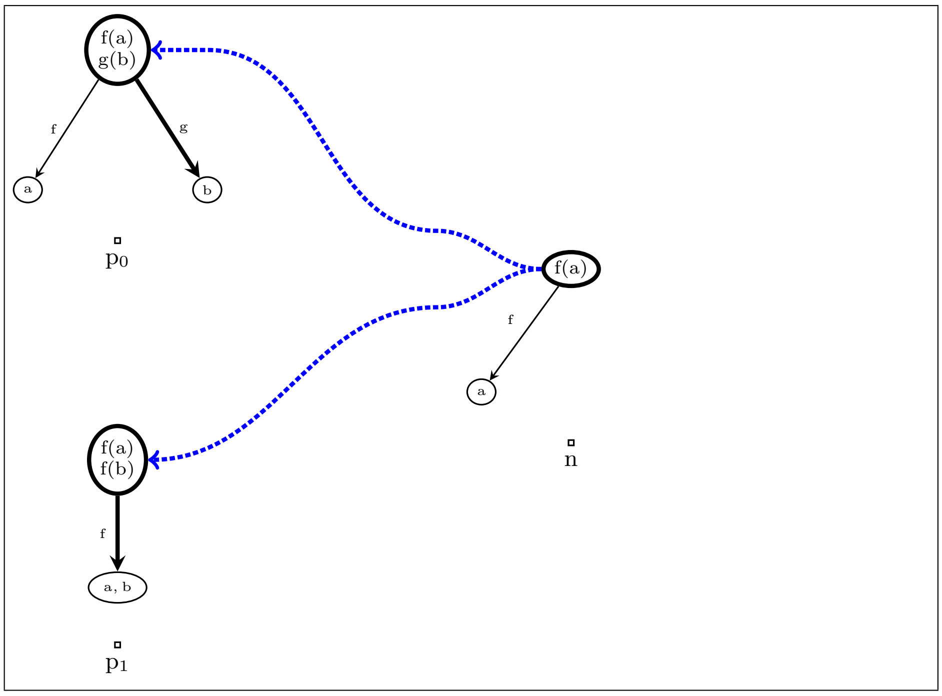


Figure 3.46: Example for GFA completeness for joins
The join is $\{a = b\} \sqcup f(a) = g(b)$.
In this case the highlighted function edges do not match.

The second conjunct defines when we have to merge two GTs at the join - as we have seen, when they share a source in *all* predecessors.
Consider the example in figure 3.48a. No two GTs at the join share both sources, but they all share one source.
We now want to invoke n.`assumeEqual([a],[c])` - if we look at the join, we see that $(a = b \vee b = c) \wedge a = c \models a = b = c$.
In figure 3.48b we can see the state after the first merge step of `assumeEqual`. We can see that $[a, c]_n$ and $[b]_n$ share sources in both joinees, and hence they are merged. In our implementation, we mark these GTs for merging already in the `addSource` method - at the point where a new source is assigned.
The invariant conjunct defining this behaviour is:

For a CFG-node n.
$\forall u, v \in g_n \cdot$
$(\forall p \in \mathbf{ps}(n) \cdot \mathbf{sources^p}(u) \cap \mathbf{sources^p}(v) \neq \emptyset) \Rightarrow u = v$

The third conjunct defines when we have to replace an RGFA with a GT - this condition is crucial for performance and even termination - if we force replacing an RGFA with a GT if *any* of the predecessors represents any term represented by the RGFA, we may have an infinite number of GTs - for example, in the join $\{a = f(a)\} \sqcup \{a = b\}$, where some successor of the join includes the equation $a = f(a)$, we will have to add a GT for each of $\left\{f^i(a) \mid i \in \mathbb{N}\right\}$, as $f(a) = a$ does not hold at the join. Hence our condition is that an RGFA must be replaced

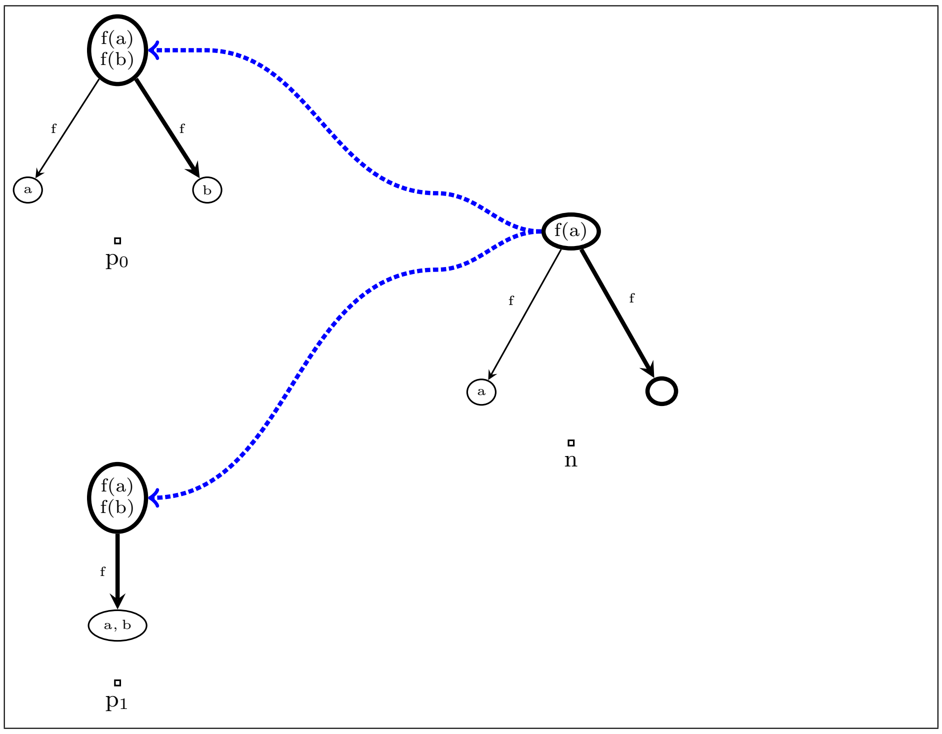
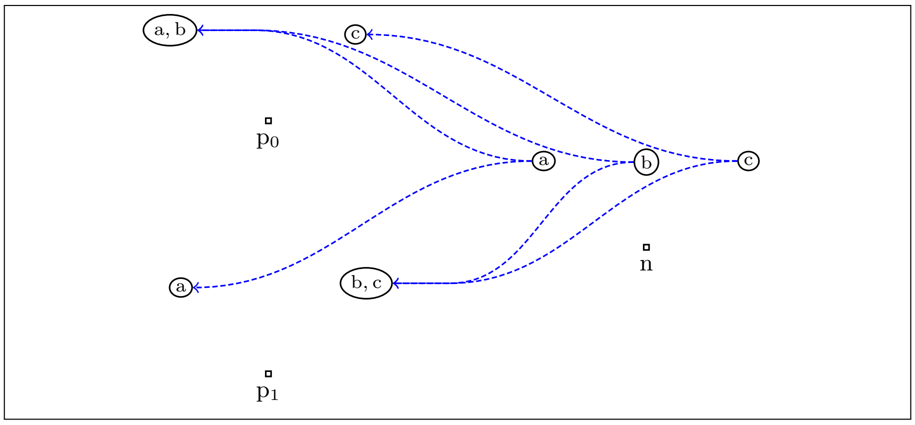
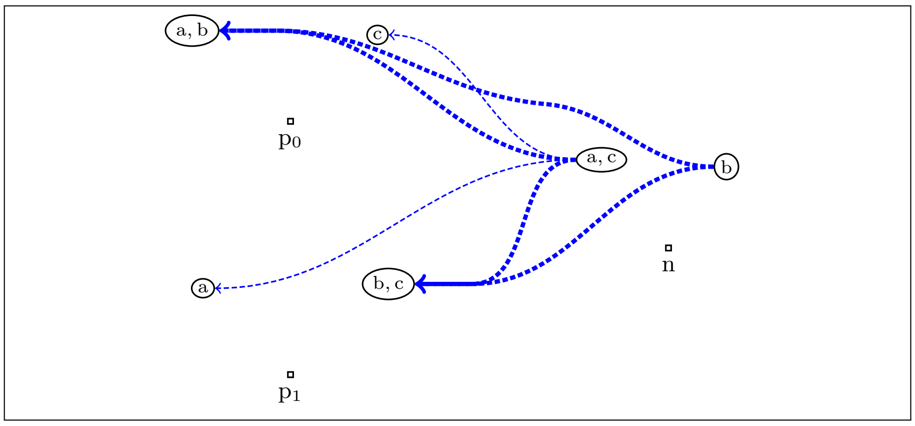


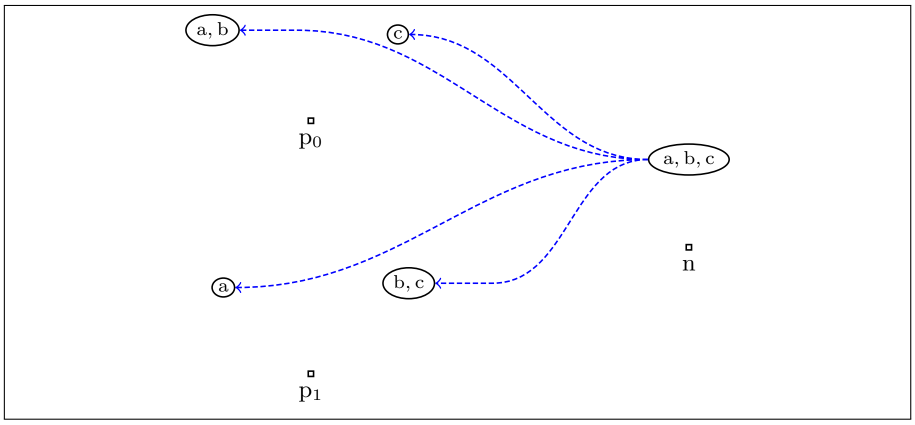
Figure 3.47: Example for GFA completeness for joins
The join is $\{a = b\} \sqcup \{f(a) = f(b)\}$.
The matching function edges are highlighted.



(a) Join shared sources
before n.`assumeEqual([a],[c])`



(b) Join shared sources
after merging [a], [c]
Common sources are highlighted



(c) Join shared sources
Final state

with a GT only if a term is represented in *all* predecessors.
The previous rule for merging GTs ensures that in this case the join will not grow larger than the product of the sizes of the joinees (as we have seen in several examples), as each such added GT must have a source at each joinee, and there are at most $m \times n$ pairs of such sources. This affects the code of `update` and `propagateUpNewSources`, which are responsible for replacing RGFAs with GTs.
Formally, the conjunct is as follows:

$$\boxed{\begin{array}{l}\forall \bar{t} \in g_n, f \cdot \\ \quad (\forall p \in \mathbf{ps}(n) \cdot \exists \bar{s} \in \mathbf{sources}^{\mathbf{p}}(\bar{t}) \cdot f(\bar{s}) \in \mathbf{gfas}_p) \Rightarrow f(\bar{t}) \notin \mathbf{rgfas}_n\end{array}}$$

The rest of the conjuncts remain the same, defining when an RGFA or GFA must exist in a predecessor and which source-edges must exist for each GT.

3.4.2 Strong join

The join we have defined in the previous section is weak in the sense that it does not guarantee propagating equalities for terms not represented on both sides of the join. In this section we show two such examples, and describe how our algorithm handles them.

The first example is show in figure 3.49.
Here, $a = b \vee f(a) = f(b) \models f(a) = f(b)$, but our algorihtm for the weak join does not add the GFA $f([b]_n)$ to $[f(a)]_n$, as there is no corresponding GFA for $f([a]_0)$ - $f(a)$ is not represented in p_1.

Before presenting our solution, we show another example - in figure 3.50 (the green markings are explained below).
In this example, we have invoked `assumeEqual(f([a]_n),g([c]_n)` and now we are in the middle of invoking `assumeEqual(f([b]_n),g([d]_n)` - we have merged the GTs $[f(b)]_n$ and $[g(d)]_n$, but now no invariant forces us to merge $[f(a), g(c)]_n$ with $[f(b), g(d)]_n$.
Here $(a = b \vee c = d) \wedge f(a) = g(c) \wedge f(b) = g(d) \models f(a) = f(b) = g(c) = g(d)$,
but our weak join algorithm fails to propagate these equalities as
$f(a), f(b), g(c), g(d)$ are all not represented in the predecessors.

We sketch here our solution to this problem without getting into many details, as we do not believe they help clarify the solution. The approach is to maintain a representation of the congruence $n \sqcap p_i$ for each $i \in \{0, 1\}$ where $\sqcap$ is a meet (conjunction) - each EC in the meet is called a join-EC, and each join-EC for p_i is associated with a set of GTs from both g_n and g_i. Each GT in g_n is associated with exactly one join-EC and each GT in g_i is associated with *at most* one join-EC. Rather than merge GTs in g_n that share sources in all predecessors, we merge them when they share the join-EC in all predecessors. In addition, we perform congruence and transitive closure on join-ECs.
In figure 3.50, the numbered green lines represent the fact that two GTs share the same join-EC for that predecessor - for example, $[f(a), g(c)]_n$ shares the same join-EC with $[f(b), g(d)]_n$ for p_0 because of congruence closure from $[a]_n, [b]_n$,

that share a join-EC for p_0 as they share a source, and similarly for p_1 because of congruence closure from $[c]_n, [d]_n$.

Looking at the example in figure 3.49, we modify the rule for adding GFAs from predecessors as follows:
Until now, we only allowed adding a GFA if *both* joinees had a GFA in sources with the same function symbol, and the tuple of the result had the corresponding source tuple in each.
Now we allow adding such a GFA at g_n if e.g. p_0 has a GFA f(ptt) and our GT also has GFA (in g_n) f(tt). the resulting tuple will be in the same join-EC for p_0 with ptt (which may have to be added) and the same join-EC for p_1 with tt. For our example, this solution is depicted in 3.51. We can see that the new GT shares now two sources with the GFAs with the same function symbol - b, which allows us to use the weak join rules to complete this example.
The key here is that we only allow adding GTs if they belong to some join-EC for *all* predecessors - otherwise the algorithm might not terminate if one predecessor has a cycle in the EC-graph.

The modifications described above require some more book-keeping state and code to hande join-ECs, but does not increase the worst case space complexity asymptotically and does not break the incrementallity property - that no operation is performed twice. The worst-case space complexity for the weak join is $|\mathbf{gfas}_0| \times |\mathbf{gfas}_1| + |\mathbf{gfas}_n|$ because each added GFA is associated with a unique pair of sources - one from each predecessor. For the strong join, the formula changes to
$|\mathbf{gfas}_0| \times |\mathbf{gfas}_1| + |\mathbf{gfas}_0| \times |\mathbf{gfas}_n| + |\mathbf{gfas}_1| \times |\mathbf{gfas}_n| + |\mathbf{gfas}_n|$,
according to the number of possible GFAs added.
In experiments, we have encountered only a few cases where the strong join discovered equalities not covered by the weak join, but at the same time performance of the strong join was not significantly slower.

Summary

We have shown how our algorithm supports an incremental, on-demand join for the unit ground equality fragment. The algorithm is not changed significantly from the one for sequential nodes, but the guarantee given by the algorithm is weaker. The space requirement of a join is at most quadratic in the size of the input including all auxiliary data-structures (and infeasible GTs) - we use depth limitations to prevent an exponential blowup for joins in sequence, described in chapter 6. In our implementation the scoping and size restrictions are integrated into the entire algorithm including the join.
The complexity of our algorithm for each join is, in fact, proportional to the product of the total sizes (number of GFAs) of all the *relevant* ECs at the join - the algorithm never looks at any predecessor GT or GFA that does not represent a sub-term of a member of an EC of one of the requested terms at the join.

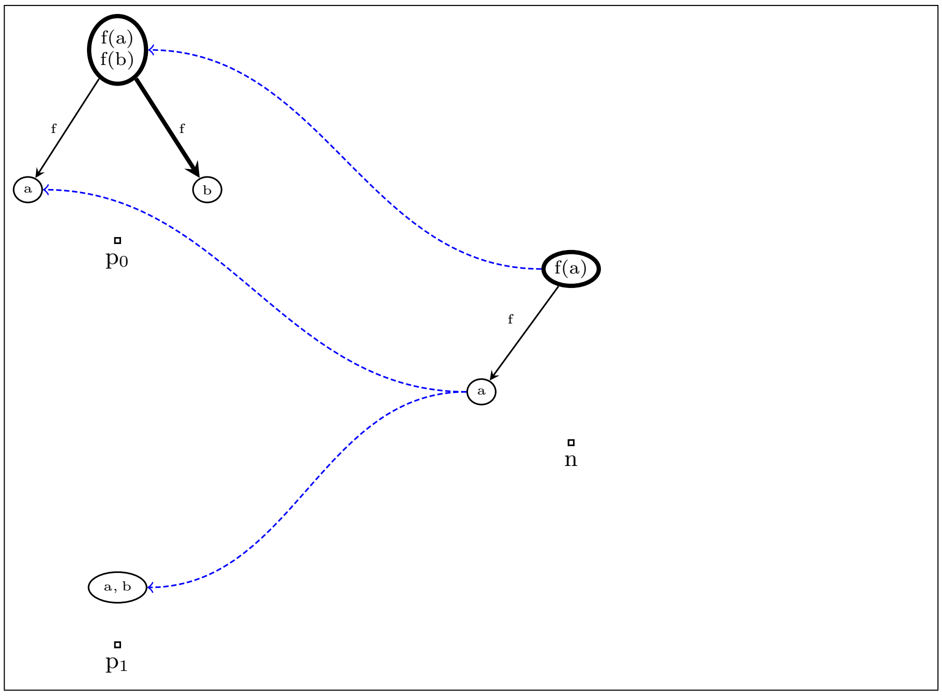


Figure 3.49: Example for the strong join
The join is $\{a = b\} \sqcup f(a) = g(b)$.
In this case the highlighted function edges do not match.

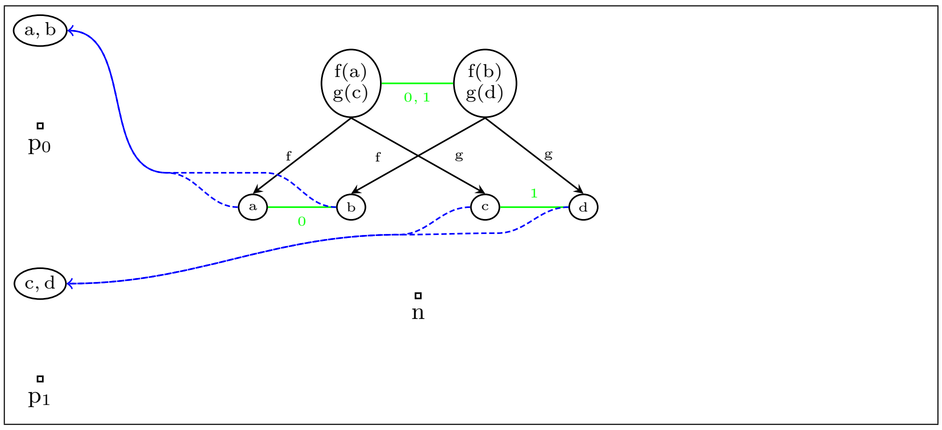


Figure 3.50: Strong join
after the first step of n.`assume(f(b)=g(d))`

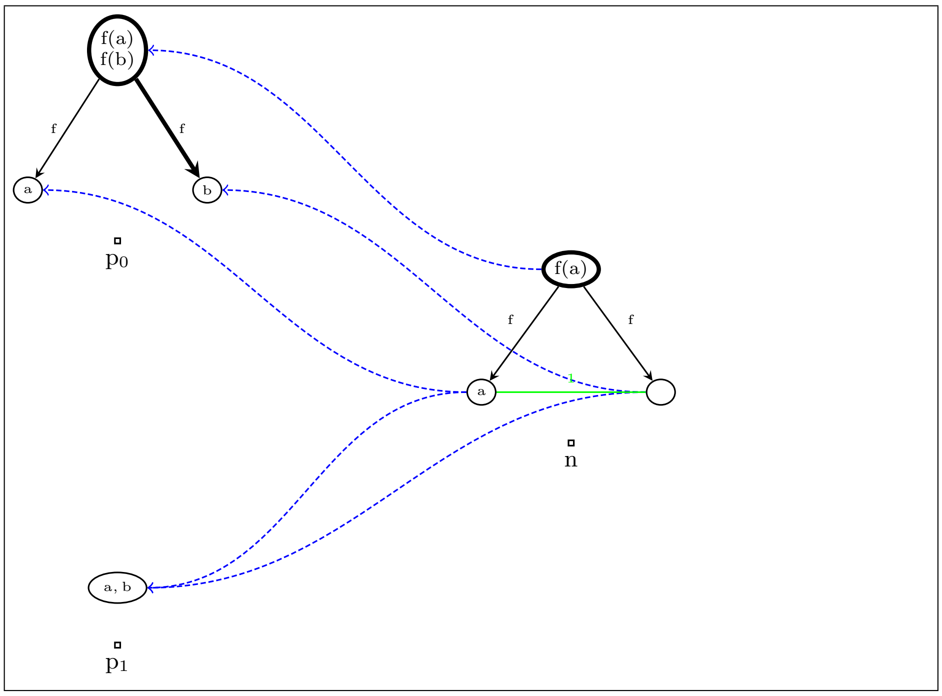


Figure 3.51: Example for the strong join
The join is $\{a = b\} \sqcup f(a) = g(b)$.

3.5 Related Work

3.5.1 Congruence closure

The earlier papers to discuss congruence closure algorithms are [DST80] (efficient algorithms, complexity bounds and alternative implementations), [Sho84] and [NO80] (combination with other theories).

A comprehensive analysis of the join of two EC-graphs is given in [GTN04], including a join algorithm that determines the equality of two terms at a join by adding the terms to the EC-graphs of both joinees. The algorithm is eager in the sense that it represents in the join all terms that occur in both joinees, and works bottom up from constants as opposed to our algorithm that works top-down and hence avoids considering irrelevant joinee terms. Their algorithm works for one join and does not help determine which terms are relevant at which CFG-node. The paper includes several examples, some of which we have repeated here, that show limitations of the congruence closure approach.

In [GN07] the problem of global value numbering is discussed, which concerns the analysis of programs that may contain loops, but include only assignments (i.e. no assume statements). For loop-less programs this is a strict subset of our problem, as essentially it means that each EC-graph node has exactly one non-constant gfa, and also no cycles. The paper gives an example loop-less program that would require an exponentially sized EC-graph at one program point in order to prove, which we have mentioned.

In [NO03] the authors give a congruence closure algorithm, mostly similar to previous ones, except that they eliminate all functions and replace them by the curry function $\cdot$ of arity two, rewriting all the equations accordingly. They also name sub-terms by constants, for which we simply use the graph nodes. We have seen that some operations, especially at join points, are sensitive to function arity, and so this transformation has some attraction, however it has some important disadvantages in our setting: the first disadvantage is that the transformation is arbitrarily non-symmetric, as we can curryfy an n-ary function to either a left or right chain of n *apply* instances, a tree of applications of depth $log(n)$ or any other form. This makes it harder to keep other fragments predictable to users, and harder to enforce size limitations. In our setting it may also make our algorithm propagate much more equality information than is needed, as the original function symbol is not used in filtering most of the terms requested from predecessors (e.g. the GFA f(a,b,c) will be rewritten to $\cdot(f, \cdot(a, \cdot(b, c)))$, and so may propagate $\cdot(b, c)$ even if $\cdot(a, \cdot(b, c))$ will not be propagated, while in our algorithm propagation will only happen if a term with the symbol f occurs).

In [CL05] the authors describe an abstract interpretation framework that includes the domains of unit ground equalities and heaps (or generally other base domains). A join operation is described on EC-graphs which is works bottom up and does not use information from successor nodes to determine which terms to represent at the join. The authors note the potential for sharing sub-graphs that are equal on both sides. The authors also note the incompleteness of the weak join, and in general the fact that information will be missing for terms that do not occur on both sides of the join, and suggest adding these terms to the graph on both sides without giving a strategy for selecting which terms to add. The reason is mostly that they apply abstract interpretation as a forward analysis, which is essentially goal insensitive, while our algorithm propagates information across joins as needed by later assertions. The algorithm in the paper is also not incremental, as in abstract interpretation the intermediate states represent an under approximation of the set of states, rather than an over approximation (until widening is applied).

3.5.2 Information propagation

Information propagation in a restricted fragment is the main theme of abstract interpretation (AI) ([CC77] and many others). The main difference is that abstract interpretation works on a combination of under-approximation and over-approximation of the set of states in order to verify program properties, and can infer invariants, while we only try to verify a given program by calculating over-approximations, and cannot handle invariants with loops or recursion. In spite of these differences, there is strong correlation between the join operation of AI and our join operation (although in AI it is usually not incremental - the value at the join node is recalculated), and some of the completeness issues are similar. Mainly, AI domains are analyzed as a fixed domain or combination of domains in a fixed order, either forwards or backwards (in program flow sense).

As mentioned above ([CL05]) has developed an AI domain for congruence closure. Several domains have been suggested for the problem of global value numbering - where only assignments are allowed (essentially, for acyclic programs). This means in our setting that the only assume statements allowed are an equality $\mathrm{v_i} = \mathrm{t_{i-1}}$, where $\mathrm{v_i}$ is the ith DSA version of the variable v and $\mathrm{t_{i-1}}$ is a term only including constants and variables of the previous DSA version - essentially this means that EC-graphs cannot have loops. This is an early problem coming from compiler optimizations that has an exponential complete solution in [Kil73] and several other results are compared in [GTN04]. [Vág03] gives a decision procedure to detect when a join of two sets of equations is finitely generated and gives an algorithm to calculate the join, which stands in accuracy between our strong join and weak join.

For programs with loops, [MS04] shows that allowing positive equality assume statements (positive guards) makes the fragment undecidable in the presence of loops, using an encoding of PCP. [GT07a] discuss complexity issues and the relation to unification.

The **IC3/PDR** ([Bra12]) is used to infer propositional invariants of programs, essentially by looking at bounded unrollings of the loop and strengthening the invariant for the nth iteration by searching for a pre-image in the transition relation in the n-1 iteration, and if the found pre-image is infeasible, the algorithm tries to strengthen it and propagate it to later iterations. IC3 has been extended to EPR ([BGKL13], linear arithmetic (e.g. [BG15] - combined with interpolation and polyhedra abstract interpretation) and others. The technique and its further developments offer several ways to propagate and generalize the clause in order to reduce the number of iterations. IC3 has proved very effective and has several extensions and generalization strategies, including a combination with interpolation in [VG14].
Compared to our work, IC3 can infer invariants and each step of IC3 requires a SAT call (or SMT call in some works) in order to find a counter example, and another call in order to calculate the pre-image, the first call only considers one program point and the second two program points, so that the SAT/SMT

problem is much smaller than whole program VC. Compared to our information propagation technique, IC3 requests to predecessors are a conjunction of ground literals (a set of models) while we request either a single literal or one ground term. In terms of complexity, IC3 ensures progress for each step, but the worst case cost of each SAT/SMT call is exponential, and even within one unrolling, the number of counter-examples that may be sent as requests can be exponential as well (although it might be that some clause generalization method can prevent this - we are not aware of such a result), and the choice of counter-examples is not easily predictable. As in other SAT/SMT based techniques, it is not immediately clear how to ensure a polynomial run-time for a sub-fragment and how to ensure progress in the presence of quantifiers.
A direct comparison when trying to prove an assertion $\mathrm{s} = \mathrm{t}$ is that we try to propagate the whole equivalence classes of s, t and then compare, while IC3 directly asks whether the pre-image of a specific state where $\mathrm{s} = \mathrm{t}$ holds. For propositional queries we request one literal and the reply is a set of clauses, while IC3 requests a conjunction of literals and the reply is one clause. Hence, IC3 asks more specific questions and propagates much less unneeded information (which is necessary in order for invariants to converge), while we ask less specific questions but can bound the number of answers needed for saturation. It would be interesting to see how we can, in some cases, utilize more directed queries as in IC3, while maintaining polynomial bounds. In order to handle quantifiers, an efficient representation for infinite counter-examples is needed, we suspect that the representation used in [BPT12] (which extends DPLL to FOL - hence has a representation for partial models with quantifiers) could be a potential, although as above, the problem is that no prover run is guaranteed to even terminate, and no progress is guaranteed for a prover time-out.

Another existing technique is based on modular sat solving ([BVB^{+}13]) - here (in our terminology) each CFG-node gets its own SAT solver, each assertion leaf node searches for a model, which is then communicated to its predecessor (in the paper it is described for a sequence, rather than tree or DAG, CFG), each SAT solver is incremental and can receive new assignments from successors and new lemmas (learned clauses) from predecessors. This approach is appealing as it is local and potentially each solver could face a much smaller problem than that of a whole VCG solver. It would be interesting to see how this approach fares when extended to SMT and DAGs (in the paper it is described as a method to implement IC3, so in fact loops are supported, but acyclic sub-CFG are converted to one SAT instance - there is no provision for joins). However, it faces some of the problems mentioned for IC3 above - each SAT instance could potentially run for exponential time, and a model has to be completely refuted (with an explaining lemma/interpolant) in order to continue verification.

Chapter 4

Ground first order logic with equality

In this chapter we present our algorithm for verifying program VCs that include only ground clauses. For many programs, a significant part of the proof is done at the ground level, hence a verifier must be able to discharge ground proofs efficiently. There ground fragment is decidable. An important property of proofs of program VCs, as opposed to general logic proofs, is that the same properties (assertions) have to be proven for many traces (propositional assignments) that differ only slightly. We extend the algorithm proposed for unit clauses to take advantage of this propositional structure during the entire proof search, and not only during VC generation.

We start by presenting the ground fragment and our basic verification algorithm for the fragment. We then compare our algorithm to VC generation and SMT and superposition based verification and show the reasoning that lead us to this algorithm. Next we sketch the completeness argument for the algorithm. The next section discusses joining non-unit clauses and the role of redundancy elimination. We then consider the issue of selecting a term ordering for the superposition calculus based on the structure of the VC. In section A.2 of the appendix we show how we combine our EC-graph algorithm from chapter 3 and the algorithms in this chapter.

4.1 The fragment of GFOLE

In this section our language is as described in the preliminaries:
The signature is $\Sigma = \mathbf{F}_\Sigma \cup \mathbf{P}_\Sigma \cup \mathbf{X}_\Sigma$,
where $\mathbf{F}_\Sigma$ is the set of function symbols (including constants), $\mathbf{P}_\Sigma$ is the set of predicate symbols and $\mathbf{X}_\Sigma$ is the set of free variables - which is empty in this section. Initially we assume that $\mathbf{P}_\Sigma$ is also empty so that the only predicate is the (interpreted) equality predicate. We also use nullary predicates for branch conditions and later discuss the benefits of using general predicate symbols. We base our handling of non-unit ground predicates on the ground superposition calculus shown in figure 4.1. The reason we choose superposition as the basis is twofold - ground superposition can be extended naturally to full superposition for a complete calculus for FOLE, as opposed to most SMT based approaches,

$$\mathrm{res}_{=} \quad \frac{C \lor \underline{s \neq s}}{C}$$

$$\mathrm{sup}_{=} \quad \frac{C \lor \underline{l} = r \quad \underline{s} = t \lor D}{C \lor s\,[r]_p = t \lor D} \qquad \begin{array}{l} l = s|_p \\ l \succ r, l = r \succ C \\ s \succ t, s = t \succ D \\ s = t \succ l = r \end{array}$$

$$\mathrm{sup}_{\neq} \quad \frac{C \lor \underline{l} = r \quad \underline{s} \neq t \lor D}{C \lor s\,[r]_p \neq t \lor D} \qquad \begin{array}{l} l = s|_p \\ l \succ r, l = r \succ C \\ s \succ t, s \neq t \succ D \end{array}$$

$$\mathrm{fact}_{=} \quad \frac{C \lor l = t \lor \underline{l} = r}{C \lor t \neq r \lor l = t} \qquad \begin{array}{l} l \succ r, r \succ t \\ l = r \succ C \end{array}$$

Figure 4.1: Ground superposition calculus - $\mathbf{SP_g}$
Clauses are sets of literals (rather than sequences or multi-sets).
The ordering $\succ$ is any reduction ordering.
The rule $\mathrm{res}_{=}$ eliminates a false literal.
The rules $\mathrm{sup}_{=}, \mathrm{sup}_{\neq}$ perform conditional rewriting of the maximal term in the second premise.
The rule $\mathrm{fact}_{=}$ is an instance of factoring required because superposition only targets the maximal term.

and superposition (and any saturation based method) admits incremental fragments of increasing strength (e.g. by bounding derivation depth), while SMT based method require a conflict in order to progress.

This calculus is based on conditional rewriting - the superposition rules rewrite l to r under the condition C - essentially a case-split - if C does not hold, then l=r holds and hence we can replace l by r.
This calculus is complete for refutation - the empty clause is derivable from a set of clauses iff that set is inconsistent. The superposition calculus admits a strong form of redundancy elimination. We give some examples in section 2. Our technique is not tightly bound to superposition, and can be instantiated with other calculi, under some restrictions.

Program VCs and proofs

Given a program CFG where all clauses at all CFG-nodes are ground clauses, we can use the above calculus in several ways in order to verify the program. We assume here we have calculated, for each CFG-node, a set of clauses that represents the statements (transition relation in DSA form) at that CFG-node, and that assertions have been negated before being converted to a set of clauses, as described in section 2.

As before, we denote by $\mathbf{C}_\mathrm{n}$ the set of clauses at the node n obtained from translating the statements at that node. Several encodings have been proposed for generating VCs for programs (e.g. [Lei05]). In general, if one VC is produced for the entire program, information about the CFG structure has to be encoded into the VC to prevent exponential explosion - weakest preconditions duplicate a part of the formula at each branch. We do not consider VCs with loops here, but assume loops have been removed as discussed in section 2. In order to present some of the difficulties faced by a saturation based theorem prover that are particular for program VCs, we consider here two opposing approaches to generating VCs - both use the DSA form of a program:

- Generate a flat VC of the program (a logical formula), where a model is a trace of the program in which an assertion fails. We use a path-literal - a nullary predicate P_n - for each CFG node, which, when true in a model, means that the trace represented by the model includes the node n. The flat VC is comprised of three parts, each a set of CNF clauses (this encoding is somewhat similar to the one used in [GCS11]):
  - The program location each clause holds:
    $\bigwedge\limits_{\substack{\mathrm{n}\in\mathrm{cfg}\\ \mathsf{c}\in\mathbf{C}_\mathrm{n}}} (\mathrm{P}_\mathrm{n} \rightarrow \mathsf{c})$
  - The structure of the CFG: ($\mathbf{ps}(\mathrm{n})$ are the direct predecessors of n) - if a trace reaches a node n, it must have reached one of n's predecessors
    $\bigwedge\limits_{\mathrm{n}\in\mathrm{cfg}\setminus\{\mathrm{root}\}} (\mathrm{P}_\mathrm{n} \rightarrow \bigvee\limits_{\mathrm{p}\in\mathbf{ps}(\mathrm{n})} \mathrm{P}_\mathrm{p})$
  - The verification condition - each assertion must hold (negated - a model is a trace for one failed assertion):
    $\bigvee\limits_{\mathrm{n}\in\mathrm{assertions}} \mathrm{P}_\mathrm{n}$

  The program flow is encoded backwards - if a trace includes a CFG-node n it must include one of its predecessors. A failing trace is one in which one of the assertion path conditions holds. Each clause is attached the relevant path-literal, and we add an implication from each node to a disjunction of its predecessors.
  There are two common alternatives on the above encoding, one alternative is similar but encodes the program flow forward - that is, the second part is replaced with each path literal implying a disjunction of its successors, and the third part is replaced by simply the root path literal. Another common encoding (used e.g. by Boogie, and geared more towards SMT solvers) is, instead of path literals, to use a literal, per CFG-node, that encodes that the program is correct from that point on (this encoding is closer to the original weakest preconditions). For our purposes there is little difference between the encodings, as in all of them essentially clauses describe the transition relation of each statement or basic block is guarded by some propositional literals that encode program flow. We note only that for lazy CNF-conversion SMT solvers the shape of the actual formula produced is very important for performance, while solvers that perform eager CNF conversion are less sensitive (superposition is sensitive to the ordering chosen, though).

```
n0 :
if (c1)
   n1 :
   assume A
else
   n2 :
   assume B
   a2 :
   assert A1
n3 :
if (c2)
   n4 :
   assume C
else
   n5 :
   assume D
   a5 :
   assert A2
a6 :
assert A3
```

Figure 4.2: Example program for simple VC comparison

- We can generate a VC per-path from the root to a leaf, and the program VC is then the disjunction of these VCs (that is, a model is the assignment to each program variable that brings to at least one failing trace)

  $$\bigvee_{\substack{\mathrm{n}\in\mathrm{assertions} \\ \mathrm{P}\in\mathbf{paths}(\mathrm{n})}} \bigwedge \mathbf{C}_{\mathrm{P}}$$

  The encoding is a disjunction of conjunctions of clauses, where each conjunction is consistent iff the assertion at the end of the path does not hold on that path. Hence, if we derive the empty clause on each disjunct, we have shown that all assertions hold in the program.
  The advantage is that we do not need to encode path conditions at all, and we can evaluate every disjunct independently (a disjunction is false iff all disjuncts are false). The disadvantage is that we can have an exponential number of paths (and hence disjuncts).
  Obviously, this is not a practical encoding in most cases, and implies a lot of duplication, we include it here to emphasize the price of the encoding of the CFG structure, but also because, in a sense, an SMT solver always sees (part of a) single trace, hence this encoding shows some of the reason that SMT solvers have more success in program verification applications.

We show now an example comparing the two approaches and discuss the differences. In the rest of this chapter we describe an algorithm that aims to answer the challenges posed by this problem. Consider the program in figure 4.2.

The first encoding produces (a formula in CNF):
$\mathrm{n}_1 \rightarrow \mathrm{A}$

$n_2 \rightarrow B$
$a_2 \rightarrow n_2$
$a_2 \rightarrow \neg A_1$
$n_3 \rightarrow n_1 \vee n_2$
$n_4 \rightarrow n_3$
$n_4 \rightarrow C$
$n_5 \rightarrow n_3$
$n_5 \rightarrow D$
$a_5 \rightarrow n_5$
$a_5 \rightarrow \neg A_2$
$a_6 \rightarrow n_4 \vee n_5$
$a_6 \rightarrow \neg A_3$
$a_6 \vee a_3 \vee a_2$

And the third encoding produces (a formula in DNF):
$(B \wedge \neg A_1)\vee$
$(A \wedge D \wedge \neg A_2)\vee$
$(B \wedge D \wedge \neg A_2)\vee$
$(A \wedge C \wedge \neg A_3)\vee$
$(A \wedge D \wedge \neg A_3)\vee$
$(B \wedge C \wedge \neg A_3)\vee$
$(B \wedge D \wedge \neg A_3)$

In each case, if we convert the formulae to CNF form and saturate the set of CNF clauses that we get w.r.t. the calculus $\mathbf{SP_g}$, we derive the empty clause iff the program is correct - every assertion in the program is satisfied by all traces of the program.

In the second encoding (per-path VC), we could instead convert each disjunct separately to CNF, and saturate separately each such set of CNF clauses - if we have derived the empty clause from the set of CNF clauses converted from one of the disjuncts, it means that on the path that disjunct represents, the assertion at the end of the path holds. If we had derived the empty clause in each CNF set converted from a disjunct, then the program is verified - all assertions hold on all paths leading to them. In this case we have many more VCs than in the first two cases (the *flat VCs*), but each VC is much simpler than the VCs for the first two cases.

The important point is that we do not need any derivation between clauses in parallel branches (e.g. between A and B) in order to ensure completeness, as evidenced by the per-path VC. However, as we have seen, *joining* clauses from parallel branches can be beneficial, sometimes reducing the minimal proof size exponentially.

Our verification algorithm can be seen as compromise between the two approaches presented above - on the one hand we integrate knowledge of the CFG in order to block derivations that are not needed for the proof (e.g. between clauses in parallel branches), as in the per-path VC. On the other hand we perform joins and maintain one set of clauses per CFG-node rather than per-path, so as to avoid an exponential explosion in the number of sets of clauses that need to be saturated.

4.1.1 Basic clause propagation

We repeat here in figure 4.3 the simple verification algorithm from 3.25 for the ground non-unit fragment that forms the basis for later improvements, with a slight modification (deatiled below). The algorithm is presented in a modular way and later sections refine and improve certain parts of the algorithm.

The algorithm maintains two sets of clauses per CFG-node n - `done` and `todo` (we write $\texttt{done}_\mathrm{n}$ to distinguish these sets for different CFG nodes where appropriate). The set `done` is inter-saturated w.r.t. $\mathbf{SP_g}$, and a clause can be moved from `todo` to `done` once all conclusions of inferences between it and other clauses in $\mathrm{done_n}$ are added to $\mathrm{todo_n}$. We say that a node n is saturated iff $\mathrm{todo_n} = \emptyset$.
The algorithm traverses the CFG-nodes in topological order, and saturates the set of clauses at each node. Saturation includes, in addition to saturation w.r.t. SP, also saturation w.r.t. the propagation of all relevant clauses from predecessors. We propagate a clause D from a node p to a transitive successor n (that is, we add $\mathrm{P} \vee \mathrm{D}$ to $\texttt{done}_\mathrm{m}$, where P is a path prefix described later) iff there is a valid inference between D and a clause $\mathrm{C} \in \texttt{todo}_\mathrm{n}$. We do not add the clause to $\texttt{todo}_\mathrm{n}$ as any inference between two propagated clauses is not needed at the node, as we show below).
We say that an inference happens at a node when the inference is between two clauses, one in `todo` and the other in `done`. Our algorithm ensures that any valid inference along a path P in the CFG happens at the earliest node n where both clauses occur (or are derived) in the prefix of P up to n. When an inference happens at a node n, we add the result of the inference to `todo`, so it can participate in further inferences at the node and also can be propagated to later nodes as per requests (once it is added to `done`). A node is moved from `todo` to `done` once all possible inferences and propagation have been performed on it. We also perform several simplifications on clauses before they are allowed to participate in inferences or propagation, described below.

The difference in the algorithm is that we import relevant clauses for the entire queue of clauses `todo` waiting for processing, rather than for each clause separately - this allows us to reduce the number of CFG traversals. We process the entire `todo` queue before enqueing any new clauses, so that each iteration performs inferences of a certain derivation depth.

The `importRelevantClauses` method, shown in figure 4.4, receives a set of clauses and propagates all relevant clauses from the transitive predecessors of the node to the current node. The definition for relevance is an extension of the definition we have seen for unit superposition in section 3.3.2 - essentially a clause is relevant for another clause if there is an inference rule instance in $\mathbf{SP_g}$ where both are premises. For example, for the clause $\mathrm{C} \vee \underline{\mathrm{f(b)}} = \mathrm{a}$ ($\mathrm{f(b)}$ maximal), the clauses $\mathrm{D} \vee \underline{\mathrm{b}} = \mathrm{a}, \underline{\mathrm{g(f(b))}} \neq \mathrm{c} \vee \mathrm{D}$ are relevant, but $\mathrm{D} \vee \mathrm{f(b)} = \mathrm{b} \vee \underline{\mathrm{g(c)}} = \mathrm{c}$ is not. We describe the relevance function for $\mathbf{SP_g}$ in detail below.

The `importRelevantClauses` uses the method `travevrseBF` from figure 3.27 to traverse the CFG backwards from n and propagate a request, where a request is propagated if it is not covered by the cache at the node (detailed below). The algorithm then traverses the CFG in reverse order and propagates the relevant clauses up in topological order. The method first constructs a propagation request (detailed in figure 4.5), which is essentially two sets of terms, and traverses the CFG first in reverse-topological order (using the method

```
CFG.verify() : Set of unverified assertions
   foreach node n in topological order
      n.verify()
      if (!n.isInfeasible and n.isLeaf)
         result.add(n)

Node.verify()
   todo.enqueue( $\mathbf{C_n}$ )
   while !todo.isEmpty
      importRelevantClauses(todo)
      var ncs := new Queue
      foreach (c ∈ todo)
         done.add(c)
         foreach (d ∈ inferences(c,done))
            if (d ∉ done)
               ncs.add(d)
      todo := ncs
```

Figure 4.3: Verification algorithm with lazy propagation.
inferences(c,done) returns all possible inference rule instances from $\mathbf{SP_g}$ between the clause c and clauses in done.
The difference from the version in 3.25 is that we import clauses for the entire todo queue in one call, rather than repeated calls per clause.

travevrseBF from figure 3.27), where each node traversed transforms the request for its direct predecessors (e.g. filtering out cached requests) - this is described in figure 4.4).

A CFG-node with more than one successor receives a request that is a join (by Request.add (described in figure 4.5) of the requests of all its successors) - without joining we would be traversing the CFG as a tree rather than a DAG and so traversal could take exponential time. Once the request reaches a dead-end on all paths (e.g. reaching the root or a node where the request is already cached), we traverse the CFG in topological order from each such dead-end node, propagating clauses forward. The propagated clauses are *relativized* using the branch literals - so that when a clause is propagated to a node n from a transitive predecessor p, it is propagated as a disjunction of the original clause with the relative path condition of n and p - this allows us to minimize the overhead of path conditions in the proof to the necessary minimum (discussed later). Each CFG-node maintains a cache of requests, and does not propagate a request if it is included in the cache - all clauses that were imported to answer a request are added to the done set.

In order to present the algorithm modularly, we define the methods importBW and importFW which define the transformation of the request and the clauses performed at each node. These transformations include caching, clause simplification and more, and are detailed in the rest of this chapter. The basic version for these two visitor methods is presented in figure 4.4 - these are unchanged from the version we have seen in figure 3.26.

Superposition requests:
We repeat here briefly the structure of a superposition request from section

```
Node.importRelevantClauses(cs : Set[Clause])
   requestMap := new Map[Node,Request]

   requestMap[this] := makeRequest(cs)

   traverseBF(this, importBW, importFW)

importBW(n : CFGNode) : Set[CFGNode]
   var r := requestMap[n]
   r.subtract(n.requestCache)
   if (!r.isEmpty && n!=root)
      foreach (p ∈ ps(n))
         requestMap[p].add(r)
      return predecessors
   else
      return ∅

importFW(n : CFGNode)
   r := requestMap[n]
   n.requestCache.add(r)
   foreach (p ∈ ps(n))
      foreach (pc ∈ p.done)
         c := relativize(p,n,pc)
         if (isRelevant(r,c))
            n.done.add(c)
```

Figure 4.4: Basic recursive import
The implementation uses the method `traverseBF` to traverse the CFG first backwards and then forwards starting at the current node, calling the visitors `importBW`,`importFW` respectively.
The backward visitor propagates the request backwards except for cached requests and the forward visitor propagates clauses forward.

```
Request(Cs : Set[Clause])
   ts$_{lhs}$:= ∪ {lhs(C) | C∈Cs}
   ts$_{rhs}$:= ∪ {rhs(C) | C∈Cs}

lhs(C:Clause) : Set[Term]
   **return**
      **if** (maxLiteral(C) is a positive equality)
         **then** {maxTerm(C)}
         **else** $\emptyset$

rhs(C:Clause) : Set[Term]
   **return** subTermClosure(maxTerm(C))

Request.add(other : Request)
   ts$_{lhs}$ ∪:= other.ts$_{lhs}$
   ts$_{rhs}$ ∪:= other.ts$_{rhs}$

Request.subtract(other : Request)
   ts$_{lhs}$ \:= other.ts$_{lhs}$
   ts$_{rhs}$ \:= other.ts$_{rhs}$

isRelevant(r : Request, C : Clause)
   **return**
      lhs(C)∩ts$_{rhs}$ $\neq \emptyset$
         ∨
      rhs(C)∩ts$_{lhs}$ $\neq \emptyset$
```

Figure 4.5: Basic recursive import - Request
A request consists of left- and right-hand-side maximal terms.
Adding and removing requests are the standard set operations.

3.3.2 - it is unchanged for from the ground unit fragment - a basic request for superposition is simply two sets of terms. The first set, $\mathtt{ts}_{\mathrm{lhs}}$ is the set of all strictly maximal terms in clauses that appear in positive equalities and so can instantiate l for clauses that serve as the left premise of a superposition inference ($\text{sup}_=$ or $\text{sup}_{\neq}$). Similarly, $\mathtt{ts}_{\mathrm{rhs}}$ is the set of terms that are strictly maximal terms in clauses at the node, and their sub-terms, so exactly the terms that can instantiate l for both superposition inference rules for the right premise. Essentially, for clauses C,D, there is an instance of a superposition inference between C and D with C as the left premise and D as the right premise iff $\text{lhs}(C) \cap \text{rhs}(D) \neq \emptyset$ (for positive superposition inferences it might be the case that C is the right premise - the condition $s = t \succ l = r$ is not encoded in our request). This request mechanism can be generalized to other calculi that depend on maximal terms and literals, as long as we can encode a join of requests - in our simple case a join for requests is simply a union of the sets of left and right hand sides.

Comparison with flat VCs

In this sub-section we show the rationale in the above algorithm by comparing its operation with that of a verifier working on the flat VC we have defined above

```
$n_0$ :
assume g(f(e))=T
if (c1)
   $n_1$ :
   assume e = b
else
   $n_2$ :
   if (c2)
      $n_3$ :
      assume f(e)=f(c)
   else
      $n_4$ :
      assume g(f(e))=g(f(d))
   $n_5$ :
$n_6$ :
assume b=u
assume c=u
assume d=u
assert g(f(u))=T
```

Figure 4.6: Example program for VC verification comparison
The ordering is $e \succ d \succ c \succ b \succ a \succ u \succ T$
The node n_5 was added to ensure all joins are binary

(the first encoding). We show how our algorithm avoids some of the inferences performed by a superposition prover on a flat VC, by exploiting the CFG structure to avoid inferences on clauses in opposing branch sides. Clauses derived from premises in opposing branch sides cannot participate in a refutation, and hence completeness is not compromised by avoiding such inferences. This is one of the main motivations for our algorithm.

Our algorithm saturates the set of clauses at each node, propagating clauses forward as necessary between nodes, but ensuring that each instance of an inference occurs only once and at a specific node - the later (in topological order) of the two nodes where the clauses appear. This means that two clauses that appear in nodes that do not share any path (e.g. on different sides of a branch) will never participate in the same instance of an inference rule.

Roughly, the semantics of a clause C appearing at a node n is that $P_n \rightarrow C$ holds where P_n is the path-condition of the node n (regardless of how path conditions are encoded) - a model is a trace of the program and P_n holds for each model where n is reached. Taking the model view we see that for each pair of clauses C at node n and D at node m where m,n do not share any path, for any model we can set either P_n to false or P_m to false (or both) - hence the result of any derivation between $\neg P_n \vee C$ and $\neg P_m \vee D$ where the maximal terms are at C,D respectively, will have a model iff there is any model to the set of clauses solely because $\neg P_n \vee \neg P_m$ is satisfiable.

If we compare the above algorithm to the flat VC case, we have avoided some derivations, namely derivations between clauses that do not share a path, and our clauses carry less "baggage" of path conditions, as we only encode the relative path condition for each clause, rather than the absolute one.

Consider the code in figure 4.6 with the flat-VC in figure 4.7. In figure

1. $\neg n_0 \lor \underline{g(f(e))} = T$

2. $\underline{\neg n_1} \lor n_0$
3. $\neg n_1 \lor \underline{e} = b$

4. $\underline{\neg n_2} \lor n_0$

5. $\underline{\neg n_3} \lor n_2$
6. $\neg n_3 \lor \underline{f(e)} = f(c)$

7. $\underline{\neg n_4} \lor n_2$
8. $\neg n_4 \lor \underline{g(f(e))} = g(f(d))$

9. $\underline{\neg n_5} \lor n_3 \lor n_4$

10. $\underline{\neg n_6} \lor n_1 \lor n_5$
11. $\neg n_6 \lor \underline{b} = u$
12. $\neg n_6 \lor \underline{c} = u$
13. $\neg n_6 \lor \underline{d} = u$
14. $\neg n_6 \lor \underline{g(f(u))} \neq T$
15. n_6

Figure 4.7: Simple flat VC for figure 4.6
maximal terms are underlined
We use the literal ordering:
$n_5 \succ n_4 \succ n_3 \succ n_2 \succ n_1 \succ n_0 \succ c_0 \succ c_1$
Where all path literals are smaller than all equality literals

4.7, we can perform superposition inferences between any pair of the set of clause $\{1, 3, 6, 8\}$ - 6 possible inferences in total. Only the inferences between 1 and $\{3, 6, 8\}$ (3 in total) are relevant to the proof, as all other inferences occur between clauses on parallel paths in the CFG and any clause that is the result of such an inference is trivially true in any model - for example, if we superpose 3 and 6:

$$\frac{\neg \mathrm{n}_1 \vee \underline{\mathrm{e}} = \mathrm{b} \quad \neg \mathrm{n}_3 \vee \underline{\mathrm{f(e)}} = \mathrm{f(c)}}{\neg \mathrm{n}_1 \vee \neg \mathrm{n}_3 \vee \mathrm{f(b)} = \mathrm{f(c)}}$$

The conclusion is redundant.

Example algorithm run:
We represents requests as ordered pairs, $(\texttt{ts}_{\text{lhs}},\texttt{ts}_{\text{lhs}})$ - the left- and right-hand-side of requests. The run goes as follows:

- n_0 Nothing to be done.
  $\texttt{done}_0 = \{\mathrm{g(f(e))} = \mathrm{T}\}$
- n_1 sends the request $(\{\mathrm{e}\}, \{\mathrm{e}\})$ and receives the response $\left\{\underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  which allows it to derive $\underline{\mathrm{g(f(b))}} = \mathrm{T}$
  The request $(\{\mathrm{g(f(b))}\},\{\mathrm{b}, \mathrm{f(b)}, \mathrm{g(f(b))}\})$ returns nothing.
  $\texttt{done}_1 = \left\{\underline{\mathrm{e}} = \mathrm{b}, \underline{\mathrm{g(f(b))}} = \mathrm{T}, \underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  $\texttt{cache}_1 = (\{\mathrm{e}, \mathrm{g(f(b))}\}, \{\mathrm{e}, \mathrm{b}, \mathrm{f(b)}, \mathrm{g(f(b))}\})$
- n_2 has nothing to do.
  $\texttt{done}_2 = \emptyset$
  $\texttt{cache}_2 = \emptyset$
- n_3 sends the request $(\{\mathrm{f(e)}\}, \{\mathrm{e}, \mathrm{f(e)}\})$ and receives the response $\left\{\underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  which allows it to derive $\underline{\mathrm{g(f(c))}} = \mathrm{T}$
  The request $(\{\mathrm{g(f(c))}\},\{\mathrm{c}, \mathrm{f(c)}, \mathrm{g(f(c))}\})$ returns nothing.
  $\texttt{done}_3 = \left\{\underline{\mathrm{f(e)}} = \mathrm{f(c)}, \underline{\mathrm{g(f(e))}} = \mathrm{T}, \underline{\mathrm{g(f(c))}} = \mathrm{T}\right\}$
  $\texttt{cache}_3 = (\{\mathrm{f(e)}, \mathrm{g(f(c))}\}, \{\mathrm{e}, \mathrm{f(e)}, \mathrm{c}, \mathrm{f(c)}, \mathrm{g(f(c))}\})$
  $\texttt{done}_2 = \left\{\underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  $\texttt{cache}_2 = (\{\mathrm{f(e)}, \mathrm{g(f(c))}\}, \{\mathrm{e}, \mathrm{f(e)}, \mathrm{c}, \mathrm{f(c)}, \mathrm{g(f(c))}\})$
- n_4 sends the request $(\{\mathrm{g(f(e))}\}, \{\mathrm{e}, \mathrm{f(e)}, \mathrm{g(f(e))}\})$,
  The request is propagated as $\{\mathrm{g(f(e))}\}, \{\mathrm{g(f(e))}\}$ from n_2 as the rest is cached.
  The response is as before (cached) $\left\{\underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  which allows us to derive $\underline{\mathrm{g(f(d))}} = \mathrm{T}$
  The request $(\{\mathrm{g(f(d))}\},\{\mathrm{d}, \mathrm{f(d)}, \mathrm{g(f(d))}\})$ returns nothing.
  $\texttt{done}_4 = \left\{\underline{\mathrm{g(f(e))}} = \mathrm{f(d)}, \underline{\mathrm{g(f(e))}} = \mathrm{T}, \underline{\mathrm{g(f(d))}} = \mathrm{T}\right\}$
  $\texttt{cache}_4 = (\{\mathrm{g(f(e))}, \mathrm{g(f(d))}\}, \{\mathrm{e}, \mathrm{f(e)}, \mathrm{d}, \mathrm{f(d)}, \mathrm{g(f(d))}\})$
  $\texttt{done}_2 = \left\{\underline{\mathrm{g(f(e))}} = \mathrm{T}\right\}$
  $\texttt{cache}_2 = (\{\mathrm{f(e)}, \mathrm{g(f(c))}, \mathrm{g(f(d))}\}, \{\mathrm{e}, \mathrm{f(e)}, \mathrm{c}, \mathrm{f(c)}, \mathrm{g(f(c))}, \mathrm{d}, \mathrm{f(d)}, \mathrm{g(f(d))}\})$

- n_5 has nothing to be done.
  $\texttt{done}_5 = \emptyset$

- n_6 requests $\{b, c, d\}, \{b, c, d, u, f(u), g(f(u))\}$ and receives (after relativisation):
  $\left\{\neg c_1 \vee \underline{g(f(b))} = T, c_1 \vee \neg c_2 \vee \underline{g(f(c))} = T, c_1 \vee c_2 \vee \underline{g(f(d))} = T\right\}$
  It then derives
  $\left\{\neg c_1 \vee \underline{g(f(u))} = T, c_1 \vee \neg c_2 \vee \underline{g(f(u))} = T, c_1 \vee c_2 \vee \underline{g(f(u))} = T\right\}$
  Superposing all three with $\underline{g(f(u))} \neq T$ would give us:
  $\left\{\underline{\neg c_1}, c_1 \vee \underline{\neg c_2}, c_1 \vee \underline{c_2}\right\}$
  From which we derive $\Box$ using resolution.

Observations: The proof is comparable to the flat VC superposition proof, but the search space is smaller - we have avoided several redundant inferences. The overvhead of requests is significant, and we can see that many requests return an empty response. The use of scoping would be useful to reduce the number of requests - we can avoid requests with the sub-term u from n_6 as u does not occur at all in any predecessor, and similarly requests for super-terms of b,c,d could be blocked where they are out of scope, thus reducing the cache at n_2 to be, for a saturated CFG:
$\text{cache}_2 = (\{f(e), g(f(e))\}, \{e, f(e), g(f(e))\})$
Rather than:
$\text{cache}_2 = (\{f(e), b, c, d, g(f(u)), g(f(c)), g(f(d))\},$
$\{e, f(e), g(f(e)), c, f(c), g(f(c)), d, f(d), g(f(d)), u, f(u), g(f(u))\})$
We discuss scoping in chapter 5

In the above example there was no opportunity for a join, although, if we were to apply our simplification inferences, we would replace (at n_6):
$\left\{\neg c_1 \vee \underline{g(f(u))} = T, c_1 \vee \neg c_2 \vee \underline{g(f(u))} = T, c_1 \vee c_2 \vee \underline{g(f(u))} = T\right\}$
With (by the rules $\text{simp}_{\text{res}}, \text{simp}_{\text{res2}}$)
$\left\{\underline{g(f(u))} = T\right\}$
Which is essentially a join that reduces the number of superposition inference in the proof - we only superpose the simplified clause with $g(f(u)) \neq T$ once, rather than the three superposition derivations above.

Modular proofs: We represent the above modular proof in a table form, which we use for further examples below. The modular proof for our example is in figure 4.8. In the proof, the first column is the clause index (local to the CFG-node, in order of derivation). The second column is the clause. The third column lists the premises used to derive the clause - no premises means the clause is part of the original program, two parenthesized premises mean the clause is the result of a superposition inference, one parenthesized premise means the clause is the result of equality resolution or equality factoring inferences and a bracketed premise means the clause has been propagated to the current CFG-node as the result of a request, where the requested term is inside the brackets. The fourth and fifth columns are related to simplifications and described in the next section. Each node performs the proof locally, where the only interaction are requests and propagated clauses.

idx	clause	premises [request]	simplified by	subsumes
$\mathbf{n_0}$				
(1)	$\underline{g(f(e))} = T$			
$\mathbf{n_1}$	$(\mathbf{n_0})$			
(1)	c_1			
(2)	$\underline{g(f(e))} = T$	[e]		
$\mathbf{n_2}$	$(\mathbf{n_0})$			
(1)	$\neg c_1$			
(2)	$\underline{g(f(e))} = T$	[e]		
$\mathbf{n_3}$	$(\mathbf{n_2})$			
(1)	c_2			
(2)	$\underline{f(e)} = f(c)$			
(3)	$\underline{g(f(e))} = T$	[e]		
(4)	$\underline{g(f(c))} = T$	(2)(3)		
$\mathbf{n_4}$	$(\mathbf{n_2})$			
(1)	$\neg c_2$			
(2)	$\underline{g(f(e))} = g(f(d))$			
(3)	$\underline{g(f(e))} = T$	[e]		
(4)	$\underline{g(f(d))} = T$	(2)(3)		
$\mathbf{n_5}$	$(\mathbf{n_3}, \mathbf{n_4})$			
(1)	$\underline{g(f(c))} = T$	[c]		
(2)	$\underline{g(f(d))} = T$	[d]		
$\mathbf{n_6}$	$(\mathbf{n_1}, \mathbf{n_5})$			
(1)	$\underline{b} = u$			
(2)	$\underline{c} = u$			
(3)	$\underline{d} = u$			
(4)	$\underline{g(f(u))} \neq T$			
(5)	$\neg c_1 \vee \underline{g(f(b))} = T$	[b]		
(6)	$c_1 \vee \neg c_2 \vee \underline{g(f(c))} = T$	[c]		
(7)	$c_1 \vee c_2 \vee \underline{g(f(d))} = T$	[d]		
(8)	$\neg c_1 \vee \underline{g(f(u))} = T$	(1)(5)		
(9)	$c_1 \vee \neg c_2 \vee \underline{g(f(u))} = T$	(2)(6)		
(10)	$c_1 \vee c_2 \vee \underline{g(f(u))} = T$	(3)(7)		
(11)	$\neg c_1$	(4)(8)		
(12)	$c_1 \vee \underline{\neg c_2}$	(4)(9)		
(13)	$c_1 \vee \underline{c_2}$	(4)(10)		
(14)	c_1	(12)(13)		
(15)	$\bot$	(11)(14)		

Figure 4.8: Modular proof for figure 4.6
Each node number is followed by the set of predecessors.
The literal ordering is $e \succ d \succ c \succ b \succ a \succ u$.
We do not distinguish the lhs and rhs of requests for simplicity.
Blue premises in square brakcets are imported by a local request
Cyan premises in square brackets are imported by a successor request
Parenthesized numbers are premises for an inference

unit	$\dfrac{\neg A \quad \cancel{C \vee A}}{C}$	
taut	$\dfrac{\cancel{C \vee A \vee \neg A}}{}$	
$\text{taut}_=$	$\dfrac{\cancel{C \vee s = s}}{}$	
sub	$\dfrac{C \quad \cancel{C \vee D}}{}$	
simp_{res}	$\dfrac{\cancel{C \vee A} \quad \cancel{C \vee \neg A}}{C}$	
$\text{simp}_{\text{res2}}$	$\dfrac{C \vee A \quad \cancel{C \vee D \vee \neg A}}{C \vee D}$	
$\text{simp}_=$	$\dfrac{l = r \quad \cancel{C}}{C\left[r\right]_p}$	$l = C\vert_p$ $l \succ r$ $C \succ l = r$

Figure 4.9: simplification rules
$\cancel{C}$ denotes that the premise C is redundant after the addition of the conclusion to the clause-set and hence can be removed.

4.1.2 Redundancy elimination

In this section we show how our algorithm can benefit from redundancy elimination and simplification inferences more than the flat VC. We will also compare to the per-path VC. There are two main points: The first idea is that we have more chances to apply unit propagation and simplification as our clauses carry less baggage of path conditions. The second idea is that, for a pair of clauses C, D that share a path (C being earlier on the path) and have a valid inference, if there is a clause E on the path between C and D, that either subsumes or simplifies C so that it is no longer relevant for D, then we can avoid the inference, while a prover running on a flat VC cannot take advantage of these simplifications without allowing path literals to be maximal - which essentially amounts to enumerating execution paths. For the per-path VC we can see that more simplifications can be performed, as it can treat the clauses on a path as a set and hence also subsume clauses regardless of the location of the subsuming clause in the execution order - which helps explain, to some degree, the success of lazy-CNF SMT solvers in proving VCs that are encoded directing the solver to explore paths.

In order to demonstrate the differences we use only the simplification inferences depicted in 4.9 , applying them eagerly. We have omitted some of the rules for simplicity of presentation, but the principles remain the same. Specifically, we have omitted rewriting by unit ground equalities (demodulation) - as it would be handled by EC-graphs in our technique (detailed in a later sub-section) - including demodulation would give our technique an even more significant advantage. In our algorithm, we simplify each term w.r.t. $\texttt{todo}_n \cup \texttt{done}_n$ before it is added to one of these sets, except for simplifications that modify the maximal literal.

```
n0 :
assume g(f(e))=T
if (c1)
   n1 :
   assume b=a∨f(e)=f(c)
   assume b≠a∨f(e)=f(d)
else
   n2 :
   assume e=c
n3 :
assume b=a
if (c2)
   n4 :
   assume c=u
   assume d=u
   assert g(f(u))=T
else
   n5 :
   assert g(a)≠g(b)
```

Figure 4.10: Simple example for VC comparison with simplification
The ordering is $e \succ d \succ c \succ b \succ a \succ u \succ T$.
The node n_5 was added to ensure all joins are binary.

In figure 4.10 we present a small example program. The program has six program points, n_0 to n_5, of which n_4, n_5 are assertion (goal) nodes. In figure 4.11 we show the proof in our technique:

The fourth column lists clauses used to simplify the result of the inference - in this example either by unit subsumption or unit propagation. The fifth column lists earlier clauses that are subsumed by the clause.

The useless inferences are $n_1.(4)$ and $n_1.(6)$, as $n_1.(4)$ is a tautology and $n_1.(6)$ is subsumed when propagated to n_3 - as $n_3.(2)$. At n_3 we were able to subsume one clause - $n_3.(2)$ - by the unit clause $n_3.(1)$, and simplify another clause $n_3.(4)$ by unit propagation. The set of clauses in each node is saturated with respect to the calculus and to requests, so no further derivations are possible. We compare our proof to the proof of a flat VC of the above program, depicted in figure 4.12

idx	clause	premises [request]	simplified by	subsumes
$\mathbf{n_0}$				
(1)	$\underline{g(f(e))} = T$			
$\mathbf{n_1}$	$(\mathbf{n_0})$			
(1)	c_1			
(2)	$b = a \lor \underline{f(e)} = f(c)$			
(3)	$b \neq a \lor \underline{f(e)} = f(d)$			
(4)	~~$b \neq a \lor b = a \lor \underline{f(d)} = f(c)$~~	(2)(3)	taut	
(5)	$\underline{g(f(e))} = T$	[f(e)]		
(6)	$b = a \lor \underline{g(f(c))} = T$	(2)(5)		
(7)	$b \neq a \lor \underline{g(f(d))} = T$	(3)(5)		
$\mathbf{n_2}$	$(\mathbf{n_0})$			
(1)	$\neg c_1$			
(2)	$\underline{e} = c$			
(3)	$\underline{g(f(e))} = T$	[e]		
(4)	$\underline{g(f(c))} = T$	(2)(3)		
$\mathbf{n_3}$	$(\mathbf{n_1}, \mathbf{n_2})$			
(1)	$\underline{b} = a$			
(2)	~~$\neg c1 \lor \boxed{b = a} \lor \underline{g(f(e))} = T$~~	[c]	(1) sub	
(3)	$c1 \lor \underline{g(f(c))} = T$	[c]		
(4)	$\neg c1 \lor$ ~~$b \neq a \lor$~~ $\underline{g(f(d))} = T$	[d]	(1)	
$\mathbf{n_4}$	$(\mathbf{n_3})$			
(1)	c_2			
(2)	$\underline{c} = u$			
(3)	$\underline{d} = u$			
(4)	$\underline{g(f(u))} \neq T$			
(5)	$c1 \lor \underline{g(f(c))} = T$	[c]		
(6)	$\neg c1 \lor \underline{g(f(d))} = T$	[d]		
(7)	$c1$ ~~$\lor \underline{g(f(u))} = T$~~	(2)(5)	(4)	
(8)	~~$\neg c1 \lor \underline{g(f(u))} = T$~~	(3)(6)	(4)(7)	
(9)	$\perp$	(8)		
$\mathbf{n_5}$	$(\mathbf{n_3})$			
(1)	$\neg c_2$			
(2)	$\underline{g(b)} \neq g(a)$			
(3)	$\underline{b} = a$	[b]		
(4)	$\underline{g(a)} \neq g(a)$	(2)(3)		
(5)	$\perp$	(4)		

Figure 4.11: Modular VC for figure 4.10 with simplifications
maximal terms are underlined
The literal ordering is $e \succ d \succ c \succ b \succ a \succ u \succ T$

(1)	$\neg n_0 \vee \underline{g(f(e))} = T$
(2)	$\underline{\neg n_1} \vee n_0$
(3)	$\neg n_1 \vee b = a \vee \underline{f(e)} = f(c)$
(4)	$\neg n_1 \vee b \neq a \vee \underline{f(e)} = f(d)$
(5)	$\underline{\neg n_2} \vee n_0$
(6)	$\neg n_2 \vee \underline{e} = c$
(7)	$\underline{\neg n_3} \vee n_1 \vee n_2$
(8)	$\neg n_3 \vee \underline{b} = a$
(9)	$\underline{\neg n_4} \vee n_3$
(10)	$\neg n_4 \vee \underline{c} = u$
(11)	$\neg n_4 \vee \underline{d} = u$
(12)	$\neg n_4 \vee \underline{g(f(u))} \neq T$
(13)	$\underline{\neg n_5} \vee n_3$
(14)	$\neg n_5 \vee \underline{g(b)} \neq g(a)$
(15)	$n_4 \vee \underline{n_5}$

Figure 4.12: Simple flat VC for figure 4.10
maximal terms are underlined

The proof steps are given in figure 4.13. The proof performs the inference steps in the order of the premises - essentially we saturate every clause with inferences with all its predecessors. Simplifications we perform eagerly, so a unit clause is often followed by several simplified versions of clauses. Clauses deleted by subsumption are listed in the last column. This choice of proof order is derived from the naive implementation of a theorem prover, and is not meant to represent the state of the art, rather we wanted to represent here the much larger search space of a flat VC proof vs. a local proof, and also the larger clauses that participate in such a proof.

We have used alphabetically suffixed clause indices to indicate a simplified version of an earlier clause. Evidently, the flat proof is much larger than the sum of local proofs above (although the local proof has the overhead of propagation). We have marked the useful (part of the refutation) inferences with blue indices, and we can see that there are many more redundant inferences than in the local case:

- Clause (16) is the equivalent to n_1.(6), and so is not blocked also in our local proof, and similarly clause (18)
- Clauses (20) and (21) are blocked in the local proof as the premises come from the parallel sides of a branch as we have seen in the previous section - note that (21) is subsumed later (at (26a)) once one of the parallel paths has been eliminated
- Clause (16) is prevented from participating in any further inferences as it is subsumed by (n_3.(1)) in the local proof, hence blocking the derivation of the clauses (24) and (27)

- Similarly, (28) and (29) are not even considered in the local proof as (21) is blocked

Clause (31) signifies that every path must pass through n_3 - which we derived a-priori in the local proof from the DAG structure.

We can see that the flat proof is significantly larger even for such a small program, mostly because it does not take advantage of existing information about the CFG structure, and also because it misses many options for simplifying inferences. This difference in proof size usually grows with the program. The overhead for local proofs is the request mechanism. The term-indexing overhead is usually smaller than in the flat proof as indices can be maintained for each CFG-node separately. A request in a large CFG from a deep leaf to the root can incur a lot of overhead not present in the flat proof, and hence we try to minimize this overhead by caching requests and responses, overapproximating requests and joining, so that each node propagates each request at most once. As described before, scoping can also reduce the request overhead dramatically.

Comparison with per-path VCs

We compare now the efficiency of our local proof to a VC generated for each possible root-to-assertion path in the program. The reason for discussing this comparison is that it factors away the CFG part of the problem and hence shows a proof that is, in a sense, pure. A per-path VC is also closely related to the set of sub-problems faced by an SMT solver or a solver such as [Vor14] based on a SAT solver for the propositional part of the problem, and a superposition solver for each propositionally satisfying assignment.

As is, a per-path VC is not a practically scalable VC as the number of paths is potentially exponential in the size of the CFG, however it represents, in a sense, the opposite side of the spectrum to the flat VC, as it allows short simple proofs for each path, but no opportunity for sharing of work between the proofs of different paths.
SMT solvers mitigate this problem mostly using the extension of CDCL ([SS99]) to GFOLE - essentially generalizing a conclusion learned in the proof of one path so that it can be used on other paths.
However, as has been shown in e.g. [BDdM08] and have been discussed in chapter 3, CDCL is somewhat limited in the clauses it can learn in GFOLE.
As can be seen in figure 4.14, in the per-path VC the clause $\mathrm{b} = \mathrm{a}$ that appears on all paths was able to block the unnecessary derivations that our local derivations could not, by subsuming the clause $\mathrm{b} = \mathrm{a} \vee \underline{\mathrm{f(e)}} = \mathrm{f(c)}$. Also, there is no need for path conditions, as each path is considered separately.

The disadvantage of this approach is apparent in the proof of the path $\mathbf{n_0.n_1.n_3.n_5}$ - here derivations (3a) and (6) are useless, and they are duplicated from the proof of $\mathbf{n_0.n_1.n_3.n_4}$, where they are not useless (so a CDCL solver that first tries $\mathbf{n_0.n_1.n_3.n_5}$ will effectively not learn from the equivalent of the derivation - operation on the CC graph). In a larger program the amount of duplication could grow exponentially with the number of non-nested branches. In our local proof we were able to factor these derivations so that they are performed only once - each derivation is assigned to a specific node and guaranteed not to be performed in any other node (as long as the premises do not occur more than once in the program).

idx	clause	premises	simplified by	subsumes
(1)	$\neg n_0 \vee \underline{g(f(e))} = T$			
(2)	$\underline{\neg n_1} \vee n_0$			
(3)	$\neg n_1 \vee b = a \vee f(e) = f(c)$			
(4)	$\neg n_1 \vee b \neq a \vee \underline{f(e)} = f(d)$			
(5)	$\underline{\neg n_2} \vee n_0$			
(6)	$\neg n_2 \vee \underline{e} = c$			
(7)	$\underline{\neg n_3} \vee n_1 \vee n_2$			
(8)	$\neg n_3 \vee \underline{b} = a$			
(9)	$\underline{\neg n_4} \vee n_3$			
(10)	$\neg n_4 \vee \underline{c} = u$			
(11)	$\neg n_4 \vee \underline{d} = u$			
(12)	$\neg n_4 \vee \underline{g(f(u))} \neq T$			
(13)	$\underline{\neg n_5} \vee n_3$			
(14)	$\neg n_5 \vee \underline{g(b)} \neq g(a)$			
(15)	$n_4 \vee \underline{n_5}$			
(16)	$\neg n_0 \vee \neg n_1 \vee b = a \vee \underline{g(f(c))} = T$	(1) (3)		
(17)	$\neg n_0 \vee \neg n_1 \vee b \neq a \vee \underline{g(f(d))} = T$	(1) (4)		
(18)	~~$\neg n_1 \vee b = a \vee b \neq a \vee f(d) = f(c)$~~	(3) (4)	taut	
(19)	$\neg n_2 \vee \underline{g(f(c))} = T$	(1) (6)		
(20)	~~$\neg n_1 \vee \neg n_2 \vee b = a \vee f(c) = f(c)$~~	(3) (6)	$\text{taut}_=$	
(21)	$\neg n_1 \vee \neg n_2 \vee b \neq a \vee \underline{f(d)} = f(c)$	(4) (6)		
(22)	$\neg n_3 \vee \underline{\neg n_5}$~~$\vee\ g(a) \neq g(a)$~~	(8) (14)	$\text{res}_=$	
(23)	$n_3 \vee \underline{n_4}$	(13) (15)		
(24)	$\neg n_0 \vee \neg n_1 \vee \neg n_4 \vee b = a \vee \underline{g(f(u))} = T$	(10) (16)		
(25)	$\neg n_0 \vee \neg n_1 \vee \neg n_4$~~$\vee\ b \neq a$~~$\vee\ \underline{g(f(u))} = T$	(11)(17)	(24)simp_{res}	(24)
(26)	$\neg n_2 \vee \neg n_4 \vee \underline{g(f(u))} = T$	(10) (19)		
(27)	~~$\neg n_0 \vee \neg n_1 \vee \neg n_2 \vee b = a \vee T = T$~~	(16) (19)	$\text{taut}_=$	
(28)	$\neg n_1 \vee \neg n_2 \vee \neg n_4 \vee b \neq a \vee \underline{f(c)} = f(u)$	(11) (21)		
(29)	~~$\neg n_0 \vee \neg n_1 \vee \neg n_2 \vee b \neq a \vee \underline{g(f(c))} = T$~~	(17) (21)	(19)sub	
(30)	$\neg n_3 \vee n_4$	(22)(15)		
(31)	n_3	(23)(9)		(9)(23)
(7a)	$n_1 \vee \underline{n_2}$	(7)	(31)	(7)
(8a)	$\underline{b} = a$	(8)	(31)	(3)(8)(16)
(22a)	$\underline{\neg n_5}$	(22)	(31)	(13)(14) (22)
(30a)	$\underline{n_4}$	(30)	(31)	(15)(30)
(4a)	$\neg n_1 \vee \underline{f(e)} = f(d)$	(4)	(8a)	(4)
(17a)	$\neg n_0 \vee \neg n_1 \vee \underline{g(f(d))} = T$	(17)	(8a)	(17)
(10a)	$\underline{c} = u$	(10)	(30a)	(10)
(11a)	$\underline{d} = u$	(11)	(30a)	(11)
(12a)	$\underline{g(f(u))} \neq T$	(12)	(30a)	(12)
(25a)	$\neg n_0 \vee \neg n_1$	(25)	(30a)(12a)	(25)
(26a)	$\neg n_2$	(26)	(30a)(12a)	(5) (6) (19) (21)(26)(28)
(7b)	$\underline{n_1}$	(7a)	(26a)	(7a)
(2a)	$\underline{n_0}$	(2)	(7b)	(2)
(4b)	$\underline{f(e)} = f(d)$	(4a)	(7b)	(4a)
(17b)	$\underline{g(f(d))} = T$	(17a)	(2a)(7b)	(17a)
(25b)	$\perp$	(25a)	(2a)(7b)	

Figure 4.13: Proof for flat VC of figure 4.10
Maximal terms are underlined.
Blue indices are clauses used in the proof.

idx	clause	premises	simplified by	subsumes
	$\mathbf{n_0.n_1.n_3.n_4}$			
(1)	$\underline{g(f(e))} = T$			
(2)	$b = a \vee \underline{f(e)} = f(c)$			
(3)	$b \neq a \vee \underline{f(e)} = f(d)$			
(4)	$\underline{b} = a$			(2)
(5)	$\underline{c} = u$			
(6)	$\underline{d} = u$			
(7)	$\underline{g(f(u))} \neq T$			
(3a)	$\underline{f(e)} = f(d)$	(3)	(4)	(3)
(8)	$\underline{g(f(d))} \neq T$	(3a)(1)		
(9)	~~$\underline{g(f(u))} = T$~~	(6)(8)	(7)	
	$\mathbf{n_0.n_2.n_3.n_4}$			
(1)	$\underline{g(f(e))} = T$			
(2)	$\underline{e} = c$			
(3)	$\underline{b} = a$			
(4)	$\underline{c} = u$			
(5)	$\underline{d} = u$			
(6)	$\underline{g(f(u))} \neq T$			
(7)	$\underline{g(f(c))} = T$	(1)(2)		
(8)	~~$\underline{g(f(u))} = T$~~	(7)(4)	(6)	
	$\mathbf{n_0.n_1.n_3.n_5}$			
(1)	$\underline{g(f(e))} = T$			
(2)	$b = a \vee \underline{f(e)} = f(c)$			
(3)	$b \neq a \vee \underline{f(e)} = f(d)$			
(4)	$\underline{b} = a$			(2)
(5)	$\underline{g(b)} \neq g(a)$			
(3a)	$\underline{f(e)} = f(d)$	(3)	(4)	(3)
(6)	$\underline{g(f(d))} \neq T$	(3a)(1)		
(7)	~~$g(a) \neq g(a)$~~	(4)(5)	$\mathrm{res}_=$	
	$\mathbf{n_0.n_2.n_3.n_5}$			
(1)	$\underline{g(f(e))} = T$			
(2)	$\underline{e} = c$			
(3)	$\underline{b} = a$			
(4)	$\underline{g(b)} \neq g(a)$			
(5)	$\underline{g(f(c))} = T$	(1)(2)		
(6)	~~$g(a) \neq g(a)$~~	(3)(4)		

Figure 4.14: Simple per-path VC proof for figure 4.10
The proof is shown separately for each path.

Another disadvantage of the per-path VC approach (and hence also SAT/SMT based approaches) is that few non-propositional joins can be performed, and hence a lot of duplication of effort can happen, this is not evident in the above example, but is discussed in the next sub-section. Techniques such as CDCL can mitigate a lot of the duplication, but mostly propositional duplication.

However, the main reason that we do not follow the per-path approach is that it is essentially a DFS proof-search, and hence does not extend readily to the non-ground case. In order to generalize a clause learned in the proof of one path, CDCL requires a conflict (i.e. a proof) on that path. Even if this requirement can be relaxed, it is not clear how a path-based approach can be extended to gradual verification using incrementally stronger fragments, as the verification algorithm will have to either remember the state of an exponential number of not-yet-verified paths, or reevaluate each path with a stronger fragment, duplicating much work. Our objective is to be able to search for proofs in fragments of increasing strength, without losing the results of previous fragments. Our proof-search is, in a sense, of the BFS approach, as we are not required to reach a refutation on any branch in order to ensure progress - progress is achieved by moving a clause to the `done` set.

Comparison Summary:
The table below compares the statistics for the proofs above:

	Flat	**Modular**	**Per-path**
Total VC clauses	15	13	22
Total inferences	32	17	14
Useful inferences	16	14	10
% unused inferences	50%	18%	28%
Expensive inferences	16	7	8
Useful expensive inferences	8	5	6
% unused expensive inferences	50%	28%	25%

We counted the number of clauses and inferences in the proof produced by each technique, and the proportion of useless inferences. We also counted the number of expensive inferences, which are the binary inferences that are not simplifications. Obviously this table only applies for the specific program and instance of each technique, but we believe it shows some trends that are valid also for the general case. Mainly, we can see that the per-path VC is larger, but can avoid some inferences that the flat VC cannot. The flat VC is, on the other hand, smaller, but produces many useless clauses. The VC for our technique is about the same size as the flat one, but we were able to avoid several expensive inferences. Obviously, this table does not take into account the cost of requests and clause-propagation.

Our local VC is an attempt to combine the advantages of both extremes we have seen - allowing smaller and simpler local VCs while avoiding the duplicate effort and exponential explosion of the per-path or SAT based techniques.

4.1.3 Improvements to the base algorithm

Simplification requests

As we have seen in the example above, early subsumption and other deletion and simplification inferences are a powerful tool in reducing the proof search

```
n0 :
assume g(f(e))=T
assume b=a
if (c1)
   n1 :
   assume b=a∨f(e)=f(c)
   assume b≠a∨f(e)=f(d)
else
   n2 :
   assume e=c
n3 :
assume c=u
assume d=u
assert g(f(u))=T
```

Figure 4.15: Example for forward propagation of unit clauses for simplification

space.
However, we did not yet show a mechanism to propagate unit-clauses that can simplify or delete clauses in later CFG-nodes - for example, if we were to modify our previous example by moving $b = a$ to the beginning (and removing one assertion) - depicted in figure 4.15.

In figure 4.15 our local proof is not be able to subsume $b = a \vee f(e) = f(c)$ and simplify $b \neq a \vee f(e) = f(d)$ at n_1 as the unit clause $b = a$ is not available there, and is not propagated as its maximal literal b is not requested by any successor. In order to benefit from such cases, we add atom requests to the maximal term requests we have seen before. Each node, when requesting the maximal term in a clause, also adds to the request the set of atoms of non-maximal literals. The reason we request the atom and not the literal is that both polarities of the atom are helpful as a unit clause - one polarity could subsume the clause and the other could simplify it by unit propagation. The response to this query adds only unit clauses that include an atom from the requested set. At join points we only propagate such a unit clause if it is found in the response of both join sides - described in section 4.3.

The modified request code is shown in figure 4.16. As can be seen, we can mostly use the existing request mechanism. Also, as before, no derivations are performed between two imported clauses (as they are sent directly to the `done` set), and hence we do not cause any duplication of effort. We do, however, incur some overhead in requesting more propagations - in our experiments we have found this feature to be largely beneficial. Note that we only propagate unit clauses forward for simplification, we never propagate a clause so it can be simplified.

Reverse propagation

In the example in figure 4.10, in the local proof, we have seen that a clause $b = a$ at the node n_3 was able to subsume a clause at n_2 and so prevent some useless inferences, but was not able to prevent inferences between clauses at n_2, although *every path from the root to an assertion that includes* n_2 *includes also* n_3. In our setting, there are two fundamental ways in which to allow

```
Request(Cs : Set[Clause])
   ...
   As:= $\cup_{C \in Cs}$ atoms(C)

Request.add(other : Request)
   ...
   As $\cup$:= other.As

Request.remove(other : Request)
   ...
   As \:= other.As

isRelevant(r:Request,C : Clause)
   return
      ...
         $\vee$
      $|C|$ = 1 $\wedge$ atoms(C)$\subseteq$r.As

atoms(C : Clause)
   return {A | A$\in$C $\vee$ $\neg$A$\in$C}
```

Figure 4.16: Recursive import request including literals.
A request consists of equality and inequality maximal terms, and a set of atoms `As`.
A clause matches a request if there is a valid inference in the calculus, and in addition if there is a unit clause that can be used in a simplifying inference.

this subsumption to take place - either propagate the clause to be subsumed forward or reverse-propagated the subsuming unit clause backward. Note that in our example the clause that was subsumed - $b = a \vee \underline{f(e)} = f(c)$ - was not eligible to be propagated forward as its maximal term is not relevant to any transitive successor, only the result of an inference where the clause is a premise - $b = a \vee \underline{g(f(c))} = T$ - is eligible to forward propagation.

As a VC transformation, propagating a clause backwards is sound and complete if the clause holds in *all* direct successors of the node to which it is propagated, but it is not a conservative transformation - if the user wants to add a property (for debugging etc.) at a node, we add another successor for the node, where the clause might not hold - hence we cannot check the property incrementally. We have not experimented with reverse propagation.

Predicate symbols

It is sometimes preferable to use predicate symbols rather than encoding them as Boolean functions. The additional inference rules for predicates are the standard ordered resolution and a rule similar to negative superposition as shown in figure 4.17. Adding predicate symbols requires that we can send requests for maximal predicates. These requests are similar to simplification requests, except that we only request one polarity of a literal. In addition, a maximal predicate literal includes in its rhs set all its top terms.

Consider the following example in figure 4.18. There is no possible derivation between the two clauses. If we were to use the function encoding we would

$$\mathrm{sup_P}\quad \frac{C \vee \underline{l} = r \quad P(\bar{s}) \vee D}{C \vee P(\bar{s}\,[r]_p) \vee D} \quad \begin{array}{l} l = \bar{s}|_p \\ l \succ r, l = r \succ C \\ P(\bar{s}) \succ D \end{array}$$

$$\mathrm{res_P}\quad \frac{C \vee \underline{P(\bar{s})} \quad \neg\underline{P(\bar{s})} \vee D}{C \vee D} \quad \begin{array}{l} P(\bar{s}) \succ C \\ \neg P(\bar{s}) \succ D \end{array}$$

Figure 4.17: Ground superposition for predicates

```
n₀ :
assume CVP(c)
...
n₁ :
assume DVP(c)
```

Figure 4.18: Example for predicate symbols

get: $\left\{C \vee \underline{f_P(c)} = T, D \vee \underline{f_P(c)} = T\right\}$. Although we cannot derive the tautology $C \vee D \vee T = T$ because of the side conditions of positive superposition, the earlier clause will still be propagated needlessly. Using predicate symbols, the request we send is includes both the predicate symbol and the polarity, hence the clause $C \vee P(c)$ won't be propagated. It is also possible to find, sometimes, that a certain predicate only occurs in one polarity, and eliminate all clauses that include it (pure literal elimination). We can calculate, at each CFG-node, a set of predicate polarity pairs that occur in any transitive predecessor and similarly for any transitive successor, and eliminate any clause that contains a literal for which there is no opposite literal.

```
n0 :
if (c1)
    n1 :
    if (c2)
        n2 :
    else
        n3 :
    n4 :
else
    n5 :
n6 :
if (c3)
    n7 :
else
    n8 :
n9 :
```

Figure 4.19: Example for branch conditions on a path

4.2 Completeness

In this section we sketch the completeness argument for our verification algorithm, and show what is required for completeness from the clause propagation mechanism and calculus.

As a clause is propagated, some path literals are added to it to ensure soundness - specifically, at a join point that joins a branch with the branch condition c, if the clause D is propagated from the **then** joinee it will be propagated as $\neg c \vee D$, and similarly from the **else** joinee as $c \vee D$.
When a clause is propagated on a path it may "collect" several such branch conditions. As these branch conditions are always strictly smaller (in $\succ$) than any non-branch literal in D (discussed in section 4.4), these additional literals do not affect a calculus where the validity of an instance of an inferences rule is determined only by maximal literals, as in the superposition calculus, until all non branch condition literals are eliminated and we are left with an ordered resolution proof. We define the set of branch literals that a clause collects as follows:
Given a path P in the CFG starting at the root, the set of path conditions on P, **pc**(P), is the set of branch conditions taken on the path *for branches that are joined on the path.* A clause path prefix for a path P in the CFG is a clause

$$\bigvee_{A \in S} \neg A \text{ for some } S \subseteq pc(P).$$

Intuitively, a clause-path-prefix for P is a clause made of only branch literals that do not hold in P. We do not use simply $S = pc(P)$ in the above definition as simplifications (e.g. unit propagation) and joins can generalize the clause-path-prefix part of a clause to include more paths (reducing the number of literals).

For example, in figure 4.19:
For the path $P = n_0.n_1.n_2$, $pc(P) = \emptyset$ as no branches are joined on the path.
For the path $P = n_0.n_1.n_2.n_4.n_6.n_7$, $pc(P) = \{c_1, c_2\}$.
For the path $P = n_0.n_1.n_3.n_4.n_6.n_8.n_9$, $pc(P) = \{c_1, \neg c_2, \neg c_3\}$.

Evidently the disjunction of the negation of any subset of pc(P) implies that the path P was not taken. We use joins in order to eliminate some of the literals in ¬pc(P).

A property of clause-path-prefixes is as follows: Given a node n, if we select a set of clauses S s.t. each path from the root to the node has a clause-path-prefix in S, then S is inconsistent. The reason is that any model of the set of clauses defines a value for each branch condition and hence defines a path from the root to the node.

We say that a clause occurs at a node if it is in the `done` set of the node, and occurs on a path if it occurs at a node on the path.

Our reasoning for completeness is intuitively as follows: for any path P in the CFG that starts at the root, for any instance of a binary inference with premises C,D in the calculus for which both premises occur on the path where the earliest (in distance from the root) occurrence of C is at n_C and of D at n_D, and w.l.o.g. n_C is earlier, C is propagated (relativized) to n_D and the inference takes place at n_D (relativized). The idea is that if a node is infeasible (w.l.o.g. one of its predecessors is feasible) then for any path reaching the node from the root the empty clause can be derived from the clauses occurring on the path (as the calculus is complete), and hence, if the propagation criterion is complete (always propagates a clause relevant for an inference), the algorithm derives a relativized empty clause which is a clause-prefix-path on some node on the path (the earliest node that is infeasible on the path). The set of clause-prefix-paths for all paths reaching a node is inconsistent and hence the algorithm derives the empty clause at the infeasible node. Note that a join node with one infeasible predecessor has to propagate the relativized empty clause - in practice we simply eliminate the infeasible predecessor and trim the CFG, and the node stops being a join.

4.3 Joins

In this section we show how our algorithm takes advantage of opportunities to simplify clauses at a join point. When our request mechanism propagates clauses to a join node, the algorithm described above simply adds a relativized version of each clause from each side of the join. If a clause is propagated from before the branch corresponding to the join, this effectively duplicates the clause and after several such joins in sequence we can get an exponential explosion in the number of relativized clauses. While the ordering we use for superposition (in section 4.4) avoids some of this problem, the problem does occur with simplifications, when we propagate clauses for instantiation, and when we use superposition in combination with EC-graphs, as described in section A.2. In addition, joins can shorten proofs by reducing the number of clauses even if the clauses do not originate from before the corresponding branch. For the branch-join problem, a simple solution is to propagate clauses directly between a branch and a join, but this is not compatible with scoping - a clause before the branch can use earlier DSA versions of terms not present at the join.

The main idea is that, if we have a set of clauses at each joinee that need to be propagated to the join, we try to match a clause from one side with a clause

```
n0 :
assume c≠b
if (c1)
   n1 :
   assume b=a
   //propagated: c≠b
   //simplified: c≠a
else
   n2 :
   //propagated: c≠b
n3 :
//Without join:
   //propagated:
   // ¬c1 ∨ c≠a
   // ¬c1 ∨ b=a
   //  c1 ∨ c≠b
//With join:
   //propagated:
   // ¬c1 ∨ b=a
   // c≠b
n4 :
assume b=a
assert c=a //negated c≠b
```

Figure 4.20: Example for simple clause join

from the other side, to find a clause that is equivalent to both clauses at the join, using the known equalities and unit clauses at both joinees and the join.

A very simple join example is given in figure 4.20. The clause $c \neq b$ was simplified to $c \neq a$ at n_1 which, without clause join, causes n_3 to propagate three clauses, while clause join allows us to propagate only two.

A slightly more elaborate example is given in figure 4.21 and the modular proof is given in figure 4.22 - we can see that a part of the proof is essentially duplicated at n_4. At n_3, the two clauses $\neg c_1 \vee f(a) = a \vee \underline{g(e)} = g(b)$ and $c_1 \vee f(a) = a \vee \underline{g(e)} = g(c)$ cannot be joined to $f(a) = a \vee \underline{g(e)} = g(c)$ or $f(a) = a \vee \underline{g(e)} = g(d)$, but, combined with $\neg c_1 \vee c = b$ they can be both replaced with $f(a) = a \vee \underline{g(e)} = g(c)$ without affecting soundness or completeness.

The proof with joins is shown in figure 4.23. As we can see, at node n_3 the join shortens the proof, and perhaps more importantly, all nodes between the join node and the assertion node have less clauses to handle (propagate, index, simplify, etc.) and the clauses are simpler.

The join problem

The join algorithm for ground clauses receives as input the set of unit equalities at the join, and, per joinee, a set of clauses and a set of unit equalities at the joinee. In our example these are:
Set of unit equalities at n_3 : $\emptyset$
n_1: $\left\{f(a) = a \vee \underline{g(e)} = g(b)\right\}, \{g(d) = g(b), c = b\}$
n_2: $\left\{f(a) = a \vee \underline{g(e)} = g(c)\right\}, \{d = c\}$

```
$n_0$ :
assume f(a)=a∨g(e)=g(d)
if (c1)
   $n_1$ :
   assume g(d)=g(b)
   assume c=b
else
   $n_2$ :
   assume d=c
$n_3$ :
if (c2)
   $n_4$ :
   assume f(a)≠a
   assert g(e)=g(c)
else
   $n_5$ :
   assert true
```

Figure 4.21: Example for clause join

Each joinee term has several *potential join representatives* at the join, determined by the equalities at that joinee and at the join - in our example, the term b at n_1 has two potential representatives at n_3 - $\{b, c\}$, implied by $c = b \in \mathbf{eqs}(n_1)$. Formally, the potential representatives of a term s of joinee p for join n are the set $\{t \mid \mathbf{eqs}(p) \cup \mathbf{eqs}(n) \models s = t\}$. Each joinee term has at least one potential join representative (this changes under scoping and bounded fragments - described in chapters 5 and 6 respectively), and could have infinitely many potential join representatives - for example, for the term a where $\mathbf{eqs}(n) = \{f(a) = a\}$). Having defined potential term join representatives, we can define potential literal representatives analogously. For clauses, we also take into account unit clauses that hold at the joinee and the join, as we want to account for unit propagation (the rule unit) - we discuss this later and for now only account for equalities.

In our example,
$f(a) = a \vee \underline{g(e)} = g(b)$ at n_1 has three potential join representatives at n_3:

$$\left\{f(a) = a \vee \underline{g(e)} = g(b), f(a) = a \vee \underline{g(e)} = g(c), f(a) = a \vee \underline{g(e)} = g(d)\right\}$$

Given a binary join and the input to the clause join algorithm, the clause join problem is that of selecting a subset S of the potential join representatives of clauses from all joinees such that each joinee clause has at least one join representative in S.
The general problem is trivial - we can simply add the relativized clauses from each side. However, the number of clauses can grow quadratically at each join, and makes the proof search-space much larger. Our objective is such an S that minimizes some measure composed of:

- The number of clauses in S

idx	clause	premises [request]	simplified by	subsumes
$\mathbf{n_0}$				
(1)	$f(a) = a \lor \underline{g(e)} = g(d)$			
$\mathbf{n_1}$				
(1)	c_1			
(2)	$g(d) = g(b)$			
(3)	$c = b$			
(4)	$f(a) = a \lor \underline{g(e)} = g(d)$	[g(e)]		
(5)	$f(a) = a \lor \underline{g(e)} = g(b)$	(4)	(2)	(4)
$\mathbf{n_2}$				
(1)	$\neg c_1$			
(2)	$d = c$			
(3)	$f(a) = a \lor \underline{g(e)} = g(d)$	[g(e)]		
(4)	$f(a) = a \lor \underline{g(e)} = g(c)$	(3)	(2)	(3)
$\mathbf{n_3}$				
(1)	$\neg c_1 \lor f(a) = a \lor \underline{g(e)} = g(b)$	[g(e)]		
(2)	$c_1 \lor f(a) = a \lor \underline{g(e)} = g(c)$	[g(e)]		
(3)	$\neg c_1 \lor \underline{c} = b$	[c]		
$\mathbf{n_4}$				
(1)	c_2			
(2)	$\underline{f(a)} \neq a$			
(3)	$\underline{g(e)} \neq g(c)$			
(4)	$\neg c_1 \lor f(a) = a \lor \underline{g(e)} = g(b)$	[g(e)]		
(5)	$c_1 \lor f(a) = a \lor \underline{g(e)} = g(c)$	[g(e)]		
(6)	$\neg c_1 \lor f(a) = a \lor \underline{g(c)} \neq g(b)$	(3)(4)		
(7)	$\neg c_1 \lor \underline{c} = b$	[c]		
(8)	$c_1 \lor \underline{f(a)} = a$	(3)(5)		
(9)	$\neg c_1 \lor \underline{f(a)} = a$	(6)(7)		
(10)	c_1	(2)(8)		
(11)	$\neg c_1$	(2)(9)		
(12)	$\bot$	(10)(11)		
$\mathbf{n_5}$				
(1)	$\bot$			

Figure 4.22: Modular VC for figure 4.21
maximal terms are underlined
The literal ordering is $e \succ d \succ c \succ b \succ a \succ u$
Eager simplification is not applied except for $\text{simp}_=$

idx	clause	premises [request]	simplified by	subsumes
$\mathbf{n_0}$				
(1)	$f(a) = a \vee \underline{g(e)} = g(d)$			
$\mathbf{n_1}$				
(1)	c_1			
(2)	$g(d) = g(b)$			
(3)	$c = b$			
(4)	$f(a) = a \vee \underline{g(e)} = g(d)$	[g(e)]		
(5)	$f(a) = a \vee \underline{g(e)} = g(b)$	(4)	(2)	(4)
$\mathbf{n_2}$				
(1)	$\neg c_1$			
(2)	$d = c$			
(3)	$f(a) = a \vee \underline{g(e)} = g(d)$	[g(e)]		
(4)	$f(a) = a \vee \underline{g(e)} = g(c)$	(3)	(2)	(3)
$\mathbf{n_3}$				
(1)	$f(a) = a \vee \underline{g(e)} = g(c)$	[g(e)] join		
$\mathbf{n_4}$				
(1)	c_2			
(2)	$\underline{f(a)} \neq a$			
(3)	$\underline{g(e)} \neq g(c)$			
(4)	$f(a) = a \vee \underline{g(e)} = g(c)$	[g(e)]		
(5)	$f(a) = a$	(3)(4)		
(6)	$\perp$	(2)(5)		
$\mathbf{n_5}$				
(1)	$\perp$			

Figure 4.23: Modular VC for figure 4.21 with clause join
The main difference is at node n_3

- The total number of distinct terms (and/or literals) in S (size of representing S)
- The position of S in (the multiset extension of) $\prec$ (in order to minimize the number of available inferences)
- The total number of distinct terms and literals that need to be added at the join - the total size of `todo, done` after the join

The first two conditions are intuitive - larger clauses are more expensive to handle. The third condition is specific to superposition (and e.g. ordered resolution) and tries to get the clauses as close as possible to the normal form at the join by the term rewrite system defined at the join. The fourth condition is an incremental measure, that tries to minimize the cost of an incremental update. Note that the conditions above may not agree with each other - e.g. larger terms are not necessarily higher in the ordering.

We have already seen examples for the first measure. For the second measure (total number of distinct terms in S) consider the following join problem (assuming $\mathrm{d} \succ \mathrm{c} \succ\!| \ \mathrm{a} \succ \mathrm{f}$): (rememeber that we use $\succ\!|$ for constant ordering to denote that the left hand side is larger and of a higher maximal ordinal than the right hand side - as non-constant function symbols are less significant in our ordering definition, this means, in our case, that any term in which either of c, d occurs is larger than any term in which they do not occur).
Set of unit equalities at join : $\emptyset$
Joinee 1: $\left\{\mathrm{g}(\mathrm{f}^5(\mathrm{a})) = \mathrm{a} \vee \underline{\mathrm{d}} = \mathrm{c}\right\}, \left\{\mathrm{f}^{10}(\mathrm{a}) = \mathrm{f}^5(\mathrm{a})\right\}$
Joinee 2: $\left\{\mathrm{g}(\mathrm{f}^7(\mathrm{a})) = \mathrm{a} \vee \underline{\mathrm{d}} = \mathrm{c}\right\}, \left\{\mathrm{f}^{14}(\mathrm{a}) = \mathrm{f}^7(\mathrm{a})\right\}$
(The powers represent the number of times the function is applied)
Here the set of representatives for $\mathrm{f}^5(\mathrm{a})$ from joinee 1 is $\left\{\mathrm{f}^{5\mathrm{k}}(\mathrm{a}) \mid \mathrm{k} \in \mathbb{N}\right\}$
and similarly the set of representatives for $\mathrm{f}^7(\mathrm{a})$ from joinee 2 is
$\left\{\mathrm{f}^{7\mathrm{k}}(\mathrm{a}) \mid \mathrm{k} \in \mathbb{N}\right\}$
Two possible solutions to the join problem are:
(1) $\left\{\mathrm{g}(\mathrm{f}^{35}(\mathrm{a})) = \mathrm{a} \vee \underline{\mathrm{d}} = \mathrm{c}\right\}$ and
(2) $\left\{\neg \mathrm{c}_1 \vee \mathrm{g}(\mathrm{f}^5(\mathrm{a})) = \mathrm{a} \vee \underline{\mathrm{d}} = \mathrm{c}, \mathrm{c}_1 \vee \mathrm{g}(\mathrm{f}^7(\mathrm{a})) = \mathrm{a} \vee \underline{\mathrm{d}} = \mathrm{c}\right\}$
The first solution minimizes the number of clauses and the number of literals per clause (width), while the second solution minimizes the number of distinct terms. When we use a bounded fragment, for example, if we bounded the depth of terms, it might be the case that $\mathrm{f}^{35}(\mathrm{a})$ is too deep and then we are left with only the option (2). The opposite would be true if we restricted clause width, which might force us to choose (1).

For the third point above, consider the join problem ($\mathrm{d} \succ \mathrm{c} \succ\!| \ \mathrm{b} \succ \mathrm{a} \succ \mathrm{u}$):
Set of unit equalities at join : $\emptyset$
Joinee 1: $\left\{\mathrm{f}(\mathrm{a}) \neq \mathrm{a} \vee \underline{\mathrm{f}(\mathrm{d})} = \mathrm{a}\right\}, \{\mathrm{c} = \mathrm{a}\}$
Joinee 2: $\left\{\mathrm{f}(\mathrm{a}) \neq \mathrm{a} \vee \underline{\mathrm{f}(\mathrm{d})} = \mathrm{b}\right\}, \{\mathrm{c} = \mathrm{b}\}$
Here we have two possible solutions for the join:
(1) $\left\{\mathrm{f}(\mathrm{a}) \neq \mathrm{a} \vee \underline{\mathrm{f}(\mathrm{d})} = \mathrm{c}\right\}$ and
(2) $\left\{\neg \mathrm{c}_1 \vee \mathrm{f}(\mathrm{a}) \neq \mathrm{a} \vee \underline{\mathrm{f}(\mathrm{d})} = \mathrm{a}, \mathrm{c}_1 \vee \mathrm{f}(\mathrm{a}) \neq \mathrm{a} \vee \underline{\mathrm{f}(\mathrm{d})} = \mathrm{b}\right\}$
Here, (1) has less clauses and distinct terms, but is higher in the ordering $\succ$, and so might require the propagation of $\neg \mathrm{c}_1 \vee \mathrm{c} = \mathrm{a}, \mathrm{c}_1 \vee \mathrm{c} = \mathrm{b}$ if e.g. $\mathrm{f}(\mathrm{d}) = \mathrm{u}$

were to hold at some later node, while selecting (2) as the solution does not require the propagation of these clauses.
Algorithm sketch:
Our algorithm **first** selects a non-empty set of join representatives for each term appearing in each joinee clause, ensuring that at least one minimal term is chosen (w.r.t. $\succ$) and that each representative that is common to another term from the other joinee is also selected. For bounded fragments such as bounded term depth, it might be that the only potential representative within the depth bound is not minimal w.r.t. $\prec$ or is not common with a term of another joinee, the algorithm ensures that in this case we always have at least one representative. Besides terms common with the other joinee, the algorithm selects a term with minimal depth.
For example, for the join $\{\mathrm{f(c) = f(b) = a, C \vee P(a)}\} \sqcup \{\mathrm{f(d) = f(b), C \vee P(f(b))}\}$ we select the sets $\{\mathrm{f(b), a}\}, \{\mathrm{f(b)}\}$ respectively - a is added although it has no parallel in the other joinee because if no match is found for the entire clause, we use it as the representative as it is minimal w.r.t. depth.

The first stage of the algorithm is implemented using EC-graphs, it works by first finding the set of EC-nodes in each joinee EC-graph that represent all terms in all clauses in that joinee (sub-term closure of the ECs of all terms occurring in each clause - simply following GFA edges). It then selects all constants in the sub-term closure that occur on both sides and builds, bottom up, the set of terms common to both sides, ensuring that a minimal depth representative is always selected.

The **second stage** selects representatives for literals, based on the term representatives (that is, literal representatives include only terms selected in the first stage). At this stage we recognize literals that hold at the join or the other joinee. The algorithm assigns representative literals eagerly, and ensures that each literal has at least one representative, and furthermore, it ensures that each pair of joinee literals, one from each joinee, that have a common potential representative, will have at least one common representative literal.

For example, for the join of
$\{\mathrm{a = b = d, P(a,a)}\}$ and
$\{\mathrm{d = c, P(a,c), P(b,b)}\}$,
our algorithm selects $\mathrm{P(a,d), P(b,b)}$ as the representatives for $\mathrm{P(a,a)}$ on the left, as both have a matching literal on the other side.

At the **third stage** we select representatives for joinee clauses. We begin by assigning for each joinee clauses a set of sets of literals, which includes a set of representatives for each literal in the clause. For each joinee clause we then look for a matching joinee clause in the other joinee. A matching clause is a clause where each set of representatives of a literal of our clause also represents some literal of the other clause, and vice versa. We add all common representatives with opposite joinee clauses to the result and remove all such represented clauses from the set of pending clauses. If a clause has no matching opposite joinee clause, we select simply the clause representative that includes the minimal representative for each literal (in our case, minimal w.r.t. number of distinct terms, but this can be easily adjusted). We continue this process until there are no more pending joinee clauses.

For example, for the join of
$\{\mathrm{a = b = d, P(a,a) \vee R()}\}$ and
$\{\mathrm{d = c, P(a,c) \vee P(b,b) \vee Q()}\}$,

Where $\neg Q(), \neg R()$ holds at the join, our algorithm selects the representative clause $P(a, d) \vee P(b, b)$ as the representatives for all joined clauses, as it includes a representative for each literal of the joined clauses, except for $Q(), R()$ which are eliminated at the join. Note that $P(a, a)$ has two representatives at the joined clause.

Properties:
The algorithm is not optimal in that it does not select the minimal set of representatives (by any combination of the above-mentioned measures). The result of the algorithm depends, to some degree, on the order in which we evaluate the pending joinee clauses. However, we give a weaker guarantee that is sufficient to ensure that a clause before a branch is joined as one clause after the branch, which is important to ensure that clause propagation does not increase the number of distinct clauses in the entire program. With the unit propagation simplification, however, we cannot guarantee that there is no duplication of clauses at joins, as detailed below.

Maximal terms:
Note that in all of our examples, the representative of the maximal term in each clause was always chosen as the term itself in the representative of the clause (a request for several terms might return two clauses, where a non-maximal term in one clause is maximal in the other clause, in which case a different representative might be chosen for the term where it is non-maximal, but not where it is maximal). The first, practical reason is that if we were to choose a different representative it would be larger (by $\succ$) than the term, and hence will not be actually a valid response for the original request.
For the second reason, consider the following join problem
$(d \succ c \succ| \, b \succ a \succ u)$:
Set of unit equalities at join : $\emptyset$
Joinee 1: $\{f(a) \neq a \vee \underline{c} = a\}, \{\underline{d} = a\}$
Joinee 2: $\{f(a) \neq a \vee \underline{c} = b\}, \{\underline{d} = b\}$
Here the requested term was c and the join could be either
(1) $\{f(a) \neq a \vee \underline{d} = c, \neg c_1 \vee \underline{d} = a, c_1 \vee \underline{d} = b\}$ or
(2) $\{\neg c_1 \vee f(a) \neq a \vee \underline{c} = a, c_1 \vee f(a) \neq a \vee \underline{c} = b\}$
While the solution (1) is complete in the sense that (1) $\models$ (2), our verification algorithm does not perform derivations between propagated clauses (they are added directly to the `done` set), and hence using (1) as the result of the join is not be complete unless we add some joined clauses to the `todo` set. As we use the join algorithm also for propagation for instantiation, we allow a representative of a clause to have a different maximal term, and we allow the relativized unit equalities from a joinee (e.g. $\neg c_1 \vee \underline{d} = a$) to be added to the `todo` set.

Unit propagation and subsumption: One of the more powerful simplification derivations is unit propagation, which is used both in saturation provers and SMT provers. In order to prevent exponential explosion in the number of clauses due to unit propagation, we need to refine our join algorithm.

In figure 4.24, if the join does not account for unit propagation, and if D is a strictly maximal literal that matches a request from the end of the program, we get the following set of clauses at n_2:
$\{A_0 \vee B_0 \vee D, A_0 \vee B_1 \vee D, A_1 \vee B_0 \vee D, A_1 \vee B_1 \vee D\}$
While a join that takes into account unit propagation gives us:
$\{A_0 \vee B_0 \vee A_1 \vee B_1 \vee D\}$.

```
n_0 :
assume A_0∨A_1∨B_0∨B_1∨D
if (c0)
   n_0.0 :
   assume ¬A_0
else
   n_0.1 :
   assume ¬A_1
n_1 :
if (c1)
   n_1.0 :
   assume ¬B_0
else
   n_1.1 :
   assume ¬B_1
n_2 :
....
```

Figure 4.24: Example for exponential clause join for unit propagation with sequential joins

The relativised clauses $\{\neg c_0 \vee \neg A_0, c_0 \vee \neg A_1, ...\}$ are only propagated on a per-need basis, as D is the maximal literal.
As far as soundness and completeness are concerned, for each join of the branch condition c where the clause $C \vee B \vee \underline{A}$ holds on one joinee and the clause $D \vee B \vee \underline{A}$ holds on the other (as $A \succ B, C, D$ and B, C and D are pair-wise disjoint), we could propagate to the join either
(1) $\{\neg c \vee C \vee B \vee A, c \vee D \vee B \vee A\}$ or
(2) $\{C \vee D \vee B \vee A\}$.
The advantage of (1) is that we have to propagate at most one clause per joinee clause, while for (2) we might need up to the product of the number of clauses in the joinees - quadratic. The advantage of (2) is that, in cases such as the one in figure 4.24, where a clause is propagated through several sequential branch-join pairs, we get only one relativised version of the clause at each join. However, with (2) we can get a quadratic number of clauses at each join for clauses that originated at the joinees and not before the corresponding branch, and this causes an exponential blow-up when we have a tree of joins from nested branches - for example, consider the program in figure 4.25.
In figure 4.25, for option (2) we get at n_1:
$A_0 \vee B_0 \vee C_0 \vee D_0 \vee E$
$A_0 \vee B_0 \vee C_0 \vee D_1 \vee E$
.....
$A_1 \vee B_1 \vee C_1 \vee D_1 \vee E$
A total of 16 clauses of width 5, while with option (1) we get only one relativised version of each original clause (8 clauses of width 3) - so linear in the size of the input.

Our solution is a pragmatic compromise between (1) and (2) - we have found that the case of figure 4.24 is much more common than the one of figure 4.25, and more specifically often unit propagation happens on only one side of a branch.

```
$n_0$ :
if (c0)
   $n_{0.0}$ :
   if ($c_0$.1)
      $n_{0.0.0}$ :
      assume $A_0 \vee E$
      assume $A_1 \vee E$
   else
      $n_{0.0.1}$ :
      assume $B_0 \vee E$
      assume $B_1 \vee E$
else
   $n_{0.1}$ :
   if ($c_1$.1)
      $n_{0.1.0}$ :
      assume $C_0 \vee E$
      assume $C_1 \vee E$
   else
      $n_{0.1.1}$ :
      assume $D_0 \vee E$
      assume $D_1 \vee E$
$n_1$ :
assume .....$\vee$E
....
```

Figure 4.25: Example for exponential clause join for unit propagation with nested branches

```
n0 :
assume ¬y>0∨x≠null
assume x=null∨x.f=0
if (c0) //if y>0
   n1 :
   assume y>0
   //we derive:
   x≠null
   x.f=0
else
   n2 :
   assume ¬y>0
   we propagate:
    x=null∨x.f=0
n3 :
assume C∨x.f=0
   //Here our join is
   x=null∨x.f=0
```

Figure 4.26: Example for join after unit propagation

Hence our solution is to join two joinee clauses if one has a representative that is a superset of the literals of the other.

In figure 4.26, at the join at n_3, a representative for the clause $x.f = 0$ at n_1 is a subset of the literals of the representative $x = null \vee x.f = 0$ at n_2, hence we join these clauses. Both exponential factors in figures 4.24 and 4.25 are limited by the width (number of literals) of clauses, so that limiting the maximal width of clauses (relative to clauses that occur in the original program) is our strategy for ensuring overall complexity bounds (at the price of completeness).

The change needed in the algorithm is that, when matching clauses, rather than have each joinee clause cover (by representative literals) each literal of the other joinee clause, it is sufficient to require that the coverage holds in one direction. Hence with unit propagation we do not guarantee that a clause is not duplicated at a join.

A more complete solution might try to mark clauses propagated and simplified over a branch, so that the join has enough information to prevent the exponential blow-up in the number of clauses without relying on bounded fragments, and we leave that as future work.

Incremental considerations:
We have described joining two sets of clauses that are a response to the same request - hence have the same strictly maximal term (or maximal literal for resolution). Our algorithm is incremental, so that new clauses can be added to any CFG-node (e.g. by the heap fragment of section 8.1) after some requests were sent, and if that new clause matches a request, it is eventually propagated. When such a new clause is propagated to an already calculated join, there are several cases we need to consider:

- A new unit equality is added to a joinee, simplifying a non-maximal term in a previously joined clause
- A new unit equality simplifies the maximal term of a previously joined

clause

- A new clause is added to one joinee, which would change the result of the join - that is, it has a common representative with a clause already propagated (relativised) from the other joinee
- A new unit equality is added to the join node, or to a joinee, which makes two previously propagated, non-matching joinee clauses match.

In all of the above cases, a non-incremental join calculated after the new clauses were added, could have a result that is not a superset of the result of the join before the added clauses (e.g. if the join was $\{\neg c_1 \vee C \vee \underline{c} = b, c_1 \vee C \vee \underline{c} = a\}$ and after adding $b = a$ to joinee 1 we get $\{C \vee \underline{c} = a\}$ instead). This is problematic because it means that the result of any incremental join depends on the order of applying different fragments, and so less predictable and dependable and also harder to give complexity bounds for. We can keep enough information in order to *fix* the join by removing joined clauses that were superseded by the new information (e.g. removing $\{\neg c_1 \vee C \vee \underline{c} = b, c_1 \vee C \vee \underline{c} = a\}$ above), but we would then have to duplicate any derivations that have been performed on the joined clause, which we have found to be very inefficient, both because of duplicated effort and because a clause could have been propagated through a long path in the CFG, and has to be fixed in all nodes.

Instead, each node keeps track of the clauses that it has propagated from its direct predecessors (updated when a predecessor clause gets simplified), and filters the list of clauses to join accordingly, so that we only add representatives for clauses that have not yet been propagated in any form. As before, each node also keeps track of the requests it has responded to, and propagates any clauses added to direct predecessors that matches a previous request. Our implementation is, hence, asymmetric in terms of the order of evaluation, but the only duplication of work is in simplifying deductions - a clause that is propagated to a node from a predecessor and later simplified in that predecessor by e.g. unit propagation will cause the unit clause to be propagated to the node and the simplification applied there on the propagated version of the clause. For example, if we had initially propagated $A \vee B \vee C \vee \underline{D}$ from a predecessor to a join node n, which already had $\neg A$ and hence the propagated clause is simplified to $\neg c_1 \vee B \vee C \vee \underline{D}$, if we have later derived $\neg B$ at the predecessor, the predecessor would simplify its clause to $A \vee C \vee \underline{D}$, but n would not re-propagate the simplified version, but just propagate the new unit clause $\neg B$ and apply the simplification as well to get $\neg c_1 \vee C \vee \underline{D}$ - this is a duplication of effort, but it is still less effort than recalculating the join from scratch or fixing the join to be optimal.
In all cases, the conclusion of a simplification that changes the maximal term of its premise is always considered a new clause, and is propagated and joined accordingly, as it is a response to a different request than the original clause.

Join summary: We have found that joining clauses is crucial for performance when propagating clauses in a DAG shaped CFG. simplifying derivations do not interact well with joins, and we have not found a perfect solution for joining simplified clauses. However, in practice, we did not encounter many cases where our join algorithm failed to prevent duplication of clauses.

4.4 Term, literal and clause ordering

In this section we discuss the term ordering we use for superposition. The ordering used for superposition can have a significant effect on performance. There are several constraints when selecting an order, and we have geared our ordering towards interpolation (detailed in chapter 5). The main property of the ordering is that earlier DSA versions are larger than later DSA versions, so that superposition tends to try and eliminate early DSA versions so that derived clauses are in scope later in the program.

As described in the preliminaries, we use a variation of transfinite Knuth-Bendix ordering (tkbo) which allows us to define a separating ordering based on constants - essentially, we partition the constants into sets that we call layers, and define a total order on these layers, and our tkbo ensures that any term (literal, clause) that includes only constants up to some layer, is smaller than any term (literal, clause) that includes any constant from a higher layer.

We have already hinted that branch conditions must be in one of the bottom layers so as not to interfere with superposition.

The classes of constants that we encounter in our CFG are as follows:

- Program constants - come from the program (e.g. DSA versions of program variables) and user specification, additionally might include symbols added by some front-end such as Boogie or a verification methodology.
- Branch condition atoms.
- Skolem functions and constants introduced during Skolemization when pre-processing clauses to CNF.
- CNF atoms - these are predicates or boolean typed functions (possibly constants) introduced during the CNF conversion of general propositional formulae in order to prevent exponential explosion (see e.g. [RV01a] chapter 5 definition 6.5)
- Boolean constants - true and false. Used when we have functions that take Boolean parameters such as if-then-else (see e.g. discussion in [RSV15])
- Theory constants - for example numbers. If a theory is handled solely by axioms then is part of the program constants.
- Predicate symbols - predicates need to be ordered relative to other layers, but also relative to the equality symbol (e.g. whether, in the literal ordering, $\mathrm{P(b,a)} \succ \mathrm{b} = \mathrm{a}$ or not)

Here we only discuss the ordering between classes. The only class where we have found the internal ordering to be significant is program constants (constants that represent DSA versions of program variables). The ordering we have chosen is as follows, from lowest to highest:

1. Theory constants
2. Branch conditions
3. Boolean constants

4. CNF atoms

5. Program and Skolem constants

Theory constants (e.g. numbers) must be at the bottom of the ordering in order to be able to send a conjunction of pure theory literals to a theory solver (as in e.g. [BW13]). We have not implemented a full integer or rational theory, but as branch conditions are not terms, theory literals are the normal form of each term-EC in which they participate.

Branch conditions are low in the ordering as resolution on branch conditions automatically brings an exponential explosion in the size and number of clauses (essentially enumerating paths). We use joins as described in section 4.3 in order to reduce the path conditions that clauses carry and several of our simplification inferences (e.g. $\text{simp}_{\text{res}}, \text{simp}_{\text{res2}}$) are aimed specifically at reducing the number of path atoms each clause carries, when possible.

Boolean constants are next as they are essentially interpreted, and hence should be nearer the bottom of the ordering in order to give priority to eliminating uninterpreted symbols.

CNF atoms are at the bottom of the uninterpreted symbols. These atoms are added to the vocabulary in order to prevent an exponential explosion in the number and size of clauses. All occurrences of a CNF atom originate from the same CFG-node, as we convert clauses to CNF independently for each CFG-node. In a flat proof, we would want to give these atoms the lowest priority so that we only try to resolve them in clauses in which we have eliminated all other symbols. In our setting we still want the interpreted symbols at a lower priority.

Program and Skolem constants are essentially uninterpreted symbols that must be handled by superposition and hence they take the top priority. Like CNF atoms, all occurrences of a given Skolem constant originate in the same CFG-node as each clause is Skolemized independently.

Predicate symbols we place lower than all program and Skolem constants (needed for interpolation), but higher than equality, as predicate symbols have less potential derivations than equality because they cannot serve as a left premise for superposition.

4.5 Related work

4.5.1 Theorem proving and CFGs

For DPLL based SMT solvers, the main technique we are aware of for avoiding interference between unrelated program points (that is, clauses that occur on parallel branches), is lazy CNF conversion ([BDS02a]), together with a VC encoding that ensures that a branch side is decided before the encoding of the statements on that branch are available. This technique achieves the same separation that we do, but does not support joins directly (only through CDCL), and does not admit a natural form of incremental verification.
For saturation based solvers that suffer from inefficiency in the propositional fragment, a technique has been suggested that combines DPLL and superposition ([Vor14]). The technique, very roughly, abstracts a first order CNF formula to a propositional problem by abstracting each atom to a distinct propositional

atom (all occurrences of the exact same atom are mapped to the same propositional atom). The technique sends the abstraction of some of the clauses to the propositional problem to SAT solver - if found unsatisfiable, so is the original problem, otherwise, they use the satisfying assignment to filter the clauses that superposition can use. In theory, this technique can allow the prover to avoid most of the interference between opposing branches, but much depends on the details of which part of the problem is sent to the SAT solver (remember that the superposition solver uses CNF form, so it is not clear if a dedicated VC as works for lazy CNF can be used here). It will be interesting to compare the approach to ours and determine if join lemmas are actually derived.

4.5.2 Clause join

Clause join is basically a combination of the term join for which we have discussed related work in chapter 3, and simple propositional clause join. The problem has been viewed, in a related setting, as unification in [GT07b]. There the authors describe the use of unification under a theory in order to join (and abstract) unit clauses in the theory with equality, although most of the theories they handle are convex and they handle only unit clauses (they handle also programs with loops, so the setting is somewhat different). However, the paper suggests an algorithm for unification in the presence of linear integer arithmetic, which we believe might have a lot of potential for simplifying clause joins in our setting. E-unification (unification under a conjunction of equations) is described in [BS01]. We essentially unify clauses under a non-convex equational theory, where either the equalities on one joinee and the join holds, or the other joinee and the join. Our algorithm essentially calculates first an approximation for the join of the equational theories (by calculating the term representatives), which produces a convex theory at the join, and then performs a rough approximation of set-cover calculations to ensure that all joined clauses are covered. CDCL ([SS99]) does not directly join clauses, but instead can learn a lemma that holds at the join as follows: first an assertion is proven on a single path, and then the proof is generalized by looking at the clauses used in the proof - any part of the proof that does not depend on the choice of a branch side for a given branch can be learned as a lemma that is useful also for the proof of the path on the other side of the branch. However, CDCL does not have enough information on one path to select a representative for a join lemma that is also useful on the other branch, as it does not know which part of the congruence relation is shared between both sides of branch. We are not aware of work that performs a join that is aware of term ordering.

Chapter 5

Scoping and Interpolation

We have referred to scoping several times in the previous chapters, and given some examples that hint at the motivation. In this chapter we start by defining scoping in our context, and show how we determine the scope of each CFG-node. We continue by showing an algorithm for generating an interpolant for two sets of unit ground equality clauses - interpolants are closely related to scoping and are needed to achieve verification completeness. Next we discuss the changes needed to the algorithm to make it applicable to inteprolants for sequences and trees. We then extend the algorithm to non-unit clauses and DAG-shaped CFGs, generalizing the previous algorithms.

Motivation: The basic intuition behind using scoping in verification is that of a search for local, small proofs that are composed into a whole-program proof. Scoping exists in many programming languages which define which program variables are accessible at which program point. A Hoare annotation of a program usually does not include a program variable name at any point in the program where the variable is not in scope - that is, before its declaration or at a parallel CFG-node (opposite branch) to the declaration. For a passive program after DSA transformation, scoping can have an even stronger effect, as at most two DSA versions of each program variable can be in scope at the same time. In general, scoping reduces the proof-search-space for the prover at each CFG-node, but in some extreme cases can admit only proofs exponentially longer than the minimal non-local proof.

If we restrict the signature of the clauses that can occur at each CFG-node, we lose the completeness guarantee offered by the superposition calculus as even ground superposition is not complete for scoped proofs (local proofs). Several techniques have been developed for deriving interpolants from scoped superposition proofs, and in this chapter we show how we adapt some of these techniques to restore completeness for the ground fragment, and improve the coverage of the non-ground fragment. We also suggest a method to calculate a scope for each CFG-node that is not minimal, but sufficient to preserve completeness.

5.1 Basics

We use scoping only on constants, as opposed to all function symbols, mostly because program variables usually translate to constants. We can convert any program VC to be constant based by converting each $\mathrm{f}(\bar{\mathrm{t}})$ to $\mathrm{apply}_{\mathbf{arity}(\mathrm{f})}(\mathrm{c_f}.\bar{\mathrm{t}})$ where, for each arity n, $\mathrm{apply_n}$ is a new function symbol of arity n+1, and for each non-constant function symbol f, $\mathrm{c_f}$ is is a new constant symbol - after the transformation we can apply scoping only to constants with a similar effect to scoping on all functions on the original VC (similar to the transformation in [NO03]).

5.1.1 Notation

We start with a few definitions:
We use the names forward and backward scope to denote the direction in which the scope is calculated - forward scope is intuitively the set of initialized variables and backward scope is intuitively the set of live variables.
Scopes:
The local scope of a CFG-node is the set of constants that appear in clauses at the node:
$\mathrm{S_n^0} \triangleq \{\mathrm{c} \in \mathrm{C}_\Sigma \mid \mathrm{c} \trianglelefteq \mathbf{C}_\mathrm{n}\}$
($\mathbf{C}_\mathrm{n}$ is the initial set of clauses at the CFG-node n).

The forward scope of a CFG-node is the set of constants that appear at a CFG-node or its transitive predecessors:

$$\mathrm{S_n^F} \triangleq \bigcup_{\mathrm{p} \in \mathbf{ps}*(\mathrm{n})} \mathrm{S_p^0} \quad (= \mathrm{S_n^0} \cup \bigcup_{\mathrm{p} \in \mathbf{ps}(\mathrm{n})} \mathrm{S_p^F})$$

Symmetrically, the backward scope of a node are all constants used in transitive successors:

$$\mathrm{S_n^B} \triangleq \bigcup_{\mathrm{s} \in \mathbf{succ}*(\mathrm{n})} \mathrm{S_s^0} \quad (= \mathrm{S_n^0} \cup \bigcup_{\mathrm{s} \in \mathbf{succ}(\mathrm{n})} \mathrm{S_s^B})$$

And finally the minimal scope of a node:

$$\mathrm{S_n^M} \triangleq \mathrm{S_n^F} \cap \mathrm{S_n^B} \quad (\supseteq \mathrm{S_n^0})$$

The definitions above correspond to a program variable being initialized (forward scope) and alive (backward scope). The minimal scope is the set of program locations in which a variable is in scope in most programming languages. It is easy to see that in our verification algorithm each CFG-node uses at most the forward-scope. Our objective is a small proof that we can find quickly, and scoping is only a tool for reducing the proof search space, so that we will not always insist on the minimal scope.

Interpolation:

Definition 5.1.1. language - $\mathfrak{L}_S$
For a set of clauses S we define the language of S, $\mathfrak{L}_S$, to be the set of constants that appear in S - formally:
$\mathfrak{L}_\mathrm{S} \triangleq \{\mathrm{c} \mid \mathrm{c}() \trianglelefteq \mathrm{S}\}$.
A term, atom, literal, clause or set of clauses are in the language $\mathfrak{L}$ iff all the

constants that appear in it are in $\mathfrak{L}$, formally:
$C \in \mathfrak{L} \overset{\triangle}{\Leftrightarrow} \forall c \in \mathbf{C}_{\mathbf{\Sigma}} \cdot c() \trianglelefteq C \Rightarrow c \in \mathfrak{L}$.
We use $\mathfrak{L}$ both for the set of constants and the set of terms, atoms, literals, clauses and sets of clauses in the language, as long as there is no ambiguity.

We begin by discussing binary interpolation - given two sets of clauses A,B, s.t. $A \cup B \models \square$, our objective is to find a set of clauses I s.t.
$A \models I$, $I \cup B \models \square$ and $I \in \mathfrak{L}_A \cap \mathfrak{L}_B$.
This problem models the interaction between two consecutive CFG-nodes in order to find a refutation on a path they share. Later we discuss how this extends to the general problem of finding an interpolant for each CFG-node that is sufficient to prove all assertions on all paths that pass through that CFG-node.
Ground first order logic with equality admits interpolation (shown e.g. in [McM04]).

We name the sets of clauses for binary interpolation the top (A) and bottom (B) sets (according to the CFG direction).
We use N_0^t for the top set and N_0^b for the bottom set.
Theses sets are the initial interpolation problem, but during our interpolation and proof process these sets evolve, and we use N^t, N^b when describing the current sets of clauses during an algorithm run.
We use N for $N^t \cup N^b$ and similarly N_0 for $N_0^t \cup N_0^b$.
We use $\mathfrak{L}^t, \mathfrak{L}^b$ for the language of N_0^t, N_0^b respectively, (top,bottom language), and also:
The interface language $\mathfrak{L}^I$ is defined:
$\mathfrak{L}^I \triangleq \mathfrak{L}^t \cap \mathfrak{L}^b$
And the private languages are:
$\mathfrak{L}^{t+} \triangleq \mathfrak{L}^t \setminus \mathfrak{L}^b$
$\mathfrak{L}^{b+} \triangleq \mathfrak{L}^b \setminus \mathfrak{L}^t$

In our terminology, the binary interpolation problem is:
Given N_0^t, N_0^b find $I \in \mathfrak{L}^I$ s.t. $N_0^t \models I$ and if $N_0^t, N_0^b \models \square$ then $I \cup N_0^b \models \square$.

For interpolation in the CFG: we look at any path P in a program CFG s.t. P=Q.p.n.R (we use . for path concatenation), Q begins at the root, p,n are consecutive CFG-nodes, R ends at an assertion node and Q,R are possibly empty sub-paths. In our CFG, the assertion at the end of the path holds on the path iff $\mathbf{C}_P \models \square$ (that is, the set of all clauses on the path is inconsistent). Because ground FOL admits interpolation, if $\mathbf{C}_P$ is ground there is a ground interpolant $I \in \mathfrak{L}_{\mathbf{C}_{Q.p}} \cap \mathfrak{L}_{\mathbf{C}_{n.R}}$ s.t.
$\mathbf{C}_{Q.p} \models I$ and $\mathbf{C}_{n.R} \cup I \models \square$.
By definition we can see that
$\mathfrak{L}_{\mathbf{C}_{Q.p}} \cap \mathfrak{L}_{\mathbf{C}_{n.R}} \subseteq S_p^M \cap S_n^M$.
Moreover, we can see that:
$S_n^M = S_n^0 \cup \bigcup\limits_{p \in \mathbf{ps}(n)} (S_p^M \cap S_n^M) \cup \bigcup\limits_{s \in \mathbf{succ}(n)} (S_s^M \cap S_n^M)$
In other words, S_n^M is exactly the minimal language guaranteed to be sufficient to represent the interpolant between the prefix and suffix of *any* path from the root to an assertion through n. For this reason, we do not consider any scope that is smaller than S_n^M and we would sometimes select a larger scope, either for performance or for the non-ground case. Note that each path through the node might have a different set of possible interpolants, and there may be an

exponential number of such paths.

Theory symbols:
In some cases, we want some symbols to be global even if they don't occur in all CFG-nodes - for example, for linear integer arithmetic, we want the number constants to be in the language of all CFG-nodes.
In addition, if we have some background axioms (e.g. from a verification methodology, heap modeling, etc.), we would want all constants that occur in the axioms to be global.

```
n₁ :
   assume P(c)=T
n₂ :
   assume Q(d)=T
n₃ :
   assert P(c)=T
```

Figure 5.1: Minimal scope

```
n₁ :
   assume c=f(a,e)
   assume d=f(b,e)
n₂ :
   ...
   //S_n^F = {a,b,c,d,e}
   //S_n^M = {a,b,c,d}
   //Interpolant for S_n^F: {c = f(a,e), d = f(b,e)}
   //Interpolant for S_n^M: a ≠ b ∨ c = d
n₃ :
   assume a=b
   assert c=d
```

Figure 5.2: Minimal scope - unit ground fragment

5.2 CFG Node Scope

Intuitively, the set of program locations in which a program variable is in scope is the part of the program where it exists in memory and accessible by the program. We use at least the minimal scope at each node as it is the smallest scope that we can determine syntactically, that is guaranteed to suffice for proving the program in the ground case.

For example, consider figure 5.1. If $c \notin S_2^M$ (that is, clauses derived at n_2 cannot have occurrences of c) there is no way to prove the assertion - there is no interpolant (not even with quantifiers) that verifies the assertion which does not include the constant c at I_{n_2}.

In our definition for a fragment interpolant, we have restricted the set of clauses of the interpolant at a node to clauses in the vocabulary in the scope of the node, which we have left as a parameter and until now assumed that this included all constants in the signature. In the ground case, it is easy to see that we can use S_n^F as the scope at node n in our verification algorithm without losing completeness, as the conclusion of each inference rule only contains symbols from the premises, and as clauses are only propagated forward.

In the ground unit fragment this is not sufficient - for example, consider figure 5.2. Here, using S_n^F allows us to prove the program in the ground unit fragment, while using S_n^M requires the non-unit fragment.

Consider also the example in figure 5.3. If we restrict the scope at each node to S_n^M, we can still prove the assertion with a low depth bound - that is, without propagating

$h_3 = wr(wr(wr(h_0, x, f, 1), x, g, 2), y, f, 3)$,

```
s :
   ... //Heap axioms
   assume f ≠ g
n1 :
   assume x≠null
   assume y≠null
   assume x≠y
   x.f := 1     //h1 = wr(h0, x, f, 1)
n2 :
   x.g := 2     //h2 = wr(h1, x, g, 2)
n3 :
   y.f := 3     //h3 = wr(h2, y, f, 3)
n4 :
   assert y.g = old(y.g)
   //negated rd(h3, y, g) ≠ rd(h0, y, g)
   //Here S^M_n4 = {y, g, h0, h3}
```

Figure 5.3: Intermediate scope
We assume the standard heap axioms without extensionality

only $\mathrm{rd}(\mathrm{h}_0, \mathrm{y}, \mathrm{g}) = \mathrm{rd}(\mathrm{h}_3, \mathrm{y}, \mathrm{g})$.
This case is interesting, as it represents a typical combination of heap updates, DSA variables and post-conditions.

In light of the above, we sometimes want to find some compromise scope S s.t. $\mathrm{S}^{\mathrm{M}}_{\mathrm{n}} \subseteq \mathrm{S} \subseteq \mathrm{S}^{\mathrm{F}}_{\mathrm{n}}$ and S still allows us to find unit and short proofs. Here, $\mathrm{a} \in \mathrm{S}^{\mathrm{F}}_{\mathrm{n}} \setminus \mathrm{S}^{\mathrm{M}}_{\mathrm{n}}$, but we can represent f(a) using the constant b and $\mathrm{b} \in \mathrm{S}_{\mathrm{n}}$.

For each node n, we are interested in finding a minimal S_{n} which satisfies $\mathrm{S}^{\mathrm{M}}_{\mathrm{n}} \subseteq \mathrm{S}_{\mathrm{n}} \subseteq \mathrm{S}^{\mathrm{F}}_{\mathrm{n}}$ but still allows us to avoid resorting to non-unit clauses for completeness (without joins) in the unit fragment (as $\mathrm{S}^{\mathrm{F}}_{\mathrm{n}}$ does).

Another reason that we are interested in finding a scope where each transitive predecessor term is representable is for completeness in the non-ground case. This is discussed in chapter 7 and we only motivate it here briefly. The standard lifting argument to show completeness for superposition uses, instead of the set of ground clauses, the set of ground instances of all clauses. It has to be proven that each inference between ground instances is an instance of an inference between two clauses in the saturated set. This does not hold in our case (as discussed e.g. in [BW13]) as some ground instances use symbols that do not exist together in any node-scope.

One solution is to ensure that each term in $\mathfrak{L}^{\mathrm{t}}$can be proven equal to a term in $\mathfrak{L}^{\mathrm{b}}$. We can then replace each mixed-color ground clause by a non-mixed color ground clause that is equivalent under the interpretation (in [BW13] a multi-sorted signature is used, and then the above has to hold essentially only for terms of sorts on which N^{b} can quantify - this is related to their notion of sufficient completeness).

An interesting example of the above, using a heap theory, is that if a memory locations becomes unreachable at some point in the program, we might not be able to prove some global heap invariants at later points - for example, consider figure 5.4. We only discuss *static syntactic scoping* - scoping determined syntactically as a pre-process using only the original set of clauses at each CFG-

```
s :
   ... //Heap axioms
   //precondition
   assume x.f.g=y
   assume f≠g
   assume x.f∈P //set of locations
n₁ :
   x.f := null //h₁ = wr(h₀, x, f, null)
n₂ :
   assert ∃p∈P·p.g=y
   //negated as
   //∀p ∈ P · rd(h₁, p, g) ≠ y
   // the object pointed to by y is reachable from P
```

Figure 5.4: Scoping and reachability
x.f at the initial state is unreachable at the assertion,
which is equivalent to $\mathrm{rd}(h_0, x, f)$ having no equivalent ground term at n_2.
Without scoping we would use the mixed term $\mathrm{rd}(h_1, \mathrm{rd}(h_0, x, f), g$ to instantiate the quantifier.

node, and not derived equalities.

Algorithm: Our algorithm begins by assigning $S_n = S_n^M$ at each node. The main idea is that we can skip adding a constant c from S_n^F to S_n if it can be represented as a term over S_n using the known equalities at n - that is, for some term t over S_n we have $\mathbf{C}_n \models c = t$. We do not actually check entailment but only use positive ground unit equalities syntactically.
We traverse the CFG in topological order, at each CFG-node we begin with $S_n = S_n^M$ and add constants not representable by terms over constants in S_n until all constants are represented. When there is more than one option we make an arbitrary choice, but prefer constants whose scope does not end at n.

For a constant c whose minimal scope ends at the CFG-node n, if the unit ground clause $c = t$ is in $\mathbf{C}_n$ for some ground term t, and if all the constants in t are in scope in successors of n, then any clause $C\left[c\right]_p$ can be propagated as $C\left[t\right]_p$ and the scope of c can end at n, without losing completeness.

Scope effects on size of proof: As can be seen in figure 5.3, aggressive scoping can reduce the set of constants at the price of increasing the minimal term depth of a proof. In the example, we were still able to prove the assertion using a low term depth because the heap update axioms could deduce equalities that reduce the depth of the relevant terms. We have identified some common patterns for VCs where keeping a constant in scope might be beneficial, allowing us to find narrow and shallow proofs while maintaining a small scope per CFG-node. For each clause appearing in our VC program ($\mathbf{C}_n$), there are, in general, two possible sources:

Some clauses come from original program statements, especially assignments (heap or local variable), or an `assume` statement modeling a branch condition.

The rest of the (non-assertion) clauses come either from specifications (e.g. method pre-conditions, or assumed post-conditions after a call), or from the verification methodology (e.g. the modeling of permissions in permission based verification), these can also sometimes take the form of (DSA transformed) assignments.

We are interested in the first occurrence of a constant along a path - the earliest CFG-node in which it is mentioned. For most DSA versions of variables, we have identified the following common patterns for initialization at first occurrence:

- Assignment of a constant (independent of previous versions) expression - e.g. `x:=1`
- Assignment of an expression that includes the previous DSA version of some variables (often including itself) - e.g. `x:=y` or `x:=x+1` or for heaps `x.f:=1` ($\mathrm{h_1 = wr(h_0, x, f, 1)}$)
- The join DSA version of a variable, assigned different values on two different paths
- Unassigned variable with some property assumed - several distinct cases:
  - The initial DSA version of a variable referenced in the method precondition
  - The initial DSA version of a variable that is modified in a loop, where Boogie would model the loop using a `havoc` statement on the variable - this is usually combined with `assuming` the loop invariant which often refers to the variables modified in the loop body
  - The initial DSA version of a variable after a loop - similarly the invariant (and negated loop condition) are `assumed`
  - The initial DSA version of a variable after return from a call in modular verification - usually the post-condition is assumed which might refer to the variable and (especially for heaps) we often have a (possibly quantified) framing assumption that correlates the uninitialized version with the version before the call
  - A variable used to model non-determinism

In the first two cases (variable explicitly assigned one value), we would expect that once the said DSA version goes out of $\mathrm{S_n^M}$scope, we could replace it with the expression used in its initialization, also recursively replaced until all DSA versions are not assigned a definite value. The recursive replacement might increase the depth of the term, which would mean that, if we search for a proof with increasing depth as in our example, reasoning will be forced to be more local.

In the third case, we can add path condition qualified non-unit clauses that correlate the join DSA version with the recursively replaced versions of the value assigned at each branch, also recursively, for example, consider the code in figure 5.5. If we add n such branches in sequence, we would need 2^n clauses, each n literals wide, for the strongest post-condition, if we remove join DSA versions from the scope, and 2n clauses without removing the join DSA versions. In the above case we can propagate more goal directed clauses also without the join DSA versions, but it is not clear that this can be done in the general case. Examples such as the above motivate us not to remove join DSA versions from the scope.

For all the uninitialized cases, we would also want to keep the uninitialized DSA version of the variable in scope as in this case there is no other way to

```
s :
   ... //Heap axioms
   assume x>0 // $x_0 > 0$
if ($c_0$)
   x:=x+1 // $x_1 = x_0 + 1$
else
   x:=x+2 // $x_1 = x_0 + 2$
$n_0$ :
   // $\neg c_0 \vee x_1 = x_0 + 1$
   // $c_0 \vee x_1 = x_0 + 2$
if ($c_1$)
   x:=x+3 // $x_2 = x_1 + 3$
   // $\neg c_0 \vee x_1 = x_0 + 1$
   // $c_0 \vee x_1 = x_0 + 2$
else
   x:=x+5 // $x_2 = x_1 + 5$
   // $\neg c_0 \vee x_1 = x_0 + 1$
   // $c_0 \vee x_1 = x_0 + 2$
$n_1$ :
   // With $S_n = \{x_0, x_2\}$
   //    $\neg c_0 \vee \neg c_1 \vee x_2 = x_0 + 4$
   //    $\neg c_0 \vee c_1 \vee x_2 = x_0 + 6$
   //    $c_0 \vee \neg c_1 \vee x_2 = x_0 + 5$
   //    $c_0 \vee c_1 \vee x_2 = x_0 + 7$
   //    $x_0 > 0$
   // With $S_n = \{x_0, \mathbf{x_1}, x_2\}$
   //    $\neg c_0 \vee x_1 = x_0 + 1$
   //    $c_0 \vee x_1 = x_0 + 2$
   //    $\neg c_1 \vee x_2 = x_1 + 3$
   //    $c_1 \vee x_2 = x_1 + 4$
   //    $x_0 > 1$
$n_a$ :
   assert x>2
```

Figure 5.5: Scoping DSA join

represent the value of these variables. Intuitively, it would make sense that e.g. the return value of a function is a useful value later in the program, and so we should be able to refer to it in clauses.

Implementation: We have implemented an (optional) pre-processing step that determines a scope for each CFG-node, based on the above: Initially, for each node n, we assign $\mathrm{S_n} = \mathrm{S_n^M}$. We then proceed in topological order and mark a subset $\mathrm{K_n}$ of $\mathrm{S_n^F}$ as *key constants* - constants that are determined on some path leading to n (or n itself) not to be representable using other key constants. Each constant in $\mathrm{S_n^F}$ can be represented as a term over $\mathrm{K_n}$. In some cases we have to make an arbitrary choice to select a key constant between a set of constants (e.g. `assume a=b=c` - one of $\{a, b, c\}$ has to be selected arbitrarily, or `a=f(b),b=g(a)` where either of a or b can be selected). We add each of $\mathrm{K_n}$ to $\mathrm{S_n}$. At the end of the process we are assured that any interpolant of the program can be transformed (with the above described recursive substitutions) to an interpolant where the clauses at each node are only over $\mathrm{S_n}$.

5.3 Ground Unit Interpolation

In this section, we demonstrate the basics of our interpolation approach, and demonstrate it on the ground unit fragment (although our interpolants are non-unit ground clauses). We also discuss the inherent complexity of interpolation. We begin by describing a simple graph-based interpolation procedure for two consecutive CFG-nodes that contain only unit ground clauses, which will form the basis for understanding interpolation for general DAGs and clauses. The treatment here is closely related to the algorithm and interpolation game of [FGG+12], where the main difference is that they extract an interpolant from a proof (one congruence closure graph) while we search for a proof for two CC graphs whose conjunction is inconsistent - hence we can extend our method for sequences, trees and DAGs. We follow with a discussion of improvements to the base algorithm and the necessary modifications to support sequence- and tree-shaped CFGs.

5.3.1 The problem

The problem we consider in this section is as follows:
We are given two sets of unit clauses (ground equalities and dis-equalities), N_0^t and N_0^b.
Our objective is to generate a set of ground clauses
$I \in \mathfrak{L}^I$ s.t. $N_0^t \models I$ and that if $N_0^t \cup N_0^b \models \Box$ then $I \cup N_0^b \models \Box$.
This is equivalent, in our setting, to having a CFG-node p and its successor n, where $N_0^t = \mathbf{C}_p, N_0^b = \mathbf{C}_n$.

Notation:
We use the following presentation for all of our examples:
We use the red letters u, v, w, x, y, z for constants in $\mathfrak{L}^{t+}$- exclusive to N^t, the black letters a, b, c, d, e for constants in the interface language $\mathfrak{L}^I$, and the blue letters l, m, n, o, p for constants in $\mathfrak{L}^{b+}$.
For non-nullary functions we use f, g, h.
Whenever the above symbols are insufficient, we add indices.

5.3.2 Basic graph-based algorithm

Our algorithm is based on the EC-graphs that have been described in chapter 3. This algorithm can also be described for standard terms and congruence closure graphs, but the presentation is made easier by using EC-graphs.
We represent N_0^t as the EC-graph g^t by starting with an empty EC-graph and `assuming` all clauses in N_0^t, and similarly g^b encodes N_0^b.
We use the definition of the sources function from chapter 3, where:
$\forall s \in g^t, t \in g^b \cdot (s \in \mathbf{sources}_b^t(n) \Leftrightarrow \mathbf{terms}(s) \cap \mathbf{terms}(t) \neq \emptyset)$
Meaning that the EC-node s in the graph g^t is a source of the EC-node t in g^b iff there is a term that is represented by both EC-nodes.
We use $\mathbf{sources(n)}$ where g^b, g^t are clear from the context.
We use $[s]^t$ for the EC-node of the term s in g^t when $s \in \mathbf{terms}(g^t)$, and similarly $[t]^b$.
We use the source invariant described in chapter 3. As we have seen, the source and propagation invariants ensure that for each pair of terms represented in g^b,

$\forall s,t \in g^t \cdot ((\exists u \in \mathbf{terms}(s), v \in \mathbf{terms}(t) \cdot N_0^t \cup N_0^b \models u = v) \Rightarrow s = t)$

Figure 5.6: Propagation guarantee

$N_0^t = \{c = f(a, x), d = f(b, x)\}$
$N_0^b = \{a = b, c \neq d\}$
$\mathfrak{L}^t = \{a, b, c, d, x\}$
$\mathfrak{L}^b = \{a, b, c, d\}$

Figure 5.7: Simple unit ground interpolation example.
The only interpolation clause is $a \neq b \vee c = d$.

if they are equal under the union of equations in g^t, g^b then they are equal in g^b - shown formally in figure 5.6.

If we enforce scoping, the above guarantee does not hold - for example, consider the example in figure 5.7.

The graphs and sources function are depicted in figure 5.8.
Without scoping:
g^b `adds` the EC-nodes $\{[a], [b], [x], [c, f([a], x)], [d, f([b], x)]\}$
After `assume(a=b)` and enforcing the invariants we get:
$\{[a, b], [x], [c, d, f([a, b], x)]\}$.
However, as $x \notin \mathfrak{L}^b$ in the scoped case, g^b adds only the EC-nodes
$\{[a, b], [c], [d]\}$ in our unit algorithm, and we do not get a refutation.

Basic idea:
The idea of our algorithm is simple: the two EC-graphs exchange equalities over pairs of nodes connected by the source function, each graph performs congruence closure according to the equalities received from the other graph, until one of the graphs is inconsistent. The exchanged equalities are used to form a set of Horn clauses that is an interpolant for the two sets of clauses represented in the graphs. We use here a small extension to the EC-graphs that we have mentioned briefly before - dis-equality edges. A dis-equality edge is an edge between two GTs (EC-Graph nodes) and represents a dis-equality between the ECs represented in the two GTs. When two GTs are merged, we take the union of the dis-equality edges of both GTs being merged. An EC-graph is consistent iff it has no dis-equality graph that is a self-loop - an inconsistent EC-graph g is denoted by $g \models \Box$.
Example algorithm run:
We start with the example and then describe the algorithm.
The initial state is depicted in figure 5.8.
The only possible equality to exchange can be seen in the graph - the only node in g^b that has more than one source - $[a, b]^b$.
The algorithm run begins by g^b sending $[a]^t = [b]^t$ to g^t - which g^t `assumes` - the result is shown in figure 5.9.
Now the propagation invariant of g^b is broken, as two nodes, $[c]^b, [d]^b$ share a source $[c, d]^t$.
g^t sends the equality $[c]^b = [d]^b$ to g^b, which `assumes` the equation - the result is shown in figure 5.10.
Now $g^b \models \Box$ (as it has a dis-equality self loop on the GT $[c, d]^b$ and we are done.
The interpolant we have found is the implication $a = b \rightarrow c = d$, which we write as the CNF Horn-clause $a \neq b \vee c = d$.

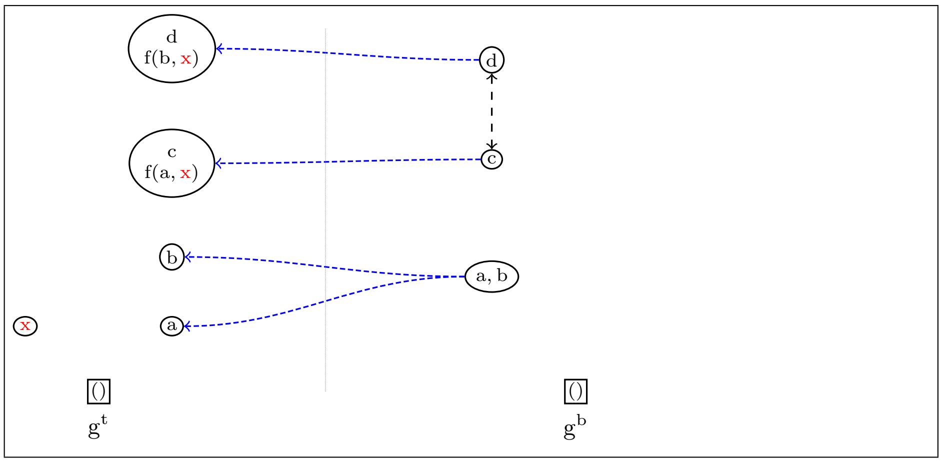


Figure 5.8: Simple unit ground interpolation example
Initial state.
Circles represent EC-nodes - GTs.
Dashed blue edges represent source edges - edges between GTs of different graphs whose EC shares a term.
Dashed black edges represent dis-equality edges.

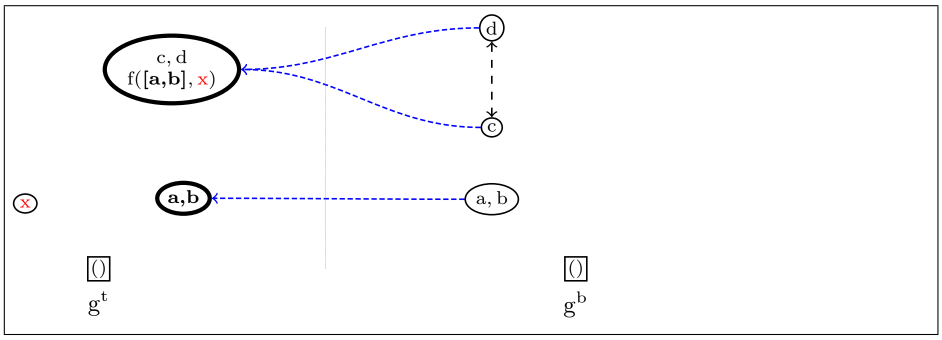


Figure 5.9: Simple unit ground interpolation example
After g^t`.assume(a=b)`
Bold text marks where g^t performed congruence closure

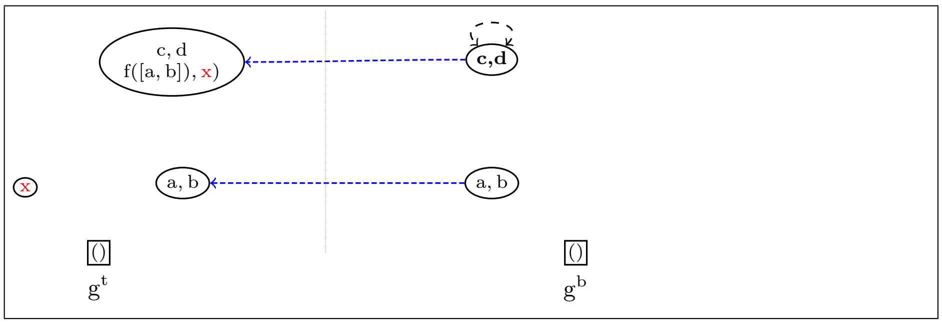


Figure 5.10: Simple unit ground interpolation example
After g^b`.assume(c=d)`
Here, $g^b \models \square$

Algorithm description:
Given the graphs g^b and g^t, we utilize the sources function that connects nodes in g^b with nodes in g^t that share at least one term. Any node in g^b or g^t that has at least one source edge is termed an interface node.
We assume that, at the initial state, the graphs g^b, g^t satisfy the source and propagation invariants - specifically, no node in g^t has more than one source-edge.

The basic step of the algorithm is composed of two stages - the first stage selects a node $t \in g^b$ where $|\mathbf{sources}(\mathbf{t})| > 1$ and then selects a pair of nodes $s_0, s_1 \in \mathbf{sources}(\mathbf{t})$ s.t. $s_0 \neq s_1$, and merges them - that is, applies `g`^{t}`.assumeEqual(`s_0`,`s_1`)`, which performs congruence closure. Before the second stage we enforce the source invariant in g^b.

The second stage is similar to the dual of the first stage - after the congruence closure in g^t and the completion of source, we can get several interface nodes in g^t that sources to the same interface node in g^b - that is, we have an EC-node $s \in g^t$ and two EC-nodes $t_0, t_1 \in g^b$ s.t. $s \in \mathbf{sources}(\mathbf{t_0}) \cap \mathbf{sources}(\mathbf{t_1})$. We now select a subset of such pairs t_0, t_1 and perform `g`^{b}`.assume(`t_0`=`t_1`)` for each.

When there are no more candidates for either the first or the second stage, we are done (and the propagation invariant holds). We claim that, once no more steps can be taken, $(g^b \models \Box \vee g^t \models \Box) \Leftrightarrow (\mathbf{eqs}(g^t) \cup \mathbf{eqs}(g^b) \models \Box)$.

Our informal argument is quite simple - if we take the resulting graph and merge each pair of nodes connected by a source-edge (one from g^t and one from g^b), and then replace each of these nodes in the GFAs in which they appear (in both graphs), we get a valid EC-graph without needing to merge any more nodes, as each node is connected with at most one source-edge and each graph is itself congruence and transitively closed. As all dis-equality edges are within a single graph, the combined graph is inconsistent iff one of its constituents is.

Extracting interpolants:
The interpolants extracted from our algorithm are always Horn clauses (as noted also in [FGG+12]). The interpolant extracted is the conjunction of one Horn-clause-EC per equation transferred from g^t to g^b, where the head is the equation and the body is the disjunction of the negation of all equalities transferred from g^b to g^t until the current stage. In order to get an interpolant set of clauses rather than clause-ECs, we select a representative term for each interface EC-node of g^t, g^n at each stage, which is an interface term. For example, we can select the minimal (by $\prec$) interface term represented by the EC-node.
We discuss the extraction of interpolants in more detail in section 5.3.5.

Source completeness:
A slightly less obvious point, which is also the reason that GFOLE admits interpolation, is that some new source-edges may have to be drawn after transferring equations, which may connect nodes that were previously not interface nodes, and hence the interpolant may include terms that are not in the initial problem - consider the example in figure 5.11.

When we transfer $a = b$ to g^b, congruence closure merges $[c, f(a, x)]^b$ and $[y, f(b, x)]^b$ into $[c, y, f([a, b], x)]^b$ and hence $[d, g([y])]^b$ is updated to $[d, g([c, y, ...])]^b$ - the term $g(c)$ did not appear on the interface originally.
Here the term $g(c)$ already appears in the original N^b (but not N^t), but we can easily modify the example to require an entirely new term - hence it is crucial that we use our EC-graph algorithm to ensure that all necessary source-edges

$\mathrm{N}_0^t = \{c = f(a, x), y = f(b, x), d = g(y)\}$
$\mathrm{N}_0^b = \{a = b, d \neq g(c)\}$

Figure 5.11: Interpolation with new terms
The interpolant is $a \neq b \vee d = g(c)$.

$\mathrm{N}_0^t = \{c = g(a, x), d = g(b, x), e = h(a, x), y = h(b, x) = f(y)\}$
$\mathrm{N}_0^b = \{a = b, m \neq f(m), e = g(c, l), m = g(d, l)\}$

Figure 5.12: Interpolation with new terms
The interpolant is $a \neq b \vee c = d, a \neq b \vee e = f(e)$.
The term $f(e)$ does not appear in the initial problem at all.

are added - show in figure 5.12. The algorithm run is show in figures 5.13-5.17.

We have not specified how the equalities communicated between the two graphs are selected. For the basic algorithm, any selection will work as long as we continue the process until no equations can be communicated in either direction. In the next sections, we discuss several improvements of the basic algorithm, including not communicating useless equations, interpolation for several g^bs and generating stronger interpolants.

5.3.3 Selecting equations to communicate

In our verification algorithm, interpolation happens in an incremental environment - that is, after we have calculated the interpolant between N^t and N^b, new (dis-)equalities can be added to N^t and N^b by other fragments and we need then to find the change in the interpolant. We are interested in understanding the space of potential interpolants, in order to define an incremental search strategy that generates optimal interpolants (in our setting optimal means smaller and usually stronger, while interpolants intended for invariants have different requirements - see e.g. the discussion in [DKPW10]).

Additionally, in some cases, the Horn clauses extracted are not the strongest possible. This happens when not all of the antecedents are needed to prove the consequent, and might depend on the order of communicating equalities.
Consider the example in figure 5.18.
Here, if we first communicate $a = b$ the interpolant we get is
$\{a \neq b \vee e_1 = e_2, a \neq b \vee c \neq d \vee e_3 = e_4\}$
While if we first communicate $c = d$ the interpolant is:
$\{c \neq d \vee e_3 = e_4, a \neq b \vee c \neq d \vee e_1 = e_2\}$
The stronger interpolant is:
$\{a \neq b \vee e_1 = e_2, c \neq d \vee e_3 = e_4\}$
In this case it seems clear that the graph N^t has two independent parts, but in some other cases it might not be easy to determine the independent parts of N^t - we discuss an improvement of the basic interpolation algorithm that produces optimal interpolants in this section. In some cases, there are several incomparable interpolants, each of which is sufficient for a refutation - for example, consider the interpolation problems in figures 5.19 and 5.20 - in both cases we have more than one interpolant and they are all of the same size.

In the next sections we discuss some modifications of the base interpolation algorithm that allow us to generate better interpolation clauses in an incremen-

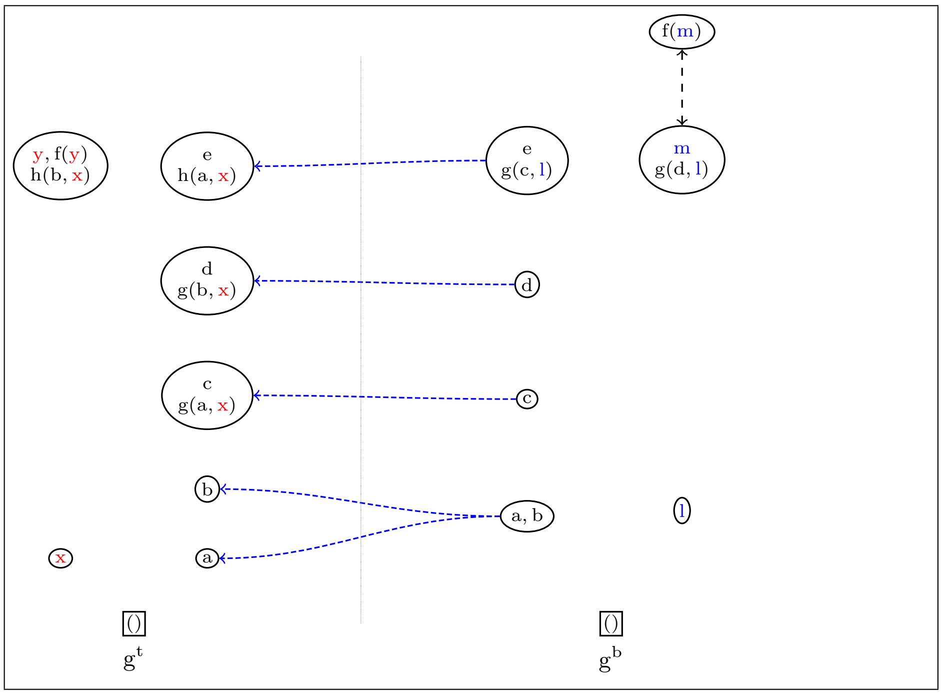


Figure 5.13: Interpolation with new terms
Initial state

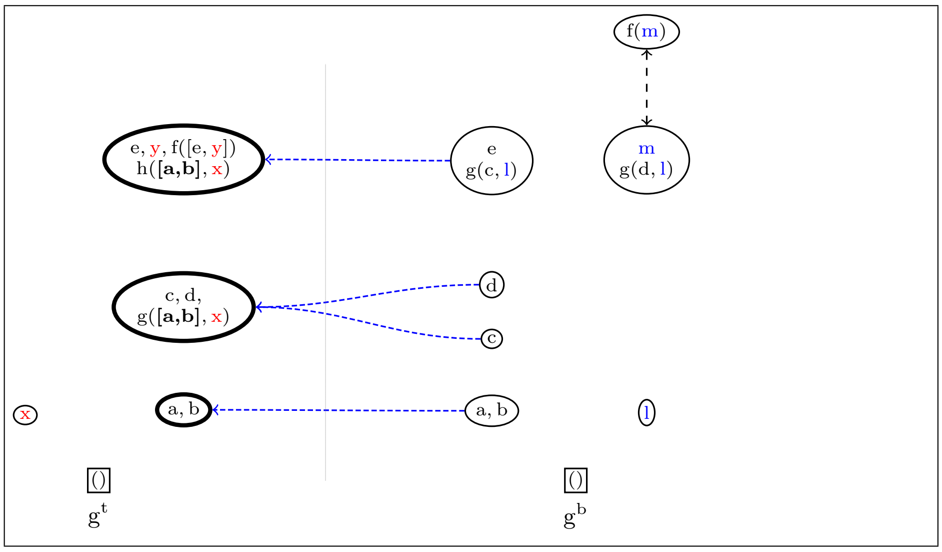


Figure 5.14: Interpolation with new terms
After g^b`.assume(a=b)`

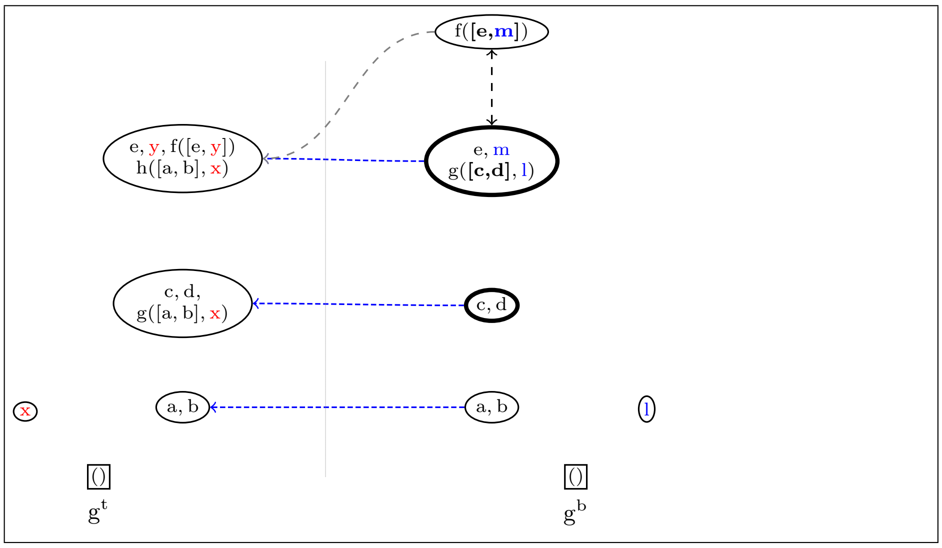


Figure 5.15: Interpolation with new terms
After g^b`.assume(c=d)`

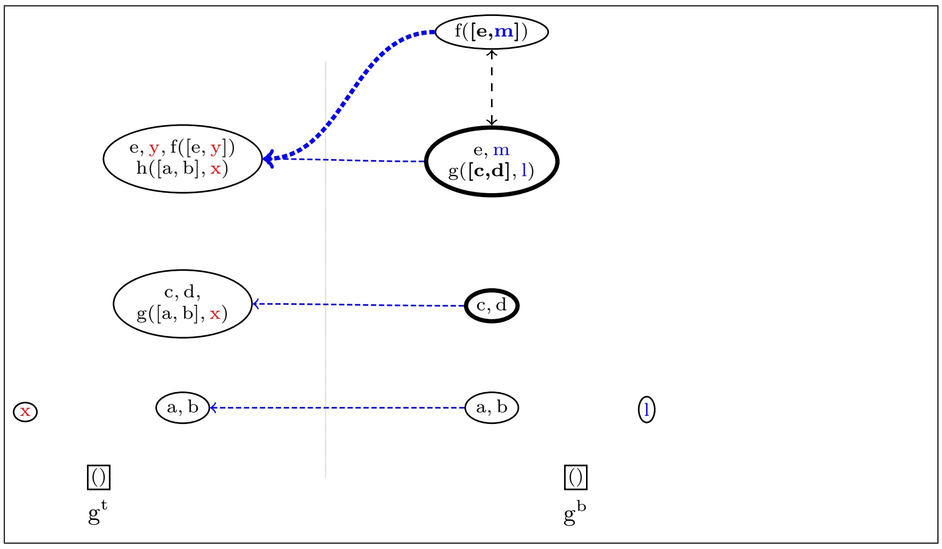


Figure 5.16: Interpolation with new terms
After g^b`.update`

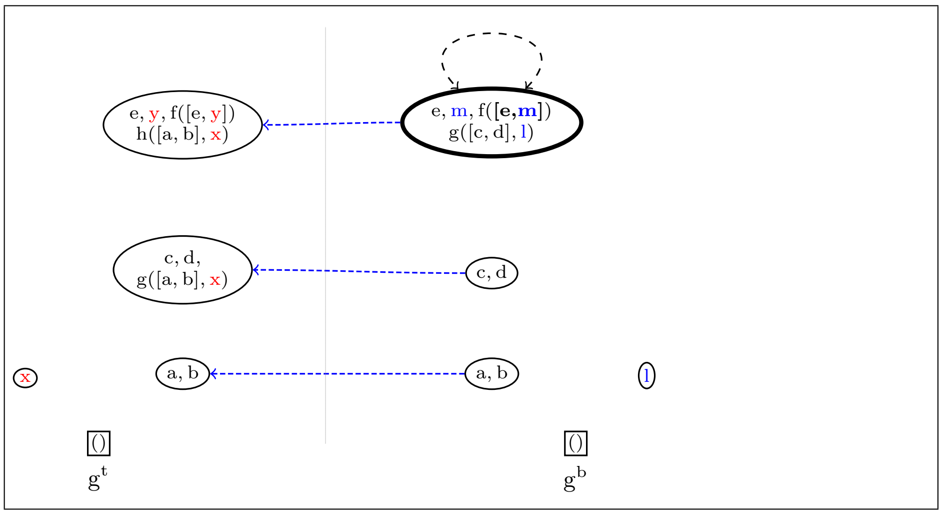


Figure 5.17: Interpolation with new terms, final state
After g^b`.assume(e=f(e))` The interpolant is $\{a \neq b \vee c = d, a \neq b \vee e = f(e)\}$
f(e) does not appear in the original problem

$N^t = \{e_1 = g(a, x), e_2 = g(b, x), e_3 = h(c, x), e_4 = h(d, x)\}$
$N^b = \{a = b, c = d, f(e_1, e_3, l) \neq f(e_2, e_4, l)\}$

Figure 5.18: Example for order dependence in extracted interpolants
The interpolant extracted by our algorithm is either
$\{a \neq b \vee e_1 = e_2, a \neq b \vee c \neq d \vee e_3 = e_4\}$ or
$\{c \neq d \vee e_3 = e_4, a \neq b \vee c \neq d \vee e_1 = e_2\}$,
depending on the order in which we communicate equalities.
The optimal interpolant is:
$\{a \neq b \vee e_1 = e_2, c \neq d \vee e_3 = e_4\}$

$N^t = \{e_1 = g(a, x) = h(c, x), e_2 = g(b, x) = h(d, x)\}$
$N^b = \{a = b, c = d, f(e_1, l) \neq f(e_2, l)\}$

Figure 5.19: Example for incomparable interpolants
The two possible interpolants are:
$\{a \neq b \vee e_1 = e_2\}$
$\{c \neq d \vee e_1 = e_2\}$
Neither is inherently preferable to the other.

$N^t = \{e_1 = g(a, b, c, x), e_2 = g(b, c, a, x)\}$
$N^b = \{a = b = c, e_1 \neq e_2\}$

Figure 5.20: Example for incomparable interpolants - transitivity
The three possible interpolants are:
$\{a \neq b \vee a \neq c \vee e_1 = e_2\}$
$\{a \neq b \vee b \neq c \vee e_1 = e_2\}$
$\{a \neq c \vee b \neq c \vee e_1 = e_2\}$

tal setting, and in a CFG (with one N^t and several N^bs).

5.3.4 Relevance calculations

In this section we describe a modification of the base interpolation algorithm presented above, which allows us to reduce the number of equations that need to be exchanged, without losing completeness. We present it here as it is used also in our algorithm for interpolation in the CFG. The basic idea is to calculate which interface equalities are *relevant* for N^t, that is, which interface equality communicated from g^b to g^t could lead to an interpolation clause. For example, if we look at figure 5.19, we can see that the interface equalities
$\{a = b, c = d\}$
can lead to interpolation clauses, while the interface equalities
$\{a = c, a = d, a = e_1, a = e_2, b = c, b = d, b = e_1, b = e_2,$
$c = e_1, c = e_2, d = e_1, d = e_2\}$
cannot lead to any interpolation clause, as they do not cause any congruence closure in g^b. Hence there is no point in communicating these equalities from g^b even if they hold in g^b.

A slightly more elaborate example is given in figure 5.21, where the only relevant interface equality is a=b, and removing any of the clauses in g^t makes all interface equalities irrelevant.

$N^t = \{$
$y_0 = f(a, x), y_1 = f(b, x),$
$z_0 = g(y_0, x), z_1 = g(y_1, x),$
$c = h(z_0, x), d = h(z_1, x)\}$
$N^b = \{a = b, f(c, w) \neq f(d, w)\}$

Figure 5.21: Example for relevance dependencies
The only possible interpolant is $a \neq b \vee c = d$.
Removing any of the clauses from N^t makes $a = b$ irrelevant for interpolation

Remember that our interpolants are sets of Horn clauses, which may have a head (positive equality) or no head (all negative equalities). The basic observation is that there are two ingredients necessary for the generation of an interpolation clause, one for the head and one for the rest of the clause:

- A goal: this is a pair of distinct terms that, if shown equal, contribute an interpolation clause- in one of the following ways:
  - A direct interface equality - for example, in figure 5.21, merging $[c]^t, [d]^t$ produces an interpolation Horn clause with the head c=d, and so the pair $\{c, d\}$ is a goal.
  - A non-interface dis-equality - for example, if we were to add $z_0 \neq z_1$ in figure 5.21 instead of the equations for c,d, merging the pair $\{[z_0]^t, [z_1]^t\}$ would produce a head-less interpolation clause - $a \neq b$
  - Equating a non-interface term with an interface term, if it can contribute to one of the above two goals - for example, in figure 5.12, merging $[y]^t, [e]^t$ produces an interpolation clause with the head $f(e) = e$

- A path from interface equalities to the goal, using congruence and transitive closure

The relevance Algorithm

We use here a set Γ of interface equalities (equalities on pairs of interface EC-nodes of N^t) to represent all the equalities that could be communicated from N^b to N^t - the algorithm is parametric in Γ, and we later use Γ as an over-approximation of the set of interface equalities between N^t and a set of N^bs - the algorithm only assumes that Γ includes at least all interface equalities of N^b.

We discuss the relevance calculations that only depends on g^t and Γ - specifically, we assume we are given g^t, the interface language $\mathfrak{L}^I$ and Γ, but not any N^b (this is needed so that we can calculate the relevance set for several successor CFG-nodes, as described in section 5.3.4).
We use now I^t for the set of (forward) interface EC-nodes of g^t - all EC-nodes in g^t for which there is a term over $\mathfrak{L}^I$.

Our algorithm calculates, given the EC-graph g^t and a set of equalities Γ not yet assumed in g^t, a subset of Γ that is guaranteed to be the only literals needed in bodies of the interpolation clauses.

The **basic idea** is somewhat similar to our join algorithm: we first proceed bottom-up and mark equivalence classes of EC-nodes in g^t that are equal under Γ, and then we proceed top-down from each goal (non-interface dis-equality or interface equality) where both sides are in the same equivalence class, and mark all pairs that can contribute to showing that the two goal EC-nodes are equal.

We sketch the algorithm in figure 5.22. The algorithm is described using EC-graphs. The input is an EC-graph `g` which represents g^t and a set of equalities Γ. The algorithm creates an EC-graph `rg` as a sequential successor of `g` using the algorithms we have shown in chapter 3 - so `rg` represents $\Gamma \cup \mathbf{eqs}(g^t)$, and maintains source-edges to `g` - edges between EC-nodes of `g` and `rg` whose ECs shares a term. We now describe the working of the algorithm.

We define first the relation $=_\Gamma$ which is the smallest congruence on g^t that satisfies Γ - formally, it is the least fixed point of the equations in figure 5.23 (represented by `rg` - specifically, for a pair of EC-node s (u,v) in `g`, $u =_\Gamma v \overset{\triangle}{\Leftrightarrow} \exists r \in \texttt{rg} \cdot u, v \in \mathbf{sources}(r)$).

In order to determine all of our goals, we need to calculate first which non-interface nodes could potentially become interface nodes under some interface equalities - we encode this using the set I^Γ which is the least set that satisfies the equations in figure 5.24.

The first rule includes all interface nodes, the second includes nodes equal in $=_\Gamma$ to potential interface nodes, and the third includes GFAs for which the tuple is a potential interface node.

In order to define the top-down part we first define the set of goals G, which are pairs of EC-nodes - shown in figure 5.25.
Goal pairs are pairs of EC-nodes of g^t that are equivalent in $=_\Gamma$and either both are potential interface nodes or they share a dis-equality edge. Our formulation here is an over-approximation, for example, in:
$N^t = \{c = f(a, x), y = f(b, x)\}$
$\{c, y\} \in G$ although no clause can be generated, while adding either
$\{d = g(a, x), y = g(b, x)\}$ or

```
method relevance(g : ECGraph, $\Gamma$ : Set[Equality])
      : Set[Pair[GT]]
   var rg := new SequentialECGraph(g)
   //rg is a sequential successor of g, maintaining
   //the propagation and source invariants

   //assume all equations from $\Gamma$ in rg
   foreach (eq $\in$ $\Gamma$)
      rg.assume(eq)

   //Add all interface nodes from g to rg
   var irgts := new Set[GT] //interface nodes in rg
   foreach (gt $\in$ $\mathrm{I}^t$)
      irgt.add(rg.makeTerm(rep(gt)))

   //Calculate $\mathrm{G}$ - we calculate the inverse source of $\mathrm{G}$
   var rG := new Set[Pair[GT]] //unordered pairs
   foreach (rgt in irgts)
      foreach (u,v $\in$ sources(rgt))
         if (u$\neq$v)
            rG.add((u,v))
   foreach ((u,v) $\in$ diseqs(g))
      ru := rg.makeTerm(rep(u))
      rv := rg.makeTerm(rep(v))
      if (ru==rv) //dis-equality refuted under $\Gamma$
         rG.add((u,v))

   return relevance(g,rg,rG)

method relevance(g,rg : ECGraph, rG : Set[Pair[GT]])
   //Calculate $\mathrm{R}$
   var $\mathrm{R}$ := new Set[Pair[GT]] //unordered pairs
   var todo := new Queue[Pair[GT]](rG)
   while (!todo.isEmpty) //Up to a quadratic number of pairs
      var (u,v) := todo.dequeue
      //Inverse congruence closure
      foreach ($\mathrm{f}(\overline{\mathrm{s}})$ $\in$ u)
         foreach ($\mathrm{f}(\overline{\mathrm{t}})$ $\in$ v)
            foreach (i $\in$ 0..arity(f))
               if ($s_i$ $\neq$ $t_i$)
                  if ($\mathrm{R}$.add(($s_i$,$t_i$))) todo.enqueue(($s_i$,$t_i$))

      //Inverse transitive closure
      var m := rg.sources$^{-1}$(u) //also rg.= sources$^{-1}$(v)
      foreach (w $\in$ sources(m)$\setminus\{$u,v$\}$)
         if (u$\notin$I or v$\notin$I or w$\notin$I)
            if ($\mathrm{R}$.add((u,w))) todo.enqueue((u,w))
            if ($\mathrm{R}$.add((w,v))) todo.enqueue((w,v))

      //Return only interface relevant pairs
      return $\{(u,v) \in \mathrm{R} \mid u,v \in \mathrm{I}^t\}$
```

Figure 5.22: Relevance algorithm

1. $\forall s = t \in \Gamma \cdot s =_\Gamma t$
2. $\forall u, v, w \in g^t \cdot ((u =_\Gamma v \wedge v =_\Gamma w) \Rightarrow u =_\Gamma w)$
3. $\forall \bar{u}, \bar{v} \in g^t, f(\bar{s}), f(\bar{t}) \in \mathbf{gfas}(g^t) \cdot (\bar{u} =_\Gamma \bar{v} \Rightarrow [f(\bar{s})]^t =_\Gamma [f(\bar{t})]^t)$

Figure 5.23: Definition of $=_\Gamma$
These equations are the standard transitive congruence closure of Γ, except that we only allow a congruence closure instance if at least one side of the conclusion is in g^t

1. $I \subseteq I^\Gamma$
2. $\forall u \in I^\Gamma, v \in g^t \cdot u =_\Gamma v \Rightarrow v \in I^\Gamma$
3. $\forall \bar{s} \in I^\Gamma, f(\bar{s}) \in \mathbf{gfas}(g^t) \cdot [f(\bar{s})]^t \in I^\Gamma$

Figure 5.24: Definition of I^Γ

$\{y \neq f(y)\}$
makes this pair a relevant goal. We leave refinement of the goal set calculations as future work, the rest of the algorithm is parametric in G. We only note here that when we interpolate in a CFG (either sequence, tree or DAG) we must add to the set of goals also pairs of EC-nodes that are on the interface with *predecessor* CFG-nodes, in order to ensure complete propagation.
The top-down part is defined using the symmetric relation R (relevant pairs) between EC-nodes of g^t, which is the least fixed point of the equations in figure 5.26.

Examples:
(as we discuss only g^t we use $[.]$ instead of $[.]^t$ for EC-nodes for simplicity)
For the example in figure 5.11 :
$g^t = \{[x], [y, f(b, x)], [a], [b], [c, f(a, x)], [d, g(y)]\}$
Which we shorten to:
$g^t = \{[x], [y], [a], [b], [c], [d]\}$
$I^t = \{[a], [b], [c], [d]\}$
We show the example for $\Gamma = \{a = b = c = d\}$
The graph `rg` represents:
$g^t / =_\Gamma \; = \; \{\{[a], [b], [c], [d], [y]\}, \{[x]\}\}$
(y is merged with c by congruence closure).
$I^\Gamma = \{[a], [b], [c], [d], [y]\}$
(y added to the interface as it was merged with the interface node for c).
`rg` has one interface node $[a]_{\mathtt{rg}}$.
The set of goal pairs are all pairs of source of merged interface nodes - there are no dis-equalities.
$G = I^\Gamma \times I^\Gamma \setminus id$
In the loop for calculating R, all of the inverse transitive closure are already in

$G \triangleq \{\{u, v\} \mid u \neq v \wedge u =_\Gamma v \wedge (u, v \in I^\Gamma \vee g^t \models u \neq v)\}$

Figure 5.25: Definition of G - goal pairs

1. $\forall \{u,v\} \in G \cup R, f(\bar{s}) \in u, f(\bar{t}) \in v \cdot$
   $((u \neq v \wedge \bar{s} =_\Gamma \bar{t}) \Rightarrow \{\bar{s},\bar{t}\} \in R)$
2. $\forall \{u,v\} \in G \cup R, w \in g^t \cdot$
   $((u \neq w \wedge w \neq v \wedge (u \notin I \vee v \notin I \vee w \notin I) \wedge$
   $u =_\Gamma w \wedge w =_\Gamma v) \Rightarrow$
   $\{u,w\}, \{w,v\} \in R)$

Figure 5.26: Definition of R - relevant pairs
Where:
$\{\bar{s},\bar{t}\} \in R \overset{\triangle}{\Leftrightarrow} \forall i \cdot (s_i \not\equiv t_i \Rightarrow \{s_i,t_i\} \in R)$
The first rule is inverse congruence closure and the second inverse transitive closure, bounded by $=_\Gamma$.
The relevant equalities are $u = v$ s.t. $\{u,v\} \in R \cap (I^t)^2$.

G hence the only added pair is $\{[a],[b]\}$ using the inverse congruence closure rule on the GFAs $f(a,x), f(b,x)$ $R = \{\{[a],[b]\}\}$ (obtained using rule 1 from the goal $\{c,y\}$)
Relevant equalities:
$\{[a] = [b]\}$

For the example in figure 5.19:
$g^t = \{[x],[a],[b],[c],[d],[e_1],[e_2]\}$
$I^t = \{[a],[b],[c],[d],[e_1],[e_2]\}$
$\Gamma = (I^t)^2$
$g^t / =_\Gamma \;=\; \{I^t, \{[x]\}\}$
$I^\Gamma = I^t$
$G = I^\Gamma \times I^\Gamma \setminus id$
As in the last example, there are no applications of the inverse transitive rule, only the two applications of the inverse congruence closure rule.
$R = \{\{[a],[b]\}, \{[c],[d]\}\}$ (rule 1 from the goal $\{e_1,e_2\}$)
Relevant equalities:
$\{[a] = [b], [c] = [d]\}$

Finally, for the example in figure 5.21:
$g^t = \{[x],[y_0],[y_1],[z_0],[z_1],[a],[b],[c],[d]\}$
$I^t = \{[a],[b],[c],[d]\}$
$\Gamma = (I^t)^2$
$g^t / =_\Gamma \;=\; \{I^t, \{[y_0],[y_1]\}, \{[z_0],[z_1]\}\}$
$I^\Gamma = I^t$
$G = I^\Gamma \times I^\Gamma \setminus id$
Here the last loop (using `todo`) finds only one additional pair - $([y_0],[y_1])$ - using the inverse of congruence closure. For this pair we apply again the inverse of conruence closure to get $([z_0],[z_1])$ and again to get the pair $([a],[b])$.
$R = \{([z_0],[z_1]), ([y_0],[y_1]), ([a],[b])\}$
Relevant equalities:
$\{[a] = [b]\}$

Having calculated the relevant equalities, we only communicate these from g^b to g^t, and when there are no relevant equalities left to communicate from g^b to g^t and no equalities at all to communicate from g^t to g^b, we are done.

Incremental updates:
The obvious representation for $=_\Gamma$ is a partition (set of disjoint subsets) of g^t, and for I^Γ simply a set. G can also be represented as a partition of a subset of g^t. R can be represented as a set of pairs, although this set can be quite large in extreme cases - a bounded fragment (in this case, bounding the depth of derivation of $=_\Gamma$) can mitigate this problem at the price of completeness.

If we enhance our basic algorithm with relevance calculations, we need to update the set of relevant equalities after each `assume` in g^t, and after updates to Γ. Remember that the interpolation algorithm works more efficiently the less equations there are in Γ.

Updating after the addition of equations to Γ is easy - each of $=_\Gamma$,I^Γ,G and R can only grow.

Removing equations from Γ (e.g. remove $s = t$ after we have proven the assertion $s \neq t$) is more complicated, and we might want to allow Γ only to grow. When we use Γ as the approximation of a candidate model for successor nodes in the CFG as in section 5.4, deleting clauses (e.g. by subsumption) may allow us to weaken Γ if we can remove equations from Γ.
If we merge two EC-nodes $u, v \in g^t$ because of an equality derived in some other fragment, we should remove the pair $\{u, v\}$ from R and G, however, propagating this change requires some more book-keeping.

Goal Relevance

For binary interpolation, we can use the same relevance calculations at N^b as follows:
The goals for N^t are dis-equalities and relevant equalities in N^b.
The goals for N^b are dis-equalities and relevant equalities in N^t.
As we can see we might need a few rounds of communications between N^t and N^b in order to establish the exact set of goals and relevant equalities.
The enhanced binary interpolation algorithm with relevance alternates between communicating relevant equalities and actual equations.

5.3.5 Scoping in the CFG

For **binary interpolation**, the algorithm given above always generates an interpolant when there is one, and can operate incrementally. The exact interpolant extracted can be tuned (e.g. congruence closure on interface terms can be performed by N^t or by N^b).
For **sequence interpolants** (e.g. an unrolled loop or scoped verification of straight line code), the above algorithm will extend easily.

Each CFG-node n maintains an EC-graph g_n, which is updated as in the binary case with both its predecessor and successor. We can communicate equalities in any order - that is, at each stage select one CFG-node and perform one stage of the above algorithm with its predecessor.

Once all consecutive pairs of CFG-nodes have been saturated - meaning that no interface node has more than one source-edge, we are done. If source completeness is enforced we claim that the algorithm is complete for verification and the generation of sequence interpolants.

Source completeness in the CFG means that, for any pair of CFG-nodes n_1, n_2, on the path P, for any pair of EC-nodes $s \in g_1, t \in g_2$, if there are terms

```
n₀:
c := f(a₀,x)
d := f(b₀,x)

n₁:
//Here x is out of scope
a₁:=a₀
b₁:=b₀
if (a₁==b₁)
   //Here a₀,b₀ are out of scope
   n₂:
   assume a₁==b₁
   assert c==d
else
   n₃:
   assume a₁!=b₁
   assert c==d //does not hold
```

Figure 5.27: Example for interpolation in the CFG

$u \in s, v \in t$ s.t. $\mathbf{eqs}(P) \models u = v$ then $s \in \mathbf{sources}_{\mathbf{2}}^{\mathbf{1}}(t)$ - meaning the two EC-nodes are connected by a source chain. We have seen that without joins we can achieve source completeness easily.

The argument for completeness is as for the binary case (we give a proof for a more general case later) - we can merge the EC-graphs of all CFG-nodes by merging each pair of EC-nodes connected by a source edge. Source completeness and the fact that each EC-node has at most one source-edge in each direction (termination condition for the binary algorithm), ensure that the resultant merged EC-graph is congruence and transitive closed - hence it is a model (as an equivalence relations) for the clauses on the path, unless one of the EC-graphs is in conflict.

For **tree interpolants** (an interpolant for a tree-shaped CFG - different from [BGKK13] which is discussed later), it is no longer sound to `assume` at the EC-graph of a CFG-node an equation from a successor, as it may not hold in another successor. Here we are essentially calculating a separate sequence interpolant for each path in the CFG, where each CFG-node calculates the interpolant between its prefix (path from the root) and, separately, *each* suffix (path) to an assertion. The simplest way to use our algorithm for sequence interpolants is to maintain, at each CFG-node, a separate EC-graph per assertion reachable from it. We can then communicate equations only between EC-graphs of the same assertion, and the interpolant at each EC-node is the union (conjunction) of the interpolation clauses for all assertions. While this is sound and complete, this is highly inefficient, as we will have a quadratic number of EC-graphs, and we are not sharing the common work between different paths. This solution also does not extend well to the DAG case, where the number of prefixes and suffixes can be exponential.

An example: Consider the example in figure 5.27. The interpolation clause from n_0 is $a \neq b \vee c = d$. We cannot simply `assume(a=b)` at g_0 because it is not sound - it will allow us to prove the assertion at n_3 which does not hold.

Our solution is as follows: we maintain two EC-graphs per CFG-node, the first is $\mathrm{g_n}$ as in chapter 3 which is *sound* in the sense that any equality that holds in $\mathrm{g_n}$ holds at the CFG-node. The second EC-graph, $\mathrm{g_n^i}$, includes all the equations implied by *all* suffixes, in this sense it is *complete* (any equation on in-scope terms that holds on any path through n holds in $\mathrm{g_n^i}$). The graph $\mathrm{g_n^i}$ is an over-approximation of the set of EC-graphs we would maintain at n in the naive implementation.

We use these two graphs as follows:
$\mathrm{g_n}$ is maintained as in the unit-ground algorithm of chapter 3.
$\mathrm{g_n^i}$ maintains source-edges to $\mathrm{g_n}$ and, independently, to $\mathrm{g_s^i}$ for each direct successor s of n. $\mathrm{g_n^i}$ enforces the source and propagation invariant with $\mathrm{g_n}$ so that all equations that hold in $\mathrm{g_n}$ hold also in $\mathrm{g_n^i}$, and all terms in $\mathrm{g_n}$ are `added` to $\mathrm{g_n^i}$.

Equations are communicated backwards between $\mathrm{g_n^i}$ and $\mathrm{g_p^i}$ as in the sequence case, except that no forward propagation of equations is performed. Instead of the forward propagation of equalities, we extract the interpolation clauses from $\mathrm{g_n^i}$ as described above, represent them as clause-ECs over $\mathrm{g_n}$, and propagate them as described in section 5.4, with a slight modification:
The idea is that we can rely on the completeness of superposition with selection (e.g. [BG91]) - essentially, whenever there is a negative literal (disequality) in a clause, we can *select* a negative literal to be the only literal that participates in derivations, even if there are larger (positive or negative) literals. As our interpolation clauses are Horn-clauses (and as a propagation in a tree-shaped CFG does not use branch conditions), each non-unit clause has at least one negative literal. Unit clauses are propagated using EC-graphs as in the unit-ground algorithm. We propagate non-unit clauses based on the maximal term of the selected negative literal. In our example, $\mathrm{g_0^i}$ encodes the equalities $\{\mathrm{a_0 = b_0, f(a_0, x) = c, f(b_0, x) = d}\}$ while $\mathrm{g_0}$ encodes only $\{\mathrm{f(a_0, x) = c, f(b_0, x) = d}\}$ - the added equality $\mathrm{a_0 = b_0}$ is in $\mathrm{g_0^i}$ although it does not hold in all transitive successor assertions.
The only generated interpolation clause is $\mathrm{a \neq \underline{b} \lor c = d}$ - propagated only for a request for b (left-hand-side) - which only comes from $\mathrm{n_2}$.
When propagated using our EC-graphs, it arrives at $\mathrm{n_2}$ as $\mathrm{[a_1] \neq [a_1] \lor [c] = [d]}$
Inspecting the superposition calculus, we can see that all superposition derivations will have a unit left-hand-side and hence they are performed automatically when a clause is propagated, as the clauses are represented as clause-ECs over EC-nodes from $\mathrm{g_n}$. Equality factoring is not relevant as no clause has more than one positive literal, and hence we are left with equality resolution. For our example the above is sufficient to prove the assertion. However, we do not cover all cases of equality resolution - we have to perform a new step - we perform equality resolution on the selected negative literal of a clause *modulo* $\mathrm{g_n^i}$, when that literal is not an interface literal with the successor. This means that a clause $\mathrm{C \lor \underline{s \neq t}}$ ($\mathrm{s \neq t}$ selected) at a node n where $\mathrm{s, t \notin \mathfrak{L}^I}$, is represented as $\mathrm{[C]_n \lor \underline{[s]_n \neq [t]_n}}$, we then check if $\mathrm{[s]_n^i = [t]_n^i}$ (that is, $\mathrm{g_n^i \models s = t}$). If this is the case, we extract a set of *justifications* for the equality - each possible body D for an interpolation clause with the head $\mathrm{s = t}$, and for each such D we add the clause $\mathrm{[C]_n \lor [D]_n}$ to the set of clause-ECs at n.
For example, consider the example in figure 5.28. Here we see two possible justification clauses, and hence two interpolation clauses.

We also perform the standard equality resolution as a simplification - eagerly

$N^t = \{d = f(a, x), e = f(b, x), f(c, x)\}$
$N^b_0 = \{a = b, d \neq e\}$ $N^b_1 = \{a = c, g(d, m) \neq g(d, m)\}$

Figure 5.28: Example for justification clauses
The justifications for $d = e$ are:
$\{a = b, a = c\}$
The interpolation clauses are:
$a \neq b \vee d = e$ $a \neq c \vee d = e$

```
assume e1 = g(a, x)
assume e2 = g(b, x)
assume e3 = h(c1, x) = h(c2, x)
assume e4 = h(d1, x) = h(d2, x)
....

if (*)
   assume c1 = d1
else
   assume c2 = d2
assume a = b
assert f(e1, e3, l) = f(e2, e4, l)
```

Figure 5.29: Example for CFG approximations
Here, for the root node, we can use $\{c_1 = d_1, c_2 = d_2, a = b\}$ as the approximation for the equalities on all paths, rather than apply the algorithm twice, once for $\{c_1 = d_1, a = b\}$ and once for $\{c_2 = d_2, a = b\}$

replacing $[C]_n \vee [s]_n \neq [s]_n$ with $[C]_n$.
As in the non-unit case, if we derive a unit clause at n it is `assumed` in the graph g_n (and propagated to successors and to g^i_n as before).

We claim that this is a complete algorithm for the ground unit fragment on a tree-shaped CFG - the reason is that we over-approximate the derivations of the superposition calculus with selection on each path in the CFG, where unit derivations are handled by the EC-graph, As explained above, superposition is also handled by propagation and the EC-graph, and equality resolution is handled by over-approximating the equational theory on the path, and adding the relevant assumptions to the conclusion. We do not give a proof here as we prove a more general result in section 5.4. In this section we concentrate on the approximation encoded in g^i_n and the extraction of assumptions (Horn clause bodies) from g^i_n.

For **DAG interpolants**, we use a similar algorithm as for trees, except that now we have branch-conditions in clauses and hence cannot rely solely on unit superposition - we describe the solution in section 5.4.

An example:
We show an example of our abstraction of the equality theory of suffixes - in figures 5.29 and 5.30. We can see that our over-approximation can sometimes produces undesirable interpolation clauses, which would not be produced by per-path interpolation - this is the price of our approximation, but the advantage is that we use only one additional EC-graph per CFG-node.

```
assume $e_1 = g(a, x)$
assume $e_2 = g(b, x)$
assume $e_3 = h(c_1, c_2, x)$
assume $e_4 = h(d_1, d_2, x)$
...
if (*)
   assume $c_1 = d_1$
else
   assume $c_2 = d_2$
assume $a = b$
assert $f(e_1, e_3, l) = f(e_2, e_4, l)$
```

Figure 5.30: Example for CFG over-approximation
Here, for the root node, if we use $\{c_1 = d_1, c_2 = d_2, a = b\}$
we produce the interpolation clause $c_1 \neq c_1 \vee c_2 \neq d_2 \vee e_3 = e_4$,
which is useless as the body holds on any path

Extracting justifications for equalities

We show here a simple algorithm for calculating the set of justifications for each equality on a pair of EC-nodes from g_n that are merged in g_n^i. Extracting a proof from congruence closure graphs has been done efficiently in [NO05], and also implemented for binary interpolation in [FGG$^+$12] (section 4.5, definition 4.2).
Our requirement is slightly different - we need to get the set of *all* possible premises for a proof that are in Γ.
For example, for the set of equations
$c = f(a_1, x) = f(b_1, x), d = f(a_2, x) = f(b_2, x)$ where
$\Gamma = \{a_1 = b_1 = a_2 = b_2 = c = d\}$,
for the pair $\{c, d\}$, we must produce the sets
$\{a_1 \neq a_2\}, \{a_1 \neq b_2\}, \{b_1 \neq a_2\}, \{b_1 \neq b_2\}$.
But for $\Gamma = \{a_1 = a_2, c = d\}$,
for the pair $\{c, d\}$, we only need the set
$\{a_1 \neq a_2\}$.

Our algorithm accepts as input the graphs g_n and g_n^i, the set of interface equalities Γ, and the pair of EC-nodes $s, t \in g_n$ s.t.
$\mathbf{sources}_{g_n^i}^{g_n\,-1}(s) = \mathbf{sources}_{g_n^i}^{g_n\,-1}(t)$.
We know that $\mathbf{eqs}(g_n) \cup \Gamma \models \mathbf{eqs}(g_n^i) \models \mathbf{eqs}(g_n)$ and hence the algorithm must return, at least, the set
$\{D \subseteq \Gamma \mid D \cup \mathbf{eqs}(g_n) \models s = t \wedge \forall D' \subset D \cdot D \cup \mathbf{eqs}(g_n) \not\models s = t\}$.
The soundness of each set D returned is expressed by
$D \cup \mathbf{eqs}(g_n) \models s = t$ and $D \subseteq (\mathbf{eqs}(g_n^i) \setminus \mathbf{eqs}(g_n))$.
Our algorithm simply collects all possible justifications recursively until saturation (needed as g_n^i might have cycles).
As the notation for the inverse source function is somewhat cumbersome, we use the notation $g_n^i \models u = v$ when $u, v \in g_n$ to denote that both EC-nodes u,v map to the same EC-node in g_n^i, we denote by u' the EC-node in g_n^i to which u maps and so $u' = v'$ is equivalent to $g_n^i \models u = v$.

We define the set P(s,t) of interface premises for a pair of EC-nodes $s, t \in g_n$ s.t. $s' = t'$ using a set of equations, where P is the least fixed point of the equations. The equations are described in figure 5.32. In order to calculate the set, any algorithm that constructs a least fixed point of equations works. We show a simple algorithm in figure 5.31. The algorithm implements the fixed point calculation as defined in figure 5.32, from the bottom (equalities in Γ) up. Each set of pairs S represents the clause $\bigvee \{s \neq t \mid \{s,t\} \in S\}$. It is sufficient to include justifications from $P(s,t)$ that are not subsumed in $P(s,t)$ - this can be added in the construction algorithm or as a post-step. Also, note that we cache the values of P we have calculated, so the algorithm is incremental.

Complexity: In section A.1.2 of the appendix we give an alternative algorithm for approximating Γ. We use the alternative algorithm to calculate the space complexity bounds for interpolants - we show that the set calculated by our algorithm in 5.32 can be of exponential size. The alternative algorithm is more complicated and we do not consider it practical in our setting, but it can be used to generate bounded interpolants - where we bound the maximal number of literals of an interpolant (losing completeness), and hence ensure a complexity bound for an incomplete algorithm.

Goal relevance: Extending the idea of goal relevance to sequence interpolants seems straightforward - the communication of relevant equalities proceeds as in the binary case, but now we have to communicate also transitive goals - consider the example in figure 5.33.
Here, $\{[c_3]_3, [d_3]_3\}$ is a goal at N_3, which is communicated to N_2 as
$\{[c_2, c_3]_2, [d_2, d_3]_2\}$ (through source edges), which must be communicated to N_1 as $\{[c_2, f(a_1, x_1)]_1, [d_2, f(b_2, x_1)]_1\}$ in order for N_1 to generate the interpolation clause-EC $[a_2 \neq b_2 \vee c_2 = d_2]_1$.
Note that even in the case of a non-scoped ground superposition proof, we can view dis-equality literals as goals, as the only derivation where a literal of the larger premise is not in the conclusion is equality resolution (equality factoring replaces a literal rather than removing it). We can view the *life cycle* of each dis-equality literal in the premises where, at each proof-tree node whose subtree includes the premise, the dis-equality is either copied to the conclusion (if it is not maximal), rewritten using some maximal equality in another clause (negative superposition) or eliminated by equality resolution (dis-equalities that originate from equality factoring start their life cycle at the conclusion of that inference).
The important property is that, for each negative superposition inference in the proof-tree, the proof sub-tree that derives the rewriting clause (the left premise of negative superposition) is independent of the proof-tree deriving the clause with the maximal dis-equality.

Hence, if we use an approximation similar to our g_n^i which includes all maximal equalities in a set of ground clauses, we can restrict negative superposition to only clauses (right premises) where the maximal dis-equality is false under g_n^i. This can be extended further (assuming superposition with selection) by only allowing a clause to participate in a derivation if *all* of its negative literals are false under g_n^i (some care needs to be taken regarding equality factoring).

While the above restriction might not be very effective for general superposition, in our case, we can use the idea to block clause propagation, not just clause derivation - if the maximal dis-equality in a clause is not false under g_n^i, there is no need to propagate it, as it cannot participate in a refutation in any successor node.

Note that in the non-ground case, the life-cycle of a dis-equality includes

```
var m : Map[Pair[GT],Set[Set[Pair[GT]]]

method P(s0,t0:GT) : Set[Clause]
   //All pairs are unordered
   var todo := new Queue[Pair[GT]]

   var RP := relevance($g_n$,$g_n^i$,{s0,t0})
   todo.enqueue(RP)

   while (!todo.isEmpty)
      ({s,t}) := todo.dequeue
      var r : Set[Set[Pair[GT]]] := $\emptyset$

      if (s=t)
         r.add($\emptyset$)
      else if s=t$\in\Gamma$
         r.add({s,t})
      else
         x := $\mathbf{sources}^{-1}(s)$ //also $= \mathbf{sources}^{-1}(t)$
         //transitivity
         foreach (v $\in$ $\mathbf{sources}(x)\setminus${s,t})
            result.add(m[{s,v}] $\uplus$ m[{v,t}])
         //congruence closure
         foreach ($f(\overline{w})$ $\in$ x.gfas)
            foreach ($\overline{u},\overline{v} \in \mathbf{sources}(\overline{w})$)
               if ($\overline{u} \neq \overline{v}$)
                  result.add($\biguplus_i$ m[{$u_i$,$v_i$}])
      //Check if set has changed
      if (result$\setminus$m[{s,t}] $\neq$ $\emptyset$)
         enqueueSuperTerms(todo,{s,t})
      m[{s,t}].add(result)

   return m[{s0,t0}] $\setminus$ {s0,t0}

   method enqueueSupers({s,t}:Pair[GT])
      var x := $\mathbf{sources}^{-1}(s)$
      foreach (u,v $\in$ $\mathbf{sources}(x)$)
         if ({u,v} $\in$ RP)
            todo.enqueue({u,v})
      foreach (y $\in$ rg.superTerms[x])
         foreach (u,v $\in$ $\mathbf{sources}(y)$)
            if ({u,v} $\in$ RP)
               todo.enqueue({u,v})
```

Figure 5.31: Justification extraction algorithm
The algorithm finds the fixed point for the equations in figure 5.32.
The algorithm first calculates the subset of Γ that is relevant, using the algorithm from figure 5.22.
The algorithm works bottom-up, starting at interface equalities in Γ.
The map `m` is an under-approximation of the function P, constructed as a fixed point.
The method `enqueueSuperTerms` ensures changes in `m` are propagated, until saturation.

$P(s,t) \triangleq PR(s,t) \setminus \{s,t\}$

$\forall s \cdot \emptyset \in P(s,s)$

$\forall s,t \cdot \{s = t\} \in \Gamma \Rightarrow \{s \neq t\} \in P(s,t)$

Transitivity
$\forall s,t \cdot \forall u \in \mathbf{sources}(s') \setminus \{s,t\} \cdot PR(s,u) \uplus PR(u,t) \subseteq P(s,t)$

Congruence closure
$\forall s,t \cdot \forall f(\overline{w}) \in s', \overline{u}, \overline{v} \in \mathbf{sources}(\overline{w}) \cdot \overline{u} \neq \overline{v} \Rightarrow \biguplus_i PR(u_i, v_i) \subseteq P(s,t)$

Where:
$U \uplus V \triangleq \{S \cup T \mid S \in U \wedge T \in V\}$

Figure 5.32: Justification extraction from g_n^i
A justification is a set of dis-equalities on I.
PR(s,t) is a partial function, only defined if $s' = t'$
The set PR(s,t) is the least fixed point of the above equations.
The building blocks of justifications are negations of equalities in Γ.
The rules mimic transitivity, reflexivity and congruence closure, collecting justifications for equality.

$N_1 = \{f(a_2, x_1) = c_2, f(b_2, x_1) = d_2\}$
$N_2 = \{c_3 = c_2, d_3 = d_2, a_3 = a_2, b_3 = b_2\}$
$N_3 = \{a_3 = b_3, f(c_3, m) \neq f(d_3, m)\}$

Figure 5.33: Example for transitive propagation of goals
The goal $\{c_3, d_3\}$ is propagated as c_2, d_2 to N_1.

unification as well, but some ideas from basic superposition (e.g. [NR92a]) might help us reduce the number of unifications that need to be considered.

5.3.6 Related work

Scoping:
Lexical scoping in the form we are using was introduced in ALGOL 60 ([BBG$^+$60]).

Unit ground equality (UGFOLE) interpolation

Interpolation for FOL was first introduced in [Cra57] with a constructive proof (for non-clausal, non-ground FOL) in [Lyn59]. The use of interpolation for verification was introduced by McMillan in [McM03a], where interpolants for propositional logic are extracted from SAT refutations and used to refine an invariant. There has been since a lot of work on the generation and use of interpolants. Earlier work concentrated on propositional interpolants, mostly extracted from a SAT solver proof for UNSAT on a bounded instance of a model. Later work extended interpolation to GFOLE, linear arithmetic ([McM05]) and some other theories. Most work has concentrated on extracting interpolants from SMT proofs. We deal with the non-unit fragment in the section 5.4. Universally quantified clauses do not admit interpolation - there are pairs of sets of clauses whose union is inconsistent, but for which there is no universal CNF interpolant - we discuss this issue in 7.

In [McM05] a calculus is given for proofs in UGFOLE which generates an interpolant from a refutation essentially by tracking the assumptions from (in our terminology) N^{b} in equalities derived in N^{t} (the system also generates interpolants for the combination of unit GFOLE and linear rational arithmetic with inequalities). The calculus used to generate proofs is similar to our CC variants and is less efficient than graph or rewriting based methods.

[BJ15] gives a survey on ground interpolation, for propositional logic, equality logic and theories.

In [FGG$^+$12], the authors present a graph based interpolation method for UGFOLE. The algorithm uses a congruence closure (CC) graph that keeps enough information to recover a proof for each equality derived (as in [NO05]). The main difference from our algorithm is that they use a single CC graph (the proof) and extract an interpolant from it, while we extract an interpolant from two (or more) separate graphs.

In their algorithm, each node represents one term and is colored as in our formulae (red, black or blue), Equality edges are colored only red or blue (a pre-process rewrite step ensures the graph is colorable), Equality edges stemming from axioms are coloured according to the source of the axioms. Equality edges derived from transitive or congruence closure are coloured according to the colour of the two nodes they connect, for interface equalities either color may be used. The algorithm collects N^{b} premises (blue edges and paths) and constructs the interpolation clauses. The authors also suggest a generalization which is an interpolation game (for general fragments, not just UGFOLE). Two players start each with a set of formulae and, at each stage, exchange formulae, where each formula communicated from one player is implied by that player's set of formulae, and added to the other's set, and is in the common vocabulary. The interpolants are constructed as implications where the right side is a formula communicated forward and the left side is the conjunction of all formulae communicated backwards until that point. It is easy to see that our basic algorithm is an instance of this game.

The extension of this game to sequence-interpolants seems simple, but the extension to trees or DAGs is not immediately obvious. The interpolation game is a general framework for generating interpolants, the authors present one instance (in addition to unit ground equalities) that uses colorable proofs, which we discuss in the next section. While our algorithm is similar to, and inspired by, theirs, the aim is different - we generate the proof and the interpolant together, rather than transform the proof in order to derive an interpolant. Our algorithm can be used for generating intrpolants from one set of equations to a set of sets of equations, which is not clear how to do with their algorithm without generating a graph per set of equations. We also add the mechanism for communicating relevance, and interpolation for trees and DAGs.

Other approaches include the encoding of equality to propositional logic and then propositional invariant generation ([KW07]) and color based approaches, which we discuss in section 5.3.

Interpolant classes

In [JM06] the authors introduce sequence interpolants, and the idea of searching for a proof and interpolant in logical fragments of increasing strength incrementally, this is done in the context of progressively unrolling a program. Sequence interpolants have since seen much research, mostly in the context of propositional logic (e.g. [VG09]).

[RHK13] gives a classification of different classes of Horn clauses, where our DAG-CFG fits in the class of recursion free linear Horn-Clauses, as does [AGC12a]. Our tree-shaped CFG is encoded as linear head-disjoint recursion free Horn clauses in their classification. The tree interpolants of [BGKK13] are different as they represent a tree of meets rather than branches and joins, and is used for modeling concurrency and recursion. [AGC12a] extracts a DAG interpolant from a proof of the unreachability of an error state in an unfolding of the program. This interpolant is used, together with abstract interpretation, to strengthen the loop invariants in the program. If insufficient the loop is unrolled further. While our DAG-interpolants are similar, we extract them while searching for a proof while they extract the interpolant from an SMT proof.

Model based approaches

Most approaches described above produce a global proof (mostly using SAT or SMT) and extract an interpolant from the proof. Other systems use resolution based coloured local proofs and we discuss these in section 5.4. Another successful approach is that of IC3/PDR ([Bra11], [EMB11]). In this approach (roughly) a program is unrolled and a failing trace is searched locally *backwards* - each unrolled instance checks for satisfiability vs. a candidate invariant for the pre-state, if a counter example (CEX) is found it is propagated to a source in the predecessor instance (through the transition relation) and then satisfiability is checked there - if a satisfiable path is found to the initial state, the program does not verify, otherwise a clause is derived that "blocks" the CEX (a clause implied by the current candidate invariant and transition relation) and this clause is propagated forward (through the transition relation) in order to strengthen the invariant of successors. In addition checks are done on candidate invariants to see if any of them is an actual invariant for the whole program, in which case the program is proven.

Our approach is similar in that the proof is local and produced incrementally, and also in that information flows in both directions. The main difference is that in IC3 the information that flows backwards is full models that are guaranteed, if found feasible, to lead to an assertion failure, while in our system each request (or backward communicated equation) represents a set of models, where the guarantee is that this set of models includes a model for all suffix paths to assertions (in IC3 we usually have one assertion at the end). The complexity difference is that IC3 may request an exponential number of models (if clause generalization does not manage to block them), and finding each potential model can take, in the worst case, exponential time (a SAT-solver call), while each such model is guaranteed to be a real model of a suffix. Furthermore, IC3 is formulated for sequence interpolants, while in a DAG, if we want to only propagate models that lead to assertion failures, we would have to propagate a model on each suffix path - again a potential exponential factor. The total number of requests and responses in our system can also be exponential (as we have seen some problems have only exponential clausal interpolants), but we do not have the exponential factor from branches and joins, as we over-approximate the set of suffix models. We pay the price of over-approximating the set of suffix models by generating useless interpolation clauses. Our algorithm is parametric in Γ - if we were to use a Γ that is exactly the set of models of suffixes, the complexity would be similar to IC3 - it would be interesting future work to find an abstraction for Γ less coarse than ours, but not as precise as per-path models - this might also be useful for IC3 if we could request the refutation of a set of models rather than one by one.

5.4 Ground Clause Interpolation

The verification algorithm presented in section 4.1 for ground programs is complete based on the completeness of ground superposition. However, once we enforce scoping, completeness is lost, as we have seen in examples even for the ground unit fragment.

In this section we present our algorithm for ground interpolation in the CFG, and show it is sound and complete using a modification of the model generation proof. For non unit clauses, we use the ideas from the unit algorithm - approximating the equational theory of suffixes. In this section we only work with terms and clauses and not with EC-nodes, clause-ECs or EC-graphs. We discuss binary interpolation, but only use Γ rather than $\mathrm{N^b}$, and hence we show completeness for any $\mathrm{N^b}$ which Γ approximates.

The main idea is to apply the superposition calculus to $\mathrm{N^t}$ with, instead of standard unification, unification modulo an equality theory that approximates a model for $\mathrm{N^b}$.

The standard completeness proof for superposition (e.g. [BG91]) shows how to build a model for a set of clauses saturated w.r.t. superposition that does not contain the empty clause, where the model is the smallest congruence that includes the maximal equalities of non-redundant clauses from the set.

We define a set of equalities Γ over $\mathfrak{L}^{\mathrm{I}}$ that is constructed from maximal equalities of clauses in $\mathrm{N^b}$ and hence over-approximate a model for $\mathrm{N^b}$, and we define a congruence relation $=_\Gamma$ that satisfies all equations in Γ. and saturate $\mathrm{N^t}$ w.r.t. superposition with unification modulo $=_\Gamma$. Whenever unification in an inference uses information from $=_\Gamma$, we add the assumptions used, as a set of negated equations, to the conclusion, in order to preserve soundness, as $=_\Gamma$ is an over-approximation. This process implies, in fact, two-way communications between $\mathrm{N^t}$ and $\mathrm{N^b}$, as the maximal equality of a clause in $\mathrm{N^b}$ can add an equality to Γ which can enable a derivation in $\mathrm{N^t}$ which in turn produces a clause in $\mathrm{N^b}$.

We show completeness using the model from the standard proof for $\mathrm{N^b}$, and extending it with a construction similar to the standard model for $\mathrm{N^t}$ that is normalized w.r.t. $\mathrm{N^b}$. We use a partial ordering $\succ_\mathrm{i}$ instead of $\succ$ in $\mathrm{N^t}$ which approximates any ordering on interface terms: $\mathrm{s} \succ_\mathrm{i} \mathrm{t}$ only if, for any $\mathrm{N^b}$, the normal form of s (in a candidate model for $\mathrm{N^b}$) is greater than the normal form of t. The algorithm in this section can be seen as an extension of (the ground part of) both [McM08] and [BW13], adjusted to account for the two-way communication we have described for the unit ground fragment. It is also related to [KV09]. We discuss the differences in the related work section.

5.4.1 Basics

In this section we discuss the basic ideas of our algorithm for superposition based ground interpolation.
The basics of the algorithm are:

- Superposition in $\mathrm{N^t}$ is done with unification modulo an equality theory $=_\Gamma$ that over-approximates a model of $\mathrm{N^b}$.
  Communication from $\mathrm{N^b}$ to $\mathrm{N^t}$ proceeds through refinement of the approximation $=_\Gamma$ while communication from $\mathrm{N^t}$ to $\mathrm{N^b}$ is done through derived interpolation clauses over $\mathfrak{L}^{\mathrm{I}}$.

- Any clause derived under an equality assumption from $=_\Gamma$ is qualified with the assumption.
- For superposition in N^t, we use a separating partial order $\prec_i$ that approximates the total order by assuming that any pair of terms over $\mathfrak{L}^I$ is unordered. This is done by selecting a limit ordinal l^t and ensuring that the tkbo weight for each symbol in $\mathfrak{L}^{t+}$ is larger than l^t and for each symbol in $\mathfrak{L}^b$ it is smaller. We then perform ordering comparisons on an ordinal truncated at l^t.

We begin with an example - shown in figure 5.34

$N^t = \{c = f(a, x), d = f(b, x)\}$
$N^b = \{a = l, b = l, c \neq d\}$

Figure 5.34: The ordering is:
$x \succ\mid d \succ c \succ b \succ a \succ\mid l$

Here superposition would produces the following refutation:

$$\cfrac{\cfrac{\cfrac{\underline{a} = l \quad c = f(\underline{a}, x)}{\underline{f(l, u)} = c} \qquad \cfrac{\underline{b} = l \quad d = f(\underline{b}, x)}{\underline{f(l, u)} = d}}{c = d} \qquad c \neq d}{\square}$$

This refutation is not a local proof, as we have mixed-color clauses, literals and terms. Our algorithm first communicates from N^b to N^t the approximation of the model $=_\Gamma$, specifically $a =_\Gamma b$ derived from $\{\underline{a} = l, \underline{b} = l\}$. Next, N^t performs some superposition steps using unification modulo $=_\Gamma$ to generate an interpolation clause which is communicated to N^b. N^b uses the interpolation clause to derive the empty clause. We use the notation $a = b \vdash_\Gamma f(a, x) = f(b, x)$ to denote that the terms $f(a, x), f(b, x)$, which are in the sub-term closure of maximal terms of N^t, are equal under the equality theory $=_\Gamma$, using the assumption $a = b$. In order to construct Γ we use a rough over-approximation of the model for N^b by considering any two maximal terms in clauses from N^b that appear in maximal positive literals to be equal - the set of such maximal terms we name M. The process is summarized below:

At N^b:

$$\cfrac{\cfrac{\cfrac{\cfrac{\underline{a} = l}{a \in M} \qquad \cfrac{\underline{b} = l}{b \in M}}{a = b \in \Gamma}}{a = b \vdash_\Gamma a = b}}{a = b \vdash_\Gamma f(a, x) = f(b, x)}$$

At N^t:

$$\cfrac{\underline{f(a, x)} = c \quad \underline{f(b, x)} = d}{b \neq a \vee \underline{d} = c}\; a = b \vdash_\Gamma f(a, x) = f(b, x)$$

The interpolation clause is now $b \neq a \lor \underline{d} = c$

Back at N^b:

$$\dfrac{\dfrac{\underline{a} = l \qquad \dfrac{\underline{b} = l \qquad \dfrac{b \neq a \lor \underline{d} = c \quad \underline{d} \neq c}{\underline{b} \neq a}}{\underline{a} \neq l}}{l \neq l}}{\square}$$

Notation

We use two limit terms l^t and l^b to describe the minimal top term and minimal interface term, respectively. In practice, we allow N^t to include some terms that are smaller than l^b, for example theory constants (e.g. numbers) which are always at the bottom of the ordering, and e.g. the first DSA version of a method parameter used in both pre- and post-conditions. This addition does not affect soundness or completeness, and we leave it out of the formalization for simplicity.
For the ordering $\succ$, we use $\succeq$ for $\succ \cup =$.
We use $\succcurlyeq$ between terms to mean that any sub-term of the left term is greater than the right term - specifically, we write $l^t \succ t \succcurlyeq l^b$ to mean that $t \in \mathfrak{L}^I$.

5.4.2 Approximation of the bottom model

We describe now our interpolation calculus.
We use our basic approximation $=_\Gamma$ for the model of N^b and the encoding of assumptions in derivations, and later discuss enhancements to both these parts.
We define first the set of maximal terms of clauses in a set of clauses S as **maxterms**(S) - formally:

Definition 5.4.1. Maximal terms
The set of maximal terms w.r.t. $\succ$
$\mathbf{maxterm}(C) \triangleq \max_{\succ} \{t \mid t \trianglelefteq C\}$
$\mathbf{maxterms}(S) \triangleq \{\mathbf{maxterm}(C) \mid C \in S\}$

We use a partial order $\prec_i$ to approximate $\prec$ as described below, we define the set of maximal terms w.r.t. this ordering, and maximal positive terms, as follows:

Definition 5.4.2. Maximal terms w.r.t. $\succ_i$
The set of maximal terms w.r.t. $\succ_i$
$\mathbf{maxterm_i}(C) \triangleq \{t \trianglelefteq C \mid \forall s \trianglelefteq C \cdot s \not\succ_i t\}$
$\mathbf{maxterms_i}(S) \triangleq \{\mathbf{maxterm_i}(C) \mid C \in S\}$
$\mathbf{maxterm_i^+}(C) \triangleq \{t \in \mathbf{maxterm_i}(C) \mid \exists s \cdot t = s \in C \land \forall A \in C \cdot A \not\succ_i t = s\}$
$\mathbf{maxterms_i^+}(S) \triangleq \bigcup\limits_{C \in S} \mathbf{maxterm_i^+}(C)$

The calculus to calculate Γ is described in figure 5.35.
The approximation of the model of N^b abstracts all terms smaller than l^b to be equal.

$$\frac{\mathrm{C} \vee \underline{\mathrm{l} = \mathrm{r}}}{\mathrm{l} = \mathrm{r} \in \Gamma} \qquad \begin{array}{l} \mathrm{l^t} \succ \mathrm{l}, \mathrm{r} \succcurlyeq \mathrm{l^b} \\ \mathrm{l} = \mathrm{r} \succ \mathrm{C} \end{array}$$

$$\frac{\mathrm{C} \vee \underline{\mathrm{l}} = \mathrm{r}}{\mathrm{l} \in \mathrm{M}} \qquad \begin{array}{l} \mathrm{l^t} \succ \mathrm{l} \succcurlyeq \mathrm{l^b} \\ \mathrm{r} \not\succcurlyeq \mathrm{l^b} \\ \mathrm{l} \succ \mathrm{r} \\ \mathrm{l} = \mathrm{r} \succ \mathrm{C} \end{array}$$

$$\frac{\mathrm{l} \in \mathrm{M} \quad \mathrm{r} \in \mathrm{M}}{\mathrm{s} = \mathrm{t} \in \Gamma} \qquad \begin{array}{l} \mathrm{s}, \mathrm{t} \preceq \mathrm{l^t} \\ \mathrm{l} \trianglelefteq \mathrm{s} \trianglelefteq \mathbf{maxterms}^{+}_{\mathbf{i}}(\mathrm{N^t}) \\ \mathrm{r} \trianglelefteq \mathrm{t} \trianglelefteq \mathbf{maxterms}_{\mathbf{i}}(\mathrm{N^t}) \end{array}$$

Figure 5.35: The calculation of the simple over-approximation Γ of the model for $\mathrm{N^b}$
The first rule communicates a maximal interface equality directly.
The second rule defines M as the maximal terms of clauses in $\mathrm{N^b}$ that are in $\mathfrak{L}^{\mathrm{I}}$ but for which the normal form in the candidate model is not representable in $\mathfrak{L}^{\mathrm{I}}$- this corresponds exactly to our requests.
The third rule equates any two super terms of requested terms that are sub-terms of maximal terms (one positive) - an over-approximation.

The relation $=_\Gamma$ is defined as the smallest congruence that satisfies all of the equalities in Γ.
The third rule defining Γ states that each pair of interface terms with each at least one sub-term that has a normal form that is not in the interface (that is, includes a term over $\mathfrak{L}^{\mathrm{b}+}$) is considered potentially equal.
There are two reasons for this rough rule:
The first reason is the way we handle joins - as we have seen in chapter 4, when we have to use a non-unit fall-back at joins with EC-graphs, we get a clause where the maximal literal is an equation on a sub-term of our maximal term, hence for the over-approximation to be complete we need to consider all super-terms of each requested term.
The second reason is that we do not communicate any equality where the larger term includes non-interface terms - for example, consider the interpolation problem in figure 5.36:

$$\mathrm{N^t} = \left\{\mathrm{C}, \mathrm{D}, \neg\mathrm{A} \vee \underline{\mathrm{g}(\mathrm{c}, \mathrm{y})} = \mathrm{x}, \neg\mathrm{B} \vee \underline{\mathrm{g}(\mathrm{f}(\mathrm{b}), \mathrm{y})} \neq \mathrm{x}\right\}$$
$$\mathrm{N^b} = \left\{\mathrm{A}, \mathrm{B}, \neg\mathrm{C} \vee \underline{\mathrm{c}} = \mathrm{f}(\mathrm{h}(\mathrm{a}, \mathrm{m})), \neg\mathrm{C} \vee \underline{\mathrm{h}(\mathrm{a}, \mathrm{m})} = \mathrm{l}, \neg\mathrm{D} \vee \underline{\mathrm{b}} = \mathrm{l}\right\}$$

Figure 5.36: interpolation over-approximation example
The ordering satisfies $\mathrm{c} \succ \mathrm{b} \succ \mathrm{f}(\mathrm{h}(\mathrm{a}, \mathrm{m})) \succ \mathrm{h}(\mathrm{a}, \mathrm{m}) \succ \mathrm{l}$

Here, the normal form of both c and f(b) in the model for $\mathrm{N^b}$ is $\mathrm{f}(\mathrm{l})$, but the only terms communicated (as M) are $\{\mathrm{c}, \mathrm{b}\}$. The inference we get is:

$$\mathbf{idas}_\Gamma(\mathrm{s},\mathrm{t}) \triangleq \bigvee \mathbf{idass}_\Gamma(\mathrm{s},\mathrm{t})$$

$$\mathbf{idass}_\Gamma(\mathrm{s}=\mathrm{f}(\bar{\mathrm{u}}),\mathrm{t}=\mathrm{g}(\bar{\mathrm{v}})) \triangleq \begin{cases} \emptyset & \mathrm{s} \equiv \mathrm{t} \\ \{\mathrm{s} \neq \mathrm{t}\} & \mathrm{s},\mathrm{t} \prec \mathrm{l}^{\mathrm{t}} \\ \bigcup_i \mathbf{idass}_\Gamma(\mathrm{u_i},\mathrm{v_i}) & \mathrm{f} \equiv \mathrm{g} \end{cases}$$

Figure 5.37: Top interface disagreement set definition
$\mathbf{idas}_\Gamma(\mathrm{s},\mathrm{t})$ is a partial function, only defined if $\mathrm{s} =_\Gamma \mathrm{t}$

$$\frac{\neg A \vee \underline{g(c,y)} = x \quad \underline{g(f(b),y)} \neq x \vee \neg B}{c \neq f(b) \vee \neg A \vee \neg B \vee x \neq x} \; c \neq f(b) \in \mathbf{idas}_\Gamma(g(c,y),g(f(b),y))$$

The interpolant is $\{C, D, \neg A \vee \neg B \vee f(b) \neq c\}$

Note that this encoding of Γ is more coarse than the one we had for unit equalities. We could replace it with our encoding of g^i_n from the previous section but include all maximal equalities, not just unit equalities. We use the simpler definition above for simplicity.

Encoding assumptions:
In order to encode the assumptions from $=_\Gamma$ we use the interface top disagreement set $\mathbf{idas}_\Gamma(s,t)$ for two terms s,t where $s =_\Gamma t$. This set is the justification set we had in section 5.3.5, simplified.

For any s,t s.t. $s =_\Gamma t$, $\mathbf{idas}_\Gamma(s,t)$ is an interface clause (disjunction of disequalities) that satisfies
$N^b \models \mathbf{idas}_\Gamma(s,t) \vee N^b \models s = t$
This is, in a sense, a negation of an interpolant, as $\mathbf{idas}_\Gamma(s,t) \vee s = t$ is a tautology.
The formal definition is found in figure 5.37.
We select recursively the top interface-terms that are not identical and add their negation to the clause - for example
$\mathbf{idas}_\Gamma(g(f(a),y),g(f(b),y)) = f(a) \neq f(b)$
rather than $a \neq b$.
However, note that this is the weakest such interpolant, but not the most concise - for example:
$\mathbf{idas}_\Gamma(g(a,f(a),y),g(b,f(b),y)) = f(a) \neq f(b) \vee a \neq b$
Which is equivalent to, but less concise than
$a \neq b$
And also
$\mathbf{idas}_\Gamma(g(a,b,c,y),g(c,a,b,y)) = a \neq c \vee b \neq a \vee b \neq c$
Where a more concise option is
$a \neq b \vee b \neq c$.
A simple method to get optimal assumptions would be to work with equality constrained clauses (e.g. as in [NR92b]) where the equality constraint will include all our assumptions. We can represent the equality constraint as an EC-graph and extract justifications only when the clause is in $\mathfrak{L}^I$. This simplifies also the implementation for more precise versions of Γ described later. We leave this as future work.

5.4.3 Interpolation ordering

We use a partial ordering $\prec_i$ for our interpolation calculus, which is an approximation of the total order $\prec$:
The order is essentially a truncated tkbo (transfinite Knuth-Bendix ordering described in section 2)- ttkbo - the definition is as per tkbo except:

- Each interface term has a tkbo weight of 0
- Each interface term is unordered with any other interface term

For example:
$N^b = \left\{ A \lor \underline{a} = l, A \lor \underline{f(b)} = m \right\}$
Assuming that $y \succ x$ but $\text{weight}(y) = \text{weight}(x)$:
$g(a, x), g(f(a), x), g(f(b), x)$ are unordered w.r.t. $\prec_i$.
$g(y, a) \succ_i g(x, f(b)) \succ_i g(f(b), y) \succ_i g(b, x)$.
Note that, although N^t knows that b is its own normal form, it cannot order the normal forms of $b, f(b)$, while N^b can.
The following properties of the ordering are easy to see:

Lemma 5.4.1. $\succ_i \subseteq \succ$
$s \succ_i t \Rightarrow s \succ t$
$\forall s, t \succeq l^t \cdot s \rhd t \Rightarrow s \succ_i t$

For N^b we use the standard ordering $\prec$.
When we are interpolating in the CFG, each EC-node will have one pair of l^t, l^b for all direct predecessors and one for all direct successors, and will apply our interpolation calculus SP_I with the pair for the successors.
We could use the abstraction Γ also for $\succ_i$, so that we could order more terms (e.g. interface terms that do not have any sub-term as a maximal term in N^b). This would mean that the ordering might change when a new clause is derived at N^b.
The problem is that the ordering can be inverted - for example, if $b \succ_i a$ because neither b nor a are maximal terms in N^b, but once $b = l$ is derived at N^b, we cannot anymore order them, and so some more inferences might become valid. On the one hand it is likely that many interface terms never become maximal terms, and so we can prevent more derivations from taking place, on the other hand, the overhead of incrementally updating the ordering might be expensive. We leave this as future work.

5.4.4 Interpolation superposition calculus

The modified superposition calculus we use for interpolation (for binary interpolation, for N^t) is shown in figure 5.38.

The calculus SP_I is a straightforward extension of our unit ground interpolation algorithm to non-unit clauses. The main differences from the standard calculus $\mathbf{SP_g}$ are:

- We use $\not\prec_i$ instead of $\succ$, as $\prec_i$ is a partial order.
- Instead of the standard (ground) unification $l = s|_p$ we use $l =_\Gamma s|_p$
  We write it as $B = \mathbf{idas}_\Gamma(l, s|_p)$ and use B in the conclusion

When each premise P satisfies $P \succeq l^t$:

$$\mathrm{sup}^{i}_{=} \quad \frac{C \lor \underline{l} = r \quad \underline{s} = t \lor D}{B \lor C \lor D \lor s\,[r]_p = t}$$

(i)$l \not\preceq_i r$, (ii)$l = r \not\preceq_i C$
(iii)$s \not\preceq_i t$, (iv)$s = t \not\preceq_i D$
(v)$s = t \not\preceq_i l = r$
(vi)$\mathbf{l}, \mathbf{s} \succeq \mathbf{l^t}$
(vii)$\mathbf{B} = \mathbf{idas}_{\Gamma}(l, \mathbf{s}|_{\mathbf{p}})$

$$\mathrm{sup}^{i}_{\neq} \quad \frac{C \lor \underline{l} = r \quad \underline{s} \neq t \lor D}{B \lor C \lor D \lor s\,[r]_p \neq t}$$

(i)$l \not\preceq_i r$, (ii)$l = r \not\preceq_i C$
(iii)$s \not\preceq_i t$, (iv)$s \neq t \succeq D$
(v)$s \neq t \not\prec_i l = r$
(vi)$\mathbf{l}, \mathbf{s} \succeq \mathbf{l^t}$
(vii)$\mathbf{B} = \mathbf{idas}_{\Gamma}(l, \mathbf{s}|_{\mathbf{p}})$

$$\mathrm{res}^{i}_{=} \quad \frac{C \lor \underline{\underline{s \neq t}}}{B \lor C}$$

(i)$s \neq t \succeq C$
(ii)$\mathbf{s} \succeq \mathbf{l^t}$
(iii)$\mathbf{B} = \mathbf{idas}_{\Gamma}(s, t)$

$$\mathrm{fact}^{i}_{=} \quad \frac{C \lor l = r \lor \underline{s = t}}{B \lor C \lor r \neq t \lor s = r}$$

(i)$s \not\preceq_i t$, (ii)$s = t \not\preceq_i C \lor l = r$
(iii)$l \not\preceq_i r$
(iv)$\mathbf{s}, \mathbf{l} \succeq \mathbf{l^t}$
(v)$\mathbf{B} = \mathbf{idas}_{\Gamma}(s, l)$

When, for each premise P, $P \prec l^t$, we use the standard calculus $\mathbf{SP_g}$.
There are no binary inferences between premises C, D s.t. $C \succ l^t \succ D$.

Figure 5.38: The interpolation superposition calculus SP_I

- We add (disjunct) the top interface disagreement set of the unification to the conclusion of each rule
- The calculus applies only to clauses in $\mathrm{N^t}$ - we use SP for clauses in $\mathrm{N^b}$

Definition 5.4.3. Colored clause sets
A set of clauses N is colored (w.r.t. $\mathrm{l^t}, \mathrm{l^b}, \prec$) iff
$\forall \mathrm{C} \in \mathrm{N} \cdot (\mathrm{C} \succcurlyeq \mathrm{l^b} \vee \mathrm{C} \prec \mathrm{l^t})$.
We repeat the definitions from before for the two parts of N:
$\mathrm{N^t} \triangleq \{\mathrm{C} \in \mathrm{N} \mid \mathrm{C} \succcurlyeq \mathrm{l^b}\}$
$\mathrm{N^b} \triangleq \{\mathrm{C} \in \mathrm{N} \mid \mathrm{C} \prec \mathrm{l^t}\}$
As $\mathrm{l^t} \succ \mathrm{l^b}$, $\mathrm{N^t} \cap \mathrm{N^b} = \emptyset$.

The calculus $\mathrm{SP_I}$ preserves the property of being colored:

Lemma 5.4.2. Preservation of the coloured property by $\mathrm{SP_I}$
Our calculus has the property that:
$\forall \mathrm{N} \cdot \mathrm{N} = \mathrm{N^t} \cup \mathrm{N^b} \Rightarrow \mathrm{SP_I(N)} = (\mathrm{SP_I(N)})^{\mathrm{t}} \cup (\mathrm{SP_I(N)})^{\mathrm{b}}$
Which means that if we begin with a colored set of clauses, the result of any derivation in $\mathrm{SP_I}$ will also be colored.

Completeness: The completeness proof for our ground interpolation calculus is given in section A.1.1 of the appendix. The proof follows the common model generation method for completeness proofs, where the model is divided to a model for $\mathrm{N^b}$ and a a model for $\mathrm{N^t}$.

5.4.5 Ordering for Interpolation

In this section we describe how we establish an adequate total order on ground terms for interpolation for the whole CFG.

Extending our interpolation calculus to a DAG CFG requires that we have a separating total order on all program constants, so that whenever a constant in scope at a CFG-node is not in scope in the successor, it is separated from the any constant that is in scope in the successor.
For a linear program, we can simply assign decreasing limit ordinals to each CFG-node, so that the root node has the maximal limit ordinal and each successor has a lower limit ordinal. We then assign each constant the weight $\mathrm{n}\omega + 1$ where n is the limit ordinal index of the last CFG-node where it appears.
However, this solution does not extend immediately to tree-shaped CFGs, as different branches may imply different orderings. Consider, for example, figure 5.39:

```
n1 :
   assume a≠b
if (*)
   n2 :
      assume f(a)=f(b)
   n3 :
      assert f(a)=c
else
   n4 :
      assume f(a)≠f(b)
   n5 :
      assert f(b)=c
```

Figure 5.39: Interpolation ordering for trees

The **then** branch implies that b $\succ\!|$ a while the **else** branch implies that a $\succ\!|$ b. There is no total order with minimal scoping that respects the separating constraint for this program.

Our solution to this problem is to add a further DSA version for every constant at a branch that is in scope in at least one successor of the branch, unless it is in scope at the join of the branch. For figure 5.39 the modified program is shown in figure 5.40:

```
n1 :
   assume a≠b
if (*)
   n2.0 :
      assume a1=a
      assume b1=b
   n2 :
      assume f(a1)=f(b1)
   n3 :
      assert f(a1)=c
else
   n4.0 :
      assume a2=a
      assume b2=b
   n4 :
      assume f(a2)≠f(b2)
   n5 :
      assert f(b2)=c
```

Figure 5.40: Interpolation ordering for trees - additional DSA versions

After this transformation, and after ensuring that the scope for each constant is contiguous (by renaming one instance of a constant that appears on two parallel branches but neither at branch nor join point), we can choose any topological order of the nodes and assign limit ordinal indices accordingly.

5.4.6 Related and future work

The main difference in our work from other works is the setting in which we work - verification in a DAG-shaped CFG. Our work is closely related to [BW13] and [McM08]. The main difference from both papers, beside the setting, is that our calculus remains in the ground fragment for ground programs, while other works use a form of abstraction that adds quantified variables to a clause - the complexity of most proof steps is lower for ground clauses.

In [BW13] (which is an extension of [BGW94]), the authors present a superposition calculus that works modulo a theory. Their calculus essentially calculates interpolants between a set of clauses and a theory - the difference with our calculus, besdies that we remain in the ground fragment for a ground input, is that we can use equalities from the theory to direct interpolation, hence improving accuracy.

The theory is encoded as the rule

$$\dfrac{C_1....C_n}{\square}$$

Where $C_1..C_n$ are a set of clauses over the signature of the theory which are unsatisfiable in the theory.
The calculus of [BW13] is similar to superposition, except that clauses that participate in the proof are converted using *weak abstraction:*
Roughly, weak abstraction is application of the following transformation:
$C[t]_p \rightarrow X \neq t \lor C[X]_p$
where t is a term in the theory vocabulary (roughly equivalent to $\mathfrak{L}^b$ in our case) that does not contain normal free variables and X is an *abstraction variable*, which is roughly a free variable that can only be instantiated with theory terms - only terms that are in $\mathfrak{L}^b$.
Unification respects abstraction variables in the sense that, for the mgu σ, $\sigma(X) \in \mathfrak{L}^b$ and $\mathfrak{L}^b$ includes all abstraction variables.
For example 5.34, using superposition with weak abstraction the derivation is roughly equivalent to:

$$\dfrac{\dfrac{f(\underline{a},x)=c}{X\neq a\lor \underline{f(X,u)}=c}\qquad\dfrac{f(\underline{b},x)=d}{Y\neq b\lor \underline{f(Y,u)}=d}}{X\neq a\lor X\neq b\lor c=d}\ \sigma=\{Y\mapsto X\}$$

And then, for the theory N^b:

$$\begin{array}{c} X\neq a\lor X\neq b\lor c=d \\ \vdots \\ \square \end{array}$$

The main difference with our approach is that, if $N^b \not\models a = b$, we can avoid the above superposition derivation in N^t (if the approximation of I^b is precise enough) while their system does not take N^b into account in unification, and hence we can ensure completeness (for the ground fragment) while allowing strictly less derivations.
In their system, using the above example, the following is *not* a valid derivation:

$$\frac{\dfrac{f(\underline{a}, z) = c}{X \neq a \lor \underline{f(X, z)} = c} \qquad \underline{f(y, z)} = d}{y \neq a \lor c = d} \ \sigma = \{X \mapsto y\}$$

Because the unifier maps an abstraction variable X to a term not in $\mathfrak{L}^b$. Hence, their calculus avoids several unnecessary derivations (derivations that would be allowed if abstraction used normal variables as in the original [BGW94]) without losing completeness (for the ground fragment).
Abstraction is restricted (roughly) to top-background terms, meaning, in our context, terms over $\mathfrak{L}^b$ whose direct super-term is not in $\mathfrak{L}^b$ - for example for:
$\forall x \cdot g(f(b, c), f(h(a, x), z)) = d$
The weak abstraction is:
$X \neq a \lor Y \neq f(b, c) \lor g(Y, f(h(X, x), z) = d$
Where b,c, are not abstracted because they are not top N^b terms (f(b,c) is their direct super-term), and x is not abstracted because it is a (non-abstraction) variable.
[BW13] shows a calculus for non-ground superposition with an opaque background theory, shows completeness in the ground case, and some conditions for completeness in the non-ground case (i.e. N^t includes non-ground clauses).
The main advantage of our method is that we use an approximation of the theory (in our case N^b) to restrict abstraction further (so that our communication is bi-directional), and we only perform the equivalent of abstraction when necessary for unification, so that we remain in the ground fragment for a ground problem - hence the interpolant also remains ground. The authors discuss an improvement to their calculus that defines a sub-set of the theory terms ($\mathfrak{L}^b$) as *domain elements* which are all known to be unequal to each other (e.g. numbers). They show that it is not necessary to abstract these elements (as, essentially, they are in their normal form already). In our calculus, if e.g. numbers are at the bottom of the ordering, we get the same effect. The non-ground aspects of the comparison with the non-ground version of our calculus will is discussed in section 7.5.

In [McM08], the author presents a superposition calculus for deriving an interpolant between two sets of clauses, as in our setting.
The author uses a separating ordering similar to ours, and ensures (ground) completeness using the following *procrastination* rule:

$$\frac{\underline{a} = l \quad \underline{f(a, x) = c}}{f(l, x) = c} \ \to \ \frac{\underline{f(a, x)} = c}{\forall y \cdot y \neq a \lor f(y, x) = c \mid a \succ y}$$

Where y is a fresh normal variable, and the constraint $\mid a \succ y$ is an ordering constraint (as in [NR01] section 5). Roughly, the constraint means that any unifier σ used in an inference with the clause, it must hold that $a \succ \sigma y$. In this system, ordering constraints are used to ensure that the proof is local. The author mentions (discussion of lemma 3) that ordering constraints could be checked for feasibility and then discarded, rather than propagated (which adds a significant overhead), and the conclusion of non-local derivations discarded.
Note that the ordering constraints are, in a sense, a more precise variant on weak abstraction, as they constrain the newly introduced variables to be smaller

than a given term, while essentially weak abstraction constrains the abstraction variables to be smaller than l^t. We write the above replacement as the following rule in proofs:

$$\frac{C \vee \underline{l} = r \quad s \bowtie t \vee D}{\forall x \cdot x \neq s|_p \vee C \vee D \vee s[x]_p \bowtie t \mid s|_p \succ x} \; s \succ l^t, l = s|_p \qquad \begin{array}{l} (i) l \not\succeq r, (ii) l = r \not\preceq C \\ (iii) s \not\prec t, (iv) s \bowtie t \not\prec D \\ (v) s \bowtie t \not\preceq l = r \\ (vi) \mathbf{s} \succeq \mathbf{l^t} \succ \mathbf{l} \end{array}$$

For example 5.34, the proof would be as follows:

$$\dfrac{\dfrac{\underline{a} = l \quad \underline{f(a, x)} = c}{\forall y \cdot y \neq a \vee \underline{f(y, x)} = c \mid a \succ y} \qquad \dfrac{\underline{b} = l \quad \underline{f(b, x)} = d}{\forall z \cdot z \neq b \vee \underline{f(z, x)} = d \mid b \succ z}}{\forall y \cdot y \neq b \vee y \neq a \vee \underline{d} = c \mid a \succ y \wedge b \succ y} \; \sigma = \{z \mapsto y\}$$

Where y,z are normal variables. We could replace $a \succ y \wedge b \succ y$ with $a \succ y$ as $b \succ a$.
This is followed, in N^b (dropping the ordering constraints):

$$\dfrac{\dfrac{\dfrac{\underline{a} = l \qquad \dfrac{\underline{b} = l \qquad \dfrac{\forall y \cdot y \neq b \vee y \neq a \vee \underline{d} = c \quad \underline{\underline{d \neq c}}}{\forall y \cdot \underline{b} \neq y \vee a \neq y}}{\forall y \cdot \underline{a} \neq y \vee y \neq l}}{\forall y \cdot y \neq l \vee y \neq l}}{l \neq l}}{\square}$$

The derivations above use the fact that a,b are maximal equality terms in order to drive abstraction, as opposed to [BW13] which (in a different setting) produces an abstraction for *each* term in $\mathfrak{L}^I$.
Dropping the ordering constraints at N^t makes the abstraction not weak, similarly to [BGW94] (i.e., in the above example, the variables y,z above can be unified with terms in $\mathfrak{L}^{t+}$such as x).

The first difference of our approach is that we always remain in the ground fragment as long as the input is ground (and do not introduce fresh variables otherwise), and so we produce ground interpolants. One might consider adding an un-abstraction rule to interpolation clauses (similar to that suggested in [BW13]) in order to produce ground interpolants:

$$\frac{\forall x \cdot x \neq t \vee C}{C[t/x]} \qquad \begin{array}{l} (i) C, t \prec l^t \\ (ii) x \ntrianglelefteq t \end{array}$$

The second difference is that their approach abstracts each clause separately, even if it is not a candidate for any inference, while our approach performs essentially a combined step of abstraction (or procrastination), superposition and then unabstraction. Our Γ is, in a sense, a summary of all the possible abstractions of maximal terms.

Other related work

Superposition modulo Shostak theories: Another related work is [GHW03], which shows a superposition calculus that works modulo a Shostak theory. A Shostak theory ([Sho84],[BDS02b]) is an alternative approach to satisfiability modulo a theory, where a Shostak theory is composed of two elements: a canonizer which is a function σ that rewrites a term to its normal form in the theory, and a solver, which essentially evaluates an equation, returns whether the equation holds, does not hold, or a set of equalities on variables in the equation if the theory does not imply the equation or its negation.
The authors use *blackbox path ordering* which is an instance of lexicographic path ordering (LPO), that satisfies some additional properties, of interest to us is that it is *compatible with canonizer application* - that is, roughly, $\mathrm{t} \succeq \sigma(\mathrm{t})$ where σ is the canonizer function and t an arbitrary term. As our EC-graphs can be thought of as rewriting to normal form, and as we have used some rewriting to normal form for the integer theory, it would be interesting to see how ttkbo interacts with such a canonizer.

Different ordering: We require at most as many limit ordinals as there are CFG-nodes, and hence our ttkbo weights are always within ω^ω.
However, as our $\mathrm{l}^\mathrm{t}, \mathrm{l}^\mathrm{b}$ are decreasing as we traverse the CFG in program execution direction, program extension (e.g. loop unrolling) might require us to shift all calculated ttkbo weights by some $\mathrm{n}\omega$, which can be implemented easily.

In [KV09], for two sets of clauses $\mathrm{N}^\mathrm{t}, \mathrm{N}^\mathrm{b}$ over scopes $\mathfrak{L}^\mathrm{t}$,$\mathfrak{L}^\mathrm{b}$ respectively , the search for an interpolant uses an ordering where
$\mathfrak{L}^{\mathrm{t}+} \succ\mid \mathfrak{L}^\mathrm{I}$ and
$\mathfrak{L}^{\mathrm{b}+} \succ\mid \mathfrak{L}^\mathrm{I}$
while in our case and in [McM08]
$\mathfrak{L}^{\mathrm{t}+} \succ\mid \mathfrak{L}^\mathrm{I} \succ\mid \mathfrak{L}^{\mathrm{b}+}$

The main advantage of their work is that they do not need to modify the superposition calculus. Our main advantage is that our ordering extends to trees and DAGs - which is required for our verification algorithm.

The first promises completeness for ground local proofs as there cannot be any valid inference with premises in both $\mathfrak{L}^{\mathrm{t}+}$and $\mathfrak{L}^{\mathrm{b}+}$(proven in [KV09] for ground FOL with equality and linear arithmetic by showing by induction that such an inference is never valid). However, it does not extend readily for sequence (or tree or DAG) interpolants as each non-leaf, non-root CFG-node has at least two interfaces, with a predecessor and a successor, and so the ordering constraints cannot be all satisfied. Their solution is to run an interpolation instance per interface.

The second extends readily to sequence and tree, and DAG interpolants, however, SP is not complete as-is for local proofs for cases as we have seen above. For this reason we have added unification relative to an equational theory, [McM08] adds procrastination and [BW13] adds weak abstraction.

Ordering:
Our ordering is oriented and separating in the style of [McM08], meaning that each term that is in scope at a node but not at its successor is *larger* in the ordering. The idea of separation using tkbo was presented in [LW07] motivated by hierarchic theorem proving and further discussed in [KMV11]. TKBO is mentioned as a potential separating ordering for interpolation in [KV09], while [McM08] suggests a variant of recursive path ordering (RPO). We have found

TKBO more appropriate for sequence, tree and DAG interpolants as it is easy to generate weights that satisfy the ordering constraints, for interface terms we can share the calculation of the weight (evaluation of $\prec_i$ is still done on a truncated weight - for each CFG-node there is a designated l^{t}), and often $\prec$ / $\prec_i$ can be determined using just the weight comparison, without resorting to the recursive inspection of terms.

Chapter 6

Bounded fragments

In this section we present several restrictions of the various fragments we have described that allow us to ensure polynomial space and time bounds. Our bounds are based on restricting the size of each term, literal or clause according to several different measures of size, and also the derivation depth.
Our objective is to have a predictable verification tool, with a predictable maximal run-time for a given selection of parameters and, given a proof for a property of the program, a simple check whether the proof lies within the bound defined by the set of parameters - regardless of random seeds, processor performance etc. For this reason all of our bounds are independent of order of evaluation - for example, we do not restrict the total number of terms or clauses, only their structure. For several of the fragments we only show that the maximal time and space are polynomial, without showing the exact degree, which remains as future work.

6.1 Term depth

One of the simplest ways to limit the size of a term is to limit the maximal term depth. For a finite signature with at least one function of more than unary arity, the maximal size of a term is exponential in the maximal term depth. The other factors are the number of functions, function arities (especially maximal arity) and, in the multi-sorted case, the actual sorts of function arguments and result. The formal definition of the depth of a term is:
$\mathrm{d}(\mathrm{f}(\bar{\mathrm{t}})) \triangleq \mathrm{d}(\mathrm{f}) + \mathrm{d}(\bar{\mathrm{t}})$
$\mathrm{d}(\bar{\mathrm{t}}) \triangleq \max_{\mathrm{i}} \mathrm{d}(\mathrm{t}_\mathrm{i})$
$\mathrm{d}(\mathrm{x}) \triangleq 1$
We assume the function $\mathrm{d}(\cdot)$ is given as a parameter for function and variable symbols - from here on we assume that the function has the value 1 for all function and variable symbols.

For the unsorted case, for the depth d and a number c of constants, the total number of terms is defined recursively as follows:
$\mathrm{G}(1) = \mathrm{c}$
$\mathrm{G}(\mathrm{d}+1) = \sum_{\mathrm{f} \in \mathrm{F}_{>0}} \mathrm{G}(\mathrm{d})^{\mathbf{arity}(\mathrm{f})}$
Where $\mathrm{G}(\mathrm{d})$ is the number of terms up to size d and $\mathrm{F}_{>0}$ is the set of non-

constant function symbols.

To get an asymptotic upper bound, first we assume that all non-nullary functions are of the maximal arity m, and denote by f the number of non-constant functions - we get the following recurrence:
$G(1) = c$
$G(d+1) = c + fG(d)^m$
Which we can bound by $G(d)^m \geq c$ for $d > 1$:
$G(d+1) \leq (f+1)G(d)^m$
And so we get:
$$G(d) \leq (f+1)^{\sum_{i=0}^{d-1} m^i} c^{m^d} = (f+1)^{\frac{m^d-1}{m-1}} c^{m^d} \approx O\left((f+1)^{m^{d-1}} c^{m^d}\right)$$

We can see that the number of potential terms grows double-exponentially with the maximal arity and with depth, and that the number of constants is somewhat more significant than the number of non-nullary functions. Most importantly, for a given finite signature and maximal depth, the number of terms is polynomial in the number of constants. We expect proofs of many program properties to fit within very low depth restrictions (e.g. 3 or at most 4), and the signatures we have encountered do not usually include functions of arity more than 4. In addition, as mentioned before, we can always reduce the maximal arity to be 2 (at the cost of deeper expressions).

6.1.1 Relative depth

The intuition behind enforcing depth limits is the search for *small* proofs. In an attempt to limit the size (total number of function predicate and variable symbol occurrences) in a proof DAG, the depth of a term is a natural candidate. As we have seen in chapter 3, many (but not all) of the examples that exhibited more than linear minimal sized proof used terms of a depth that was at least linear in the input size. For any given program P, the minimal term depth $d_{min}(P)$ that is sufficient to represent the program is the maximal depth of any ground term appearing in the program (definitions for general clauses follow later, but do not change the basic notions).

Our objective is incremental verification, and hence we want to try and verify the program with some term depth limit d_0, simplify and trim the program according to the derived equalities (and clauses in general), and then, if there are still unproven assertions, proceed with depth d_1, which could be, for example, $d_0 + 1$. If we set $d_0 = d_{min}(P)$, we might miss the opportunity to discharge many simple assertions and simplify the program significantly before increasing the allowed term-depth - for example, consider the code in figure 6.1: here the minimal proof has a term depth of 4 as it must include the term $P(f(f(c,c), f(c,c)))$. There are many terms of depth 4 in the signature that can be obtained by resolution or instantiation, including $P(f(f(d,d), f(d,d)))$. Intuitively, $P(f(f(c,c), f(c,c)))$ is nearer to the original problem than
$P(f(f(d,d), f(d,d)))$ as $f(f(c,c), f(c,c))$ appears in the original problem but the nearest term that includes d is d. The idea is that the absolute depth (or in general, size) of terms in the proof is less important than the *distance* of the term from terms that appear in the original program. Before discussing the concept of distance we present another motivating example, taken from the tree-rotation code of AVL-trees and shown in figure 6.2.

Here h_0, h_1, h_2 and $this_0, this_1$ are the DSA versions of the heap and **this**,

```
s :
 //Axioms
 assume $\forall x \cdot P(x) \Rightarrow Q(x)$
 assume $\forall x \cdot P(x) \Rightarrow P(f(x,x))$
$n_0$ :
 //Prelude
 assume $P(d)$
 assume $P(c)$
$n_1$ :
 assert $\neg Q(f(f(c,c),f(c,c)))$ // negated $Q(f(f(c,c),f(c,c)))$
```

Figure 6.1: Relative term depth example

```
s :
 //Heap axioms
 assume $row_{1a} : \forall h,x,f,v,y,g \cdot x = y \vee rd(wr(h,x,f,v),y,g) = rd(h,y,g)$
 assume $row_{1b} : \forall h,x,f,v,y,g \cdot f = g \vee rd(wr(h,x,f,v),y,g) = rd(h,y,g)$
 assume $row_2 : \forall h,x,f,v \cdot rd(wr(h,x,f,v),x,f) = v$
$n_0$ :
 //Prelude
 assume $left \neq right$  //$left \neq right$
 assume $this \neq null$  //$this_0 \neq null$
$n_1$ :
 assume $this.left \neq this$  //$rd(h_0,this_0,left) \neq this_0$
  ...
 **if** (left != **null**)
  $t_0$ :
  // $rd(h_0,this_0,left) \neq null$
  **if** (right == **null**)
   $t_1$ :
    // $rd(h_0,this_0,right) = null$
   $t_2$ :
    lr := left.right      //$lr = rd(h_0,rd(h_0,this_0,left),right)$
    left.right := **this**   //$h_1 = wr(h_0,rd(h_0,this_0,left),right,this_0)$
    **this** := left          //$this_1 = rd(h_1,this_0,left)$
    right.left := lr      //$h_2 = wr(h_1,rd(h_1,this_1,right),left,lr)$
   $t_a$ :
    assert right.right==**null** //$rd(h_2,rd(h_2,this_1,right),right) \neq null$
```

Figure 6.2: Relative term depth example
Translated clauses are given in comments

$$
\begin{aligned}
&\mathrm{h}_1 = \mathrm{wr}(\mathrm{h}_0, \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}), \mathrm{right}, \mathrm{this}_0)\\
&\mathrm{this}_1 = \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_0, \mathrm{left})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}), \mathbf{right}, \mathrm{this}_0), \mathrm{this}_0, \mathbf{left})\\
&\quad = \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}) \quad (\mathbf{row_{1b}} : \Leftarrow \mathrm{left} \neq \mathrm{right})\\
&\mathrm{rd}(\mathrm{h}_1, \mathrm{this}_1, \mathrm{right}) =\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}), \mathrm{right}, \mathrm{this}_0), \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_0, \mathrm{left}), \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathbf{rd}(\mathbf{h_0}, \mathbf{this_0}, \mathbf{left}), \mathbf{right}, \mathrm{this}_0), \mathbf{rd}(\mathbf{h_0}, \mathbf{this_0}, \mathbf{left}), \mathbf{right})\\
&\quad = \mathrm{this}_0 \quad (\Leftarrow \mathbf{row_2})\\
&\mathrm{h}_2 = \mathrm{wr}(\mathrm{h}_1, \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_1, \mathrm{right}), \mathrm{left}, \mathrm{lr})\\
&\mathrm{rd}(\mathrm{h}_2, \mathrm{this}_1, \mathrm{right}) = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_1, \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_1, \mathrm{right}), \mathrm{left}, \mathrm{lr}), \mathrm{this}_1, \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_1, \mathrm{this}_0, \mathbf{left}, \mathrm{this}_1), \mathrm{this}_1, \mathbf{right})\\
&\quad = \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_1, \mathrm{right}) \quad (\mathbf{row_{1b}} : \Leftarrow \mathrm{left} \neq \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{h}_1, \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}), \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathbf{rd}(\mathbf{h_0}, \mathbf{this_0}, \mathbf{left}), \mathbf{right}, \mathrm{this}_0), \mathbf{rd}(\mathbf{h_0}, \mathbf{this_0}, \mathbf{left}), \mathbf{right})\\
&\quad = \mathrm{this}_0 \quad (\Leftarrow \mathbf{row_2})\\
&\mathrm{rd}(\mathrm{h}_2, \mathrm{rd}(\mathrm{h}_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{h}_2, \mathrm{this}_0, \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_1, \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_1, \mathrm{right}), \mathbf{left}, \mathrm{lr}), \mathrm{this}_0, \mathbf{right}))\\
&\quad = \mathrm{rd}(\mathrm{h}_1, \mathrm{this}_0, \mathrm{right}) \quad (\Leftarrow \mathbf{row_{1b}} : \mathrm{left} \neq \mathrm{right})\\
&\quad = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathbf{rd}(\mathbf{h_0}, \mathbf{this_0}, \mathbf{left}), \mathrm{right}, \mathrm{this}_0), \mathbf{this_0}, \mathrm{right}))\\
&\quad = \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{right}) \quad (\Leftarrow \mathbf{row_{1a}} : \mathrm{rd}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}) \neq \mathrm{this}_0)\\
&\quad = \mathrm{null}
\end{aligned}
$$

Figure 6.3: Equational proof of program from 6.2 att_2

respectively, rd, wr are map read and write operations, $\mathrm{left}, \mathrm{right}$ are field-id constants and `null` is a reference constant. The minimal term depth for representing the program is 3.
In figure 6.3 we show the part of an equation proof of the assertion that must be performed at t_2 (according to the scoping rules). One can immediately see that the maximal depth of a term in the proof is 4.
The property of this proof is that all sub-terms that appear in the proof are provably equal to a term in the original program.
If we look at the equivalence class of each term according to the equalities we can prove on this term, for example at t_2, the depth of the equivalence class of $[\mathrm{read}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left})]_2$ is 2, as it has only one member.
The members of the EC $[\mathrm{read}(\mathrm{h}_0, \mathrm{read}(\mathrm{h}_0, \mathrm{this}_0, \mathrm{left}), \mathrm{right}), \mathrm{lr}]_2$ have the depth 3 and 1, respectively, what should be the depth of the EC-node?

Assuming that we have other axioms that could generate other terms up to our depth limit, we want a depth measure that first searches for a proof using the existing terms and terms shown equivalent (so that they share the same EC-graph node), and only later try to generate new terms. Our definition for relative term depth assigns each EC-node with the minimum of the depth of its EC-members. The term $\mathrm{rd}(\mathrm{h}_2, \mathrm{rd}(\mathrm{h}_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right})$ appears only at the CFG-node t_a, and must participate in the proof. The constants h_2, this_1 appear at t_2 but not at t_1, while h_0, h_1, this_0 do not appear at t_a. If we enforce minimal scoping, the above term could be represented at t_1 as follows (repeated sub-terms are named)

$\mathrm{rd}(\mathrm{h}_2, \mathrm{rd}(\mathrm{h}_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right}) =$

$\mathrm{rd}(s_6, \mathrm{rd}(s_6, s_3, \mathrm{right}), \mathrm{right})$

$s_1 \triangleq \mathrm{rd}(h_0, \mathrm{this}_0, \mathrm{left})$	$d = 2$
$s_2 \triangleq \mathrm{wr}(h_0, s_1, \mathrm{right}, \mathrm{this}_0)$	$d = 3$
$s_3 \triangleq \mathrm{rd}(s_2, \mathrm{this}_0, \mathrm{left})$	$d = 4$
$s_4 \triangleq \mathrm{rd}(s_2, s_3, \mathrm{right})$	$d = 5$
$s_5 \triangleq \mathrm{rd}(h_0, s_1, \mathrm{right})$	$d = 3$
$s_6 \triangleq \mathrm{wr}(s_2, s_4, \mathrm{left}, s_5)$	$d = 6$

This representation is quite large - the depth (both relative and absolute) is 8. On the other hand, h_1 ,h_0, this_0 are out of scope at t_a, so that the parts of the proof shown above cannot be deduced in t_a (there is no term in the scope of t_a that can represent h_0 - this would require extensionality). Hence we see that if we look for a proof in increasing relative depth, we are forced to find the proof at t_2, at a relative depth of 1.

If we define the relative depth of a term at an CFG-node to be the minimal depth of a member in its EC according to the equalities implied (by the whole program) at the CFG-node, then we can only bound it from above for any intermediate step in our verification process, as new equalities deduced at a later stage can add new members to the EC with a lower depth. In light of this we only state an upper bound for relative depths unless we have found that the term is equivalent to an original program term, at which case it is of relative depth 1 as described below.

For a given verification state (that is, a given EC-graph g_n at each EC-node n), we denote the upper bound on the depth of a term as the minimal depth of its *known* equivalence class, using $d_n^{\leq}(t)$ as the upper bound for the depth of the term t at CFG-node n - for example:
$d_{t_3}^{\leq}(\mathrm{rd}(h_2, \mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right})) = 1$
$d_{t_2}^{\leq}(\mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right})) = 2$
$d_{t_2}^{\leq}(\mathrm{rd}(h_2, \mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right})) = 3$
However, in our system we can still prove the above program at a maximal term depth of 2:
Initially we can only add $\mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right})$ to t_2 at relative depth 2 (we describe the reasoning behind adding ground terms in the quantification chapter).
Instantiating the axioms using E-matching allows us to derive
$\mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right}) = \mathrm{this}_0$ at t_2 without introducing any new terms, thus reducing the depth of the equivalence class:
$d_{t_2}^{\leq}(\mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right})) = d_{t_2}^{\leq}(\mathrm{this}_0) = 1$
Now:
$d_{t_2}^{\leq}(\mathrm{rd}(h_2, \mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right})) = 2$
Also $d_{t_2}^{\leq}(\mathrm{rd}(h_1, \mathrm{this}_0, \mathrm{right})) = 2$
We derive:
$\mathrm{rd}(h_2, \mathrm{rd}(h_2, \mathrm{this}_1, \mathrm{right}), \mathrm{right}) = \mathrm{rd}(h_1, \mathrm{this}_0, \mathrm{right})$, introducing only the new term $\mathrm{rd}(h_1, \mathrm{this}_0, \mathrm{right})$ of depth 2.
Now also
$d_{t_2}^{\leq}(\mathrm{rd}(h_0, \mathrm{this}_0, \mathrm{right})) = 2$
So we can derive
$\mathrm{rd}(h_1, \mathrm{this}_0, \mathrm{right}) = \mathrm{rd}(h_0, \mathrm{this}_0, \mathrm{right})$
Which triggers the propagation of $\mathrm{rd}(h_0, \mathrm{this}_0, \mathrm{right}) = \mathrm{null}$ by the EC-graph mechanism.

```
 ...
if ($c_1$) //from if (left != null)
 assume c_1
 assume left != null // $rd(h_0, this_0, left) \neq null$
 if ($c_2$) //from if (right == null)
  $t_1$ :
  assume c_2
  assume right == null // $rd(h_0, this_0, right) = null$
  $t_2$ :
   $t_{2a}$ :
   assert left != null    //negated $rd(h_0, this_0, left) = null$
  $t_3$ :
  lr := left.right    //$lr = rd(h_0, rd(h_0, this_0, left), right)$
   $t_{3a}$ :
   assert left != null    //negated $rd(h_0, this_0, left) = null$
  $t_4$ :
  left.right := this //$h_1 = wr(h_0, rd(h_0, this_0, left), right, this_0)$
  this := left        //$this_1 = rd(h_1, this_0, left)$
   $t_{4a}$ :
   assert right != null    //negated $rd(h_1, this_1, right) = null$
  $t_5$ :
  right.left := lr    //$h_2 = wr(h_1, rd(h_1, this_1, right), left, lr)$
  $t_a$ :
  assert right.right==null //$rd(h_2, rd(h_2, this_1, right), right) \neq null$
```

Figure 6.4: Relative term depth example 2

All steps are performed on term-ECs of relative depth at most 2.

We use o_n for the set of EC-nodes in g_n that represent terms in the original program at the CFG-node n (when a set of nodes is merged into one, the merged node is in o_n iff any of the nodes in the set was). Our new depth definition would be, for an EC-node n in the EC-graph g_n at the CFG-node n, the least fixed point of:

$d^{\leq}_{n}(n) = \textbf{if } n \in o_n \textbf{ then } 1 \textbf{ else } \min_{f(\bar{t}) \in n}(d^{\leq}_{n} f(\bar{t}))$

$d^{\leq}_{n}(f(\bar{t})) = 1 + \max_{i}(d^{\leq}_{n} t_i)$

The least fixed point is well defined as the function is monotonic.

We could, of course, select any non-negative constant for original program terms.

Caveats: The local EC-node depth can be somewhat sensitive to the formulation of the VC, for example, if we take the above program and add a nullity check before each heap dereference, we get the program in figure 6.4.

Here, for a relative term depth bound of 2:

At t_5 we must apply one axiom in order to even allow $rd(h_2, rd(h_2, this_1, right), right))$ to be represented.

At t_4, $rd(h_1, rd(h_1, this_1, right), right)$ is only allowed after we establish that $rd(h_1, this_1, right) = this_0$.

The program shown in figure 6.5 can only be proven with depth 3. The term $rd(h_0, rd(h_0, this_0, left), left)$ requires a depth of 3 at t_2, while if we were to merge t_2 to t_3 we would have a depth of 1, and similarly if we were to merge t_2 with t_1. Essentially, a term equivalent to the depth 3 term appears in both a successor and a predecessor of the CFG-node, and hence a user of the verification tool

```
...
assume this ≠ null
if (c₁) //from if (left.left != null)
 t₁ :
 assume c₁
 assume left.left!= null    //rd(h₀, rd(h₀, this₀, left)) ≠ null
 t₂ :
 right := null //h₁ = wr(h₀, this₀, right, null)
 //rd(h₁, this₀, left) =
 //    rd(wr(h₀, this₀, right, null), this₀, left)
 //    rd(h₀, this₀, left)  (⇐ row1b :left ≠ right)
 //rd(h₁, rd(h₁, this₀, left), left) =
 //    rd(wr(h₀, this₀, right, null), rd(h₀, this₀, left), left)
 //    rd(h₀, rd(h₀, this₀, left), left)  (⇐ row1b :left ≠ right)
 t₃ :
 assert left.left!=null //negated rd(h₁, rd(h₁, this₀, left), left) = null
 //rd(h₁, rd(h₁, this₀, left), left)
```
Figure 6.5: Relative term depth limitation

might expect a proof search at t_2 to include the term.

Inheriting the *original* term-ECs from either the successor or the predecessor would mean that the size of o_n is proportional to the number of DSA versions of variables rather than to the number of variables in the program, which somewhat defeats the purpose of scoping. We have not found a more satisfactory way of determining the original set of term-ECs, and it remains as future work.

In light of the limitations of depth restrictions, we use these in combination with other methods to control the complexity. Note that most of the worst case complexity examples of chapter 3 are rejected using depth alone.

6.1.2 Enforcing the depth restriction

A simple technique to enforce the depth restriction in an EC-graph propagation algorithm is to apply the algorithm as is, calculate the depth for each node in the result, and trim away any EC-node (and potential-EC-node etc) as a post-process.

However, this would not be very efficient as we can have a quadratic (in the bound) number of EC-nodes that would be trimmed away (e.g. at a join, for E-unification the number is exponential) - hence we want to enforce the restriction at the construction level.

Our EC-graph join algorithm propagates equalities by adding sub-terms from predecessors top-down. We build the GFAs and EC-nodes top-down, starting with a known EC-node and adding GFAs to it until we reach a known tuple - for example, when `adding` the term c to the join n of
$\{[a]_0, [f(a)]_0, [c, f(f(a))]_0\} \sqcup \{[a, b]_1, [f(a, b)]_1, [c, f(f(a, b))]_1\}$, we begin by adding $[c]_n$, and descend the potential-relevant EC-node chain f(.),f(f(.)),f(f(a)) as long as both joinees agree on the function symbol and until we find an already existing EC-node or get stuck. We annotate each such potential-relevant EC-node with a *reverse depth* which marks its distance from an existing EC-node (an existing node must have a relative depth within the bound).

In our example, f(.) has a reverse depth of 1, f(f(.)) a reverse depth of 2 and f(f(a)) a reverse depth of 3. It is easy to see that if we have reached the reverse-depth bound and not encountered an existing EC-node, any EC-node that is built using the GT of maximal reverse depth will be out of the depth bound, and hence we do need to expand the chain further. In addition, we do not need requests and RGFAs that is deeper than the depth bound, as any response to such a request is not representable.

Applying all of the above ensures that, for each EC-graph g_n, the set of all representative terms of all GFAs in all versions of g_n (throughout the verification process) is bounded by the limit calculated above. Note that this still allows an exponential total number of possible states of g_n, as we can select any subset of the allowed GFAs and any subset of the possible equalities on them, but in any given verification run only a polynomial number of these can appear. This is where we differ from SAT/SMT as, because all of our operations (`add` and `assume`) are monotonic, the size of any sequence of non-redundant operations (operations that actually modify g_n) is linear in the maximal size of the data structure (after applying non-redundant operations exhaustively we get an EC-graph with all possible GFAs in one EC-node).

We have implemented size bound on construction both for the EC-graph and for E-unification.

6.2 Size bounds for clauses

For clauses, whether ground or non-ground, the bounds on terms give us an initial limit to the size of each clause, as each equality literal includes exactly two top terms and hence the number of possible literals is limited. This limitation is, however, usually insufficient for our purpose as the number of possible clauses is exponential in the number of possible literals which is exponential in the maximal depth.

For this reason we add a further bound that allows only a bounded number of literals per clause - a width bound. If the number of possible literals is l and the width bound is w, the unbounded number of possible clauses is 2^l while the bounded number is $O(l^w)$.

Some evidence is given in [BW01] that restricting clause width can help us restrict the search for shorter proofs (although they deal with propositional proofs). The results in the paper are for general resolution and tree-like resolution, while the propositional fragment of superposition is equivalent to ordered resolution. Ordered resolution is known to have minimal width proofs that are wider than those of general resolution with some orders. We use several propositional simplification inferences (that do not rely on ordering) in order to mitigate this, and have not encountered examples where clauses much wider than the input were needed in a proof.

Clause width is easy to enforce in inferences as the number of literals of the conclusion is usually easily computable from that of the premises. Enforcing term-depth bounds for ground superposition is also relatively straightforward except when we use relative depth - in that case the relative depth of the EC of a super-term may be lower than the relative depth of a sub-term. Our algorithm for ground superposition, when generating the term-EC $t[r]_p$ from $t[l]_p$ and r by replacement, attempts to perform the replacement bottom up and aborts at the first step the depth bound is not satisfied.

6.3 Derivation depth

A proof system that maintains a set of clauses and updates it using generating and simplifying inferences simulates the search for a DAG-shaped proof. We define the derivation depth of each proof-DAG-node as the length of the longest proof-DAG-path from the clause to a leaf. If a clause can be derived in several different ways, the shortest derivation depth of these proofs is taken.
One of the main features of our system is looking at the derivation depth *per-fragment* - each fragment being a set of inference rules. Given a proof-DAG that includes inferences from different fragments, a fragment F and a clause C derived in the proof, we count the maximum number of inferences in F along any path from a leaf to C.

Given a set of clauses saturated for some derivation depth d_F in each fragment F, there is a marked difference in the cost of saturating the set for derivation depth $d_F + 1$ for each fragment F. While fragments such as unit ground propagation can take almost linear time (depending on the indexing and search structures), in the non-ground superposition fragment, a clause with a maximal literal $x = y$ can have twice as many superposition inferences with a clause C as there are sub-term occurrences in C. A clause such as $x = y \vee z = w$ would have even more option. For a clause width w and a maximal size (number of sub-term occurrences) s for a literal we could have at least $2w \times sw$ inferences per pair of clauses in total.

Another factor is the interaction between the derivation depth and the different measures for terms and clauses - depth and width. In most inferences we consider, the maximal size measures of the result can be calculated easily from the premises, before performing unification or replacement. Relative depth makes this calculation an over-approximation, in which case we reject an inference only if an under-approximation shows the result is not within the bounds, and otherwise attempt the inference and abort it at the first indication that the result is not within the bounds.

Some fragments only decrease all size measures - for example, unit propagation. However, such a fragment might take a non-trivial time to saturate - for example, simplification by unit rewriting in the non-ground case requires non-trivial indexing and unification operations, even if it ensures that the result is smaller or equal to the original in all measures. Some fragments can also ensure an overall decrease in e.g. the tkbo order (which is well founded) of the whole problem (the result is smaller or equal to the original seen as a multiset of clauses ordered by tkbo), but it may take a large number of steps to saturate the fragment - for example, rewriting by ground unit equalities without EC-graphs (even ground) can take a quadratic number of steps to reach the normal form (e.g. rewrite rules $\{t_1 \rightarrow t_2, ..., t_{n-1} \rightarrow t_n\}$ and the term $f(t_1, ..., t_1)$ with f of arity n). We separate a subset of the fragments that ensure a decrease in all measures (or no modification - we do not want fragments that permutate the state), and ensure a low polynomial (or log-linear) time to saturate (without a depth bound), and apply them eagerly. These include the saturation of EC-graph propagation with the weak join, unit ground propagation and rewrite by unit equalities with EC-graphs (fully reduced rewrite system).

In light of the above, our strategy is to search for a proof of increasing derivation depth, but increasing the derivation depth of each fragment separately. We give higher priority to cheaper fragments and our system receives as a parameter the ratio between the derivation depth of different fragments.

The main technique in which our implementation enforces the derivation depth limitations described above is by dividing the proof process to passes, where each pass traverses the CFG in topological order (though see the implementation part for other possible evaluation orders), and applies a per-pass subset of the fragments. Each pass must also ensure the propagation of the relevant clauses from predecessors. Our implementation usually performs propagation for most fragments together in order to reduce the overhead of requests. This allows us to avoid annotating each clause with its derivation depth per-fragment.

6.4 Summary

We have seen how to obtain bounded fragments that limit the size and structure of clauses and terms that can appear in proofs, and how to combine such bounded fragments to get a predictable worst-case run-time. Limiting the shape of a proof in a local way (that is, limiting the shape of each clause or derivation step rather than e.g. the number of clauses or overall time-budget), allows users of a verification tool to predict the behaviour of the tool for a given program, and to adjust parameters accordingly. In addition, the results of the evaluation of each fragment is usable for other fragments, as the state of each fragment is representable as simply a set of clauses.
For several of the bounded fragments, the total worst-case run-time is still of a high polynomial degree, and so more work is needed in order to obtain a tractable and predictable verifier.

Chapter 7

Quantification

Many programs require reasoning that goes beyond ground reasoning with uninterpreted functions (GFOLE) for verifications. Domains used in programs include integer arithmetic, arrays, and heaps. In addition, many specification languages and verification techniques require reasoning with additional theories such as sets, sequences and maps. While dedicated decision procedures have been developed for some of the required domains, support for quantification in a prover allows users to model arbitrary concepts and domains that may not have a dedicated decision procedure.

In this section we describe the extensions needed for the previously described ground fragments in order to support universally quantified clauses. We discuss first the underlying superposition calculus we use and then the needed extensions to the propagation mechanism and for interpolation.

The main difference with the addition of universal quantification is that the fragment is no longer decidable, while most of the fragments we have seen until now have been at most of exponential complexity. In practice, even with the best systems for handling quantifiers, derivations tend to be very prolific, and particularly tend to create many new terms. Our main objective is to reduce the depth of non-ground inferences required to prove assertions by prioritizing ground inferences and certain instances of non-ground derivations that can be shown not to increase the problem size, such as simplifications.

7.1 Preliminaries

In this section we are dealing with unrestricted terms, literals and clauses over the entire signature, including free variables. In each clause, all free variables are implicitly universally quantified. Any set of assumptions and assertions in general FOL-E formulae can be brought to an equi-satisfiable universal clausal from (see e.g. [BEL$^+$01]). This transformation might introduce new function symbols through Skolemization. We use a standard superposition calculus based on [BG91], with a transfinite Knuth-Bendix ordering based on [KMV11]. We fix the terminology in this section.

We assume the set of free variables in each clause is distinct and rename variables otherwise (we discuss renaming later). We use V for the set of variables and T for the set of (ground and non-ground) terms. The multi-set of free-variables (needed for ordering) of a term t, FV(t), is:

$$\begin{array}{lll} \mathrm{FV}(x) & \triangleq & \{x\} \\ \mathrm{FV}(f(\bar{s})) & \triangleq & \bigcup\limits_i \mathrm{FV}(s_i) \end{array}$$

A substitution $\sigma \in V \to T$ is a function from variables to terms. We are only interested in finite substitutions, where σ is finite iff it differs from the identity function for only a finite number of inputs - $|\{v \in V \mid \sigma(v) \neq v\}| \in \mathbb{N}$.
For a term t and a substitution σ applying the substitution σ to t, denoted by $t\sigma$, is defined as follows:
$x\sigma = \sigma(x)$
$f(\bar{t})\sigma = f(\bar{t}\sigma)$
$(\bar{t}\sigma)_i = t_i\sigma$

The domain of a substitution σ is the set $\mathrm{dom}(\sigma) = \{v \in V \mid \sigma(v) \neq v\}$ and the range of a substitution is the set $\mathrm{ran}(\sigma) = \{x\sigma \mid x \in \mathrm{dom}(\sigma)\}$.

A renaming is a substitution σ s.t. $\mathrm{ran}(\sigma) \subseteq V$ and $|\mathrm{ran}(\sigma)| = |\mathrm{dom}(\sigma)|$ - that is, variables are mapped only to variables and no two variables are mapped to the same variable.

A composition of substitutions $\sigma\tau$ is simply functional composition -
$x(\sigma\tau) = (x\sigma)\tau$.

We say that a substitution is fully reduced if it is idempotent - essentially, no element of the range has any free variable that is in the domain - formally:
$\forall x \cdot \mathrm{FV}(x\sigma) \cap \mathrm{dom}(\sigma) = \emptyset$

We use a partial order on substitutions denoted by $\leq$. We say that σ is more general than τ if $\sigma \leq \tau$. The definition of the order is as follows:
$\sigma \leq \tau \overset{\triangle}{\Leftrightarrow} \exists \tau' \cdot \tau = \sigma\tau'$

A unifier of two terms $s, t \in T$ is a substitution σ s.t. $s\sigma = t\sigma$.
A most general unifier - mgu(s,t) of two terms $s, t \in T$ is a unifier σ for s,t that is more general than all other unifiers for s,t - formally:
$\sigma = \mathrm{mgu}(s,t) \overset{\triangle}{\Leftrightarrow} (\forall \tau \cdot (s\tau = t\tau \Rightarrow \sigma \leq \tau))$.
For first order logic each unifiable pair of terms has an mgu (shown in e.g. [BS01]) that is further unique up to renaming.

We use some quantifier instantiation in addition to superposition, as it allows some shorter proofs as we discuss below. Quantifier instantiation with EC-graphs requires E-unification. An E-unifier for an equational theory E (a congruence relation $=_E$) for two terms s,t is a substitution σ s.t. $s\sigma =_E t\sigma$. Depending on E, an E-unifiable pair s,t may have one mgu (up to renaming and E-equivalence) or the minimal size of a minimal set of minimal E-unifiers (that is, a set of E-unifiers where none is more general than another in the set and that every E-unifier is less general than one in the set) could be infinite or not exist at all (see [BS01]).
The direct extension of our ground algorithm with clause-ECs to quantified formulae does not require E-unification for completeness, as each clause-EC

represents the minimal member of its EC and hence we only need standard unification between maximal terms of minimal representative clauses.
However, in some cases, using E-unification can help reduce the derivation depth of some assertions. We discuss this in section 7.2.

An equality theory E is called finitely presented if there is a finite set of ground equations for which $=_{\mathrm{E}}$ is the smallest containing congruence.

Calculus

We use the standard superposition calculus with a transfinite Knuth-Bendix order. The order we use is, as before, tkbo, with the definition repeated in figure 7.1.

The underlying superposition calculus is taken from [BG91], where it is shown complete even with redundancy elimination, and presented in figure 7.2.

The order has two parameters:
A strict total order $\succ$ on the set of function symbols F.
A weight function $\mathrm{w} : \mathrm{F} \cup \mathrm{V} \to \mathcal{O}$ that satisfies
$\forall \mathrm{f} \in \mathrm{F} \cdot \mathrm{w}(\mathrm{f}) > 0$ and
$\forall \mathrm{x} \in \mathrm{V} \cdot \mathrm{w}(\mathrm{x}) = 1$.

The weight of a term is defined recursively as:
$\mathrm{w}(\mathrm{f}(\bar{\mathrm{s}})) \triangleq \mathrm{w}(\mathrm{f}) + \bigoplus_{\mathrm{i}} \mathrm{w}(\mathrm{s_i})$
Where, for free variables, $\mathrm{w}(\mathrm{x})$ is defined by w.

The transfinite Knuth Bendix ordering (tkbo) for terms:
$\mathrm{s} \succ \mathrm{t}$ iff $\mathrm{FV}(\mathrm{s}) \supseteq \mathrm{FV}(\mathrm{t})$ and

- $\mathrm{w}(\mathrm{s}) > \mathrm{w}(\mathrm{t})$ or
- $\mathrm{w}(\mathrm{s}) = \mathrm{w}(\mathrm{t}), \mathrm{s} \equiv \mathrm{f}(\bar{\mathrm{s}}), \mathrm{t} \equiv \mathrm{g}(\bar{\mathrm{t}})$ and

  $\mathrm{f} \succ \mathrm{g}$ or

  $\mathrm{f} \equiv \mathrm{g}$ and $\bar{\mathrm{s}} \succ \bar{\mathrm{t}}$

The order on tuples is lexicographic:
$\bar{\mathrm{s}} \succ \bar{\mathrm{t}} \overset{\triangle}{\Leftrightarrow} \exists \mathrm{i} \cdot ((\forall \mathrm{j} < \mathrm{i} \cdot \mathrm{s_j} = \mathrm{t_j}) \wedge \mathrm{s_i} \succ \mathrm{t_i})$

For an order $\succ$ on a set D, the multi-set extension of $\succ$, $\succ_{mult}$ on the set of multi-sets of D ($\mathrm{D} \to \mathbb{N}$) is defined as follows:
$\mathrm{S} \succ_{\mathrm{mult}} \mathrm{T} \overset{\triangle}{\Leftrightarrow} \forall \mathrm{t} \in \mathrm{T} \cdot \exists \mathrm{s} \in \mathrm{S} \cdot (\mathrm{s} \succ \mathrm{t} \vee (\mathrm{s} = \mathrm{t} \wedge \mathrm{S}(\mathrm{s}) > \mathrm{S}(\mathrm{t})))$
This extension is total if $\succ$ is, and orders multi-sets by the number of occurrences of maximal terms.
Literals are ordered with the multi-set extensions of the term ordering where the literal $\mathrm{s} = \mathrm{t}$ is represented by the multi-set $\{\mathrm{s}, \mathrm{t}\}$ and the literal $\{\mathrm{s} \neq \mathrm{t}\}$ by $\{\mathrm{s}, \mathrm{s}, \mathrm{t}, \mathrm{t}\}$.
Clauses are ordered with the multi-set extension of the literal ordering where each clause is the multi-set of its literals.
We overload the meaning of $\succ$ for the literal and clause orderings where there is no ambiguity.
A literal A is strictly maximal in a clause C (A not in C) when:
$\mathrm{A} \succ \mathrm{C} \overset{\triangle}{\Leftrightarrow} \forall \mathrm{B} \in \mathrm{C} \cdot \mathrm{A} \succ \mathrm{B}$
For the negation (which we use as an approximation with partial orders) we use a dual definition:
$\mathrm{A} \not\preceq \mathrm{C} \overset{\triangle}{\Leftrightarrow} \forall \mathrm{B} \in \mathrm{C} \cdot \mathrm{A} \not\preceq \mathrm{B}$
And similarly for non-strict maximality with $\mathrm{A} \succeq \mathrm{C}, \mathrm{A} \not\prec \mathrm{C}$.
The ordering and its multi-set extensions are stable under substitution and compatible with contexts:
$\forall \mathrm{C}, \mathrm{s}, \mathrm{t} \in \mathrm{T}, \sigma, \mathrm{p} \cdot \mathrm{s} \succ \mathrm{t} \Rightarrow \mathrm{C}\left[\mathrm{s}\sigma\right]_{\mathrm{p}} \succ \mathrm{C}\left[\mathrm{t}\sigma\right]_{\mathrm{p}}$
And additionally are total on ground terms.

Figure 7.1: Transfinite Knuth Bendix Ordering (tkbo)

$\mathrm{res}_=$	$\dfrac{\mathrm{C} \vee \mathrm{l} \neq \mathrm{l}'}{\mathrm{C}\sigma}$	$(\mathrm{i})\sigma = \mathrm{mgu}(\mathrm{l},\mathrm{l}')$ $(\mathrm{ii})\mathrm{l} \neq \mathrm{l}' \nprec \mathrm{C}$
$\mathrm{sup}_=$	$\dfrac{\mathrm{C} \vee \mathrm{l} = \mathrm{r} \quad \mathrm{s} = \mathrm{t} \vee \mathrm{D}}{(\mathrm{C} \vee \mathrm{s}\,[\mathrm{r}]_\mathrm{p} = \mathrm{t} \vee \mathrm{D})\sigma}$	$(\mathrm{i})\sigma = \mathrm{mgu}(\mathrm{l}, \mathrm{s}\|_\mathrm{p})$ $(\mathrm{ii})\mathrm{l}\sigma \not\preceq \mathrm{r}\sigma, (\mathrm{iii})(\mathrm{l} = \mathrm{r})\sigma \not\preceq \mathrm{C}\sigma$ $(\mathrm{iv})\mathrm{s}\sigma \nprec \mathrm{t}\sigma, (\mathrm{v})(\mathrm{s} = \mathrm{t})\sigma \not\preceq \mathrm{D}\sigma$ $(\mathrm{vi})\ \mathrm{s}\|_\mathrm{p} \notin \mathrm{V}$ $(\mathrm{vii})(\mathrm{s} = \mathrm{t})\sigma \not\preceq (\mathrm{l} = \mathrm{r})\sigma$
$\mathrm{sup}_{\neq}$	$\dfrac{\mathrm{C} \vee \mathrm{l} = \mathrm{r} \quad \mathrm{s} \neq \mathrm{t} \vee \mathrm{D}}{(\mathrm{C} \vee \mathrm{s}\,[\mathrm{r}]_\mathrm{p} \neq \mathrm{t} \vee \mathrm{D})\sigma}$	$(\mathrm{i})\sigma = \mathrm{mgu}(\mathrm{l}, \mathrm{s}\|_\mathrm{p})$ $(\mathrm{ii})\mathrm{l}\sigma \not\preceq \mathrm{r}\sigma, (\mathrm{iii})(\mathrm{l} = \mathrm{r})\sigma \not\preceq \mathrm{C}\sigma$ $(\mathrm{iv})\mathrm{s}\sigma \nprec \mathrm{t}\sigma, (\mathrm{v})(\mathrm{s} = \mathrm{t})\sigma \nprec \mathrm{D}\sigma$ $(\mathrm{vi})\ \mathrm{s}\|_\mathrm{p} \notin \mathrm{V}$
fact	$\dfrac{\mathrm{C} \vee \mathrm{l}' = \mathrm{t} \vee \mathrm{l} = \mathrm{r}}{(\mathrm{C} \vee \mathrm{t} \neq \mathrm{r} \vee \mathrm{l} = \mathrm{r})\sigma}$	$(\mathrm{i})\sigma = \mathrm{mgu}(\mathrm{l},\mathrm{l}')$ $(\mathrm{ii})\mathrm{l}\sigma \not\preceq \mathrm{r}\sigma$ $(\mathrm{iii})(\mathrm{l} = \mathrm{r})\sigma \nprec (\mathrm{C} \vee \mathrm{l}' = \mathrm{t})\sigma$ $(\mathrm{iv})\mathrm{l}'\sigma \nprec \mathrm{t}\sigma$

Figure 7.2: Superposition
The main differences from the ground case are:
The ordering $\succeq$ is partial (although potentially total on ground terms,literals and clauses).
Side condition (vi) for both superposition rules means that we do not need to consider superposition into a variable, which is undesirable as variables unify with all terms, it can be shown that the calculus is complete even with this condition.
In all ordering side conditions above (both for terms and literals), $\mathrm{u}\sigma \not\preceq \mathrm{v}\sigma$ is an over-approximation of $\exists\tau \cdot \mathrm{FV}(\mathrm{u}\sigma\tau, \mathrm{v}\sigma\tau) = \emptyset \wedge \mathrm{u}\sigma\tau \succ \mathrm{v}\sigma\tau$
which we use as it is easier to compute.
In fact, for requests, we employ a coarser approximation $\mathrm{u} \not\preceq \mathrm{v}$ rather than $\mathrm{u}\sigma \not\preceq \mathrm{v}\sigma$, as it is much easier to cache and propagate, and check the stricter side-conditions only when performing actual inferences. This also includes condition (vii) of $\mathrm{sup}_=$ which requires both premises.

Unification and indexing in the CFG

In automated theorem proving, unification is often closely tied with indexing - given a set of clauses, indexing algorithms calculate an index that summarizes the maximal terms of all clauses in a search data structure, and the maximal terms of any newly derived clause has to be matched with this index, so that in one operation we can find all potential clauses for superposition with the new clause (a similar index is used also for e.g. subsumption). The basics of indexing are described in [SRV01].
In our case, the number of clauses per CFG-node is relatively small, and so we have not experimented with advanced per-CFG-node indexing. However,

we have a slightly different indexing scenario if we look at the whole CFG - essentially, our request caches combined with the local CFG-node index are a form of a lazy global index. We ensure that each term covered by the local CFG-node index is also requested from all predecessors, so that the relevant clauses are propagated from all predecessors and the local index covers all relevant clauses in all transitive predecessors.

As discussed below, an overall goal for quantifier instantiation in our setting is to have an index at each CFG-node that can E-match (decide E-unifiablity) a term with a term in a transitive predecessor node using the disjunction of equality theories of each path between the CFG nodes, however, our data structures only roughly approximate this goal and it remains as future work.

Unification complexity

We repeat here some known results about unification complexity, and then describe the issue of unification complexity for bounded fragments. The minimal size for a fully reduced mgu for first order unification may be exponential, for example (from [BS01] example 2.8) - the mgu of:
$\mathrm{h}(\mathrm{f}(\mathrm{x}_0,\mathrm{x}_0),....\mathrm{f}(\mathrm{x}_{\mathrm{n}-1},\mathrm{x}_{\mathrm{n}-1}),\mathrm{f}(\mathrm{y}_0,\mathrm{y}_0),....\mathrm{f}(\mathrm{y}_{\mathrm{n}-1},\mathrm{y}_{\mathrm{n}-1}),\mathrm{x}_\mathrm{n})$ and
$\mathrm{h}(\mathrm{x}_1,...\mathrm{x}_\mathrm{n},\mathrm{y}_1,....\mathrm{y}_\mathrm{n},\mathrm{y}_\mathrm{n})$
A fully reduced mgu σ will have both $\mathrm{x}_\mathrm{n}\sigma$ and $\mathrm{y}_\mathrm{n}\sigma$ exponential sized terms containing 2^n instances of x_1 or y_1. Importantly, the weight of the term is exponential but the depth is linear. This exponential explosion can be mitigated by sharing sub-expressions (term-DAG rather than term-tree), which, combined with some more techniques, gives a linear time unification algorithm (see [PW78]).

E-unification:
For E-unification, the situation is slightly more complicated - for a finitely presented equality theory E the E-unification problem is NP-complete ([Koz77]). By extending the simple unification algorithm to E-unification, we can see that the potential number of unifiers we have to consider might be at most exponential (up to renaming and E-equivalence in the range).

(E-)unification for bounded fragments

In chapter 6 we introduce a fragment of FOLE that only allows clauses with terms less than a certain depth. In a bounded fragment, we are only interested in derivations where the conclusion respects some size bounds (depth and/or width).
In most cases, this would mean that we do not want to consider (E-)unifiers whose domain in the fully reduced form contains terms that are not within the bound - for example: If we have a depth bound of 4 and the following inference:

$$\frac{\mathrm{P}(\mathrm{x}) \vee \underline{\mathrm{f}(\mathrm{x},\mathrm{g}(\mathrm{g}(\mathrm{c})))} = \mathrm{c} \quad \mathrm{Q}(\mathrm{z}) \vee \underline{\mathrm{f}(\mathrm{g}(\mathrm{g}(\mathrm{z})),\mathrm{z})} \neq \mathrm{c}}{\mathrm{P}(\mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{c}))))) \vee \mathrm{Q}(\mathrm{g}(\mathrm{g}(\mathrm{c}))) \vee \mathrm{c} \neq \mathrm{c}}$$

The fully reduced unifier is $\{\mathrm{x} \mapsto \mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{g}(\mathrm{c})))), \mathrm{z} \mapsto \mathrm{g}(\mathrm{g}(\mathrm{c}))\}$ and hence superposition or resolution is not applicable in the bounded fragment as the literal

P(g(g(g(g(c))))) is not within the bound. Compare this to the inference (difference highlighted):

$$\frac{\mathrm{P}(\mathbf{c}) \vee \underline{\mathrm{f}(\mathrm{x}, \mathrm{g}(\mathrm{g}(\mathrm{c})))} = \mathrm{c} \quad \mathrm{Q}(\mathrm{z}) \vee \underline{\mathrm{f}(\mathrm{g}(\mathrm{g}(\mathrm{z})), \mathrm{z})} \neq \mathrm{c}}{\mathrm{P}(\mathbf{c}) \vee \mathrm{Q}(\mathrm{g}(\mathrm{g}(\mathrm{c}))) \vee \mathrm{c} \neq \mathrm{c}}$$

which uses the same unifier but the conclusion of the derivation is within the bound even though the unifier is not. Our implementation rejects the inference in this case as we abort unification when any term in the range of the unifier is not within the bound. A possible improvement would only reject unifiers where a variable in the domain that occurs in another term in one of the premises is too large. We have used the simplest unification algorithm based on Robinson's algorithm (described e.g. in [BS01]) which maintains a partial unifier that is fully reduced throughout the algorithm, and so we abort unification once an element in the range is too large.

7.2 Propagation and requests

In this section we describe how we extend our clause propagation mechanism for quantified clauses.

Term requests

For clause propagation (without EC-graphs) the request mechanism is not changed significantly from the ground case - the main differences are:

- Each clause may have more than one maximal literal and each literal may have more than one maximal term
- Requests are sets of non-ground terms rather than sets of ground terms as before (see discussion below)
- Requested terms are equivalent up to variable renaming, and might subsume each other
- Simplification by unit rewriting does not always subsume the larger premise, and similarly for unit propagation. An easy to detect case in which it does is when the unifier is a renaming

For the third point, consider the request {f(y)} followed by the request {f(g(x)), f(h(c)), f(z)} (x,y,z free variables) - the second request is subsumed by the first one. While it is not mandatory for completeness to detect this, it might improve performance. For the fourth point, the simplification rule is as follows:

$$\mathrm{simp}_{=} \quad \frac{\mathrm{l} = \mathrm{r} \quad \cancel{\mathrm{C}}}{\mathrm{C}\left[\mathrm{r}\sigma\right]_{\mathrm{p}}} \quad \begin{array}{l} (\mathrm{i})\mathrm{l}\sigma = \mathrm{C}|_{\mathrm{p}} \\ (\mathrm{ii})\mathrm{l}\sigma \succ \mathrm{r}\sigma \end{array}$$

The main difference is that the unifier must be the identity for the right premise, otherwise the result does not subsume the right premise. For unit propagation the condition is similar:

$$\text{unit} \quad \frac{\mathrm{A} \quad \cancel{\neg \mathrm{B} \vee \mathrm{C}}}{\mathrm{C}} \quad \text{(i)}\mathrm{A}\sigma = \mathrm{B}$$

Combination with EC-graphs

The combination of non-ground clauses with EC-graphs is also straightforward. Non-ground terms are represented as non-ground term-ECs which are similar to term-ECs except that, in addition to GFAs, nodes can contain also variables and non-ground function applications. We limit our representation so that a term-EC-node for a non-ground term contains only one member - a function application or a variable (that is, we do not include non-ground equalities in the graph). This implies that the non-ground part of the EC-graph is a DAG while the ground part can have cycles.

As mentioned earlier, we do not need to perform E-unification in order to ensure completeness for superposition, so that we use the standard unification algorithm for superposition on the representative of each term-EC (the least (by $\prec$) member of the EC).

Requests

For term requests for non-ground terms we use a rough over-approximation of the set of requested terms, both for the requests and the request cache at each CFG node. The reason is that it allows us an easy bound on the total size of the request cache (most requests have a negative response in our examples). This over-approximation is used also in the ground case in some cases where a source-chain is not complete, as described below.

Our requests for non-ground terms are an approximation for a term - basically we look at the top function symbol and one level deeper. For example, the term $\mathrm{f}(\mathrm{g}(\mathrm{x}), \mathrm{a})$ is abstracted as $\mathrm{f}(\mathrm{g}(?), \mathrm{a})$, the term $\mathrm{f}(\mathrm{x}, \mathrm{x})$ is abstracted as $\mathrm{f}(?, ?)$ and the term $\mathrm{f}(\mathrm{g}(\mathrm{c}), \mathrm{g}(\mathrm{b}))$ is abstracted as $\mathrm{f}(\mathrm{g}(?), \mathrm{g}(?))$. We use the abstracted request in addition to ground requests. We have found this abstraction mostly sufficient, as in most programs the majority of quantified clauses are axioms, and we do not use abstracted requests for axioms. The advantage of abstracted requests is that we need less CFG-traversal, while the disadvantage is that we may propagate clauses needlessly.

We store axioms at the root CFG-node so that they can benefit from simplifications, but we propagate axioms directly to each requesting CFG-node (propagation means encoding the axiom as a clause-EC at the CFG-node). An axiom is represented as a clause-EC in the root EC-graph, so we use normal unification on representatives to check if an axiom is relevant for a given CFG-node. Hence each CFG-node must remember (cache) which term-ECs it has imported axioms for.

The complexity of the request cache is, in theory, f^{d+1}, where f is the number of function symbols and d is the maximal arity of function symbols, as, for each leading function symbols, we use a set of tuples of arity d. Instead, we use a tuple of sets of function symbols (for each leading function symbol) for the requests and the cache, for a complexity of f^2d. This representation implies a further abstraction - for example, if we request $f(a, a)$ and $f(b, b)$ the request cache is $[f \mapsto \{a, b\} \times \{a, b\}]$ - so covers also $f(a, b), f(b, a)$. In practice, we have not found this request cache or the imprecision of requests to be a problem.

Literal requests

Literal requests behave much like term requests. For literal requests we have used an over-approximation which includes the predicate symbol and the leading function symbols of all arguments of the predicate symbol. We cache requests as for terms, with a predicate symbol instead of the leading function symbol.

7.3 Quantifiers in bounded fragments

In this section we discuss the specifics of enforcing bounds on the sizes of terms and clauses for the non-ground fragment. We have already discussed bounded unification and hence focus on bounding the total size of the representation of each CFG-node.

For ground fragments, it is easy to enumerate the set of terms, literals and clauses that can be generated from a given set of constants and functions with a bound on depth. For non-ground fragments, this is slightly more involved, as clauses are isomorphic up to variable renaming, but this only holds for literals and terms if it holds for the clause in which they occur - for example, consider the transitivity clause
$\forall x, y, z \cdot \neg P(x, y) \vee \neg P(y, z) \vee P(x, z)$
The two literals $\neg P(x, y), \neg P(y, z)$ are isomorphic modulo variable renaming, but cannot directly share a representation as they need to be distinct in the clause. Another clause
$\forall u, v, w \cdot \neg P(u, v) \vee \neg P(v, w) \vee P(u, w)$
is isomorphic to the first, and the isomorphism $\{x \mapsto u, y \mapsto v, z \mapsto w\}$ carries to literals and terms.

If we do not rename variables (to fresh versions) in the conclusions of inferences, then we need some more work in unification and indexing (as the sets of variables of premises may be non-disjoint), but we can share terms and literals between a clause and clauses derived from it, and no new variables are introduced in any derivation. This sharing is somewhat arbitrary as it depends on the orientation of the mgu selected for previous derivations (e.g.$\{x \mapsto y\}$vs.$\{y \mapsto x\}$). Avoiding the renaming of variables can also help with joins, as non-ground clauses that both originated from the same pre-branch clause can be more easily recognized. In light of the above, depth restrictions on terms and literals can only help us with establishing a complexity upper bound if we do not rename variables and ignore isomorphism up to renaming. With renaming we can only rely on the (non-ground) derivation depth to give an overall space complexity bound.

In order to define a useful bounded non-ground fragment we use a depth bound on clauses, terms and literals, and, in addition, a bound on the number of free variables in a clause. For a free-variable-count bound of n we can use the same n variables in all clauses and hence use the ground complexity bounds replacing the constant count c with c+n. When the set of free-variables of the premises is not disjoint the unification may need to rename some variables temporarily (e.g. add an index 0,1 to each variable from the left,right premise) and hence has to detect when too many variables are required in the conclusion of the rule - it is not necessary to do that at the unification level (as the maximal number of free-variables is, at most, double the bound, so the penalty is at most linear) although it could be more efficient. Bounding the total number of free variables that can participate in unification also helps us bound the time spent in calculating unifiers (along with the bound on the sizes of terms).

7.4 Instantiation

In addition to superposition, several theorem proving techniques use various forms of quantifier instantiation. We do not require instantiation for completeness, but it can help us reduce the derivation depth of some assertions.
We define instantiation as follows - for the set of clauses S and a clause $C \in S$, an instantiation is $C\sigma$ for some substitution σ. Usually σ is a function of S and C. A short survey of instantiation based methods is given in [BT12]. We say that an instantiation is ground when $C\sigma$ is ground. We usually derive the candidate substitution σ for an instantiation by unifying one or more terms (and possibly literals).

We call an instantiation of C with σ safe if the instance is unit and ground (after simplification) and all new terms in the instance are smaller than the terms used in the unification that defined σ. When an instantiation can be guaranteed to be safe we apply it eagerly.

SMT solvers often employ some form of ground instantiation to handle quantifiers (e.g. [RTdM14a]), although this method is usually not complete. Often, instantiation in SMT is based on the set of existing ground terms - all the ground terms of all derived clauses (see e.g. [DCKP12]).

While our superposition calculus does not require any additional instantiation, we have found that in some cases, especially those involving scoping, some instantiations are beneficial.
For example, consider the axiom $\forall x \cdot f(x) = g(x)$:
If the terms f(c) and g(c) appear in already derived clauses, and assuming we are working with EC-graphs, instantiating the axiom for $\{x \mapsto c\}$ can only reduce the size of the problem by merging (at least) two existing nodes in the EC-graph and potentially simplifying some clauses. We have also experimented with instantiating non-unit clauses such as $\forall x \cdot \neg P(x) \vee f(x) = g(x)$ when P(c) has been already derived and f(c),g(c) already occur in some clause - hence the result of instantiation adds only a new equality on existing terms.

Heaps: A specific case where instantiation is useful is with heap axioms - for example, assuming rd and wr are the read and write functions, respectively:
$\forall h', x', f', y', g', v' \cdot f' = g' \vee \underline{rd(wr(h', y', g', v'), x', f')} = rd(h', x', f')$
Here, for any substitution, the term $rd(wr(h', y', g', v'), x', f')$ is always greater (w.r.t. tkbo or number of symbols) than $rd(h', x', f')$ regardless of the instantiation, so it is safe (in the sense that instantiation only produces smaller terms).

In addition, the term includes all the free variables of the clause and hence any substitution that grounds it grounds the clause.
If we know that $\mathrm{f} \neq \mathrm{g}$ and we have unified $\mathrm{rd}(\mathrm{wr}(\mathrm{h}', \mathrm{y}', \mathrm{g}', \mathrm{v}'), \mathrm{x}', \mathrm{f}')$ with some ground term $\mathrm{s} = \mathrm{rd}(\mathrm{wr}(\mathrm{h}_0, \mathrm{y}, \mathrm{g}, \mathrm{v}_0), \mathrm{x}, \mathrm{f})$ and got the unifier σ, the simplified instance is $\mathrm{s} = \mathrm{rd}(\mathrm{h}_0, \mathrm{x}, \mathrm{f})$ and so the only new term in the instance is $\mathrm{rd}(\mathrm{h}_0, \mathrm{x}, \mathrm{f})$ which is smaller than s (by both tkbo, weight and depth).

Inspired by heap axioms, we identified some patterns for axioms that we can instantiate without enlarging the problem significantly, and without risking a matching loop - an unbounded sequence of instantiation. We instantiate the clauses where we can match a term or literal that includes all non-ground terms (essentially a single pattern), and where the depth of any term in the instance is smaller than the term matching the pattern. We also instantiate using leading predicate symbols, where we can use the polarity to detect a predicate symbol that only occurs non-ground in one polarity, and hence cannot, in itself, create a matching loop. We have found that even simple instantiation can allow us to discharge several assertions, hence leaving a simpler problem for stronger and costlier fragments.

7.5 Scoping for quantified clauses

In this section we discuss the way scoping works for non-ground clauses. Neither our system, nor any of the other systems discussed for ground interpolation, has an obvious extension that is complete in the non-ground case. In fact, there is not always an interpolant in the form of universally quantified clauses. We discuss some limited cases where completeness can be achieved and highlight the problem in extending the completeness proof of the ground case. We end the discussion by mentioning some related work and potential future directions for a complete system.

7.5.1 Incompleteness

With scoping, our system is not complete anymore, as first order logic is not complete for universal clausal interpolation (proof in e.g. [KV09]) - for example:
Consider the following two sets of clauses whose union is unsatisfiable:
$\mathrm{N}^{\mathrm{t}} = \{\forall \mathrm{y} \cdot \mathrm{f}(\mathrm{a}, \mathrm{y}) = \mathrm{c}\}$
$\mathrm{N}^{\mathrm{b}} = \{\forall \mathrm{x} \cdot \mathrm{f}(\mathrm{x}, \mathrm{b}) \neq \mathrm{c}\}$
The problem here is that the unifier has mixed colours:
$\mathrm{mgu}(\mathrm{f}(\mathrm{a}, \mathrm{y}), \mathrm{f}(\mathrm{x}, \mathrm{b})) = \{\mathrm{x} \mapsto \mathrm{a}, \mathrm{y} \mapsto \mathrm{b}\}$
None of the interpolation systems we have discussed can handle this case.
First order logic does have the interpolation property and a possible interpolant in this case is $\exists \mathrm{u} \cdot \forall \mathrm{y} \cdot \mathrm{f}(\mathrm{u}, \mathrm{y}) = \mathrm{c}$. It is easy to see that there is no universal interpolant for this example, and hence any interpolation system complete for FOL-E will have to go beyond universal CNF clauses.

Our completeness proof for the ground interpolation system is based on the completeness proof for superposition, which does extend to the non-ground case. The superposition completeness proof uses a lifting argument as follows:
Instead of looking at the sequence of clauses in a saturated set organized by $\prec$ as in the ground case, we look at the sequence of *all ground instances of clauses* from a saturated set organized by $\prec$. The lifting argument is that any ground

```
s :
   ... //Heap axioms
   //precondition
   assume x.f.g=y
   assume f≠g
   assume x.f∈P //set of locations
n₁ :
   x.f := null //h₁ = wr(h₀, x, f, null)
n₂ :
   assert ∃p∈P·p.g=y
   //negated as
   //∀p ∈ P · rd(h₁, p, g) ≠ y
   // the object pointed to by y is reachable from P
```

Figure 7.3: Scoping and heap reachability
`x.f` at the initial state is unreachable at the assertion,
which is equivalent to $\mathrm{rd}(h_0, x, f)$ having no equivalent ground term at n_2.
Without scoping we use the mixed term $\mathrm{rd}(h_1, \mathrm{rd}(h_0, x, f), g$ to instantiate the quantifier.

inference between the ground instances $C\sigma$ and $D\sigma$ whose existence the proof requires has to be a ground instance of an inference between C and D (the side condition that prevents superposition into a variable requires some care in the proof). This property fails for a colored proof when the required non-ground inference has a mixed color mgu and conclusion as above.

7.5.2 Completeness for some restricted cases

A simple condition to ensure completeness in this setting is to require that for each ground term $t \in \mathfrak{L}^t$ there is a ground term $s \in \mathfrak{L}^b$ s.t. $N \models s = t$ - this is (roughly) a variation of sufficient completeness w.r.t. simple instances from ([BW13]). As we have seen, this condition is too strong in our case for minimal scoping as e.g. in example 7.3 that we have already seen, we cannot refute an existential property of memory if a memory location becomes unreachable during the program execution. As mentioned before, we can remedy this by relaxing the scoping criteria as we have described in section 5.2 and keeping some more symbols in scope, but this is essentially a work-around. A solution to the problem will have to go beyond traditional superposition as there is not always a clausal universal interpolant.

We use the natural extension of our ground interpolation system for supporting scoping, and extend the scope as described above in order to maintain completeness. In practice, we have not found many cases that required inferences as above.

Chapter 8

Implementation and experimental results

In this section we present some experimental results and some of the more interesting implementation issues.

8.1 Order of evaluation

Our algorithm specifies which clauses need to be propagated and which derivations have to be performed at each CFG-node in order to ensure completeness, but does not specify in which order derivations and propagations are performed. In this section we discuss the effects the order of evaluation has on performance. We describe the experience we had with different orders and the choice we made for the algorithm.

There are two extremes for the order of evaluation, roughly in line with the difference between BFS and DFS (with regard to proof depth, not CFG structure): a BFS performs one step of one fragment at each CFG-node before performing the next step, while a DFS approach tries to discharge one assertion before moving on to the next. In general, we have found that neither of the extremes works well, and our approach is to search generally in BFS but to allow several cheaper fragments to saturate for each CFG-node before moving on to the next, including equality and heaps.

We give now some examples to the effects of evaluation order. The basic operations a CFG-node can perform are saturation w.r.t. a given fragment, a propagation request to its immediate predecessors (that can include requests from its immediate successors) or the propagation of the response of its direct predecessors. We mark these operations as $n_1.frag$, $n_1.req$ and $n_1.prop$, respectively.
We use *sup* for the fragment superposition, $req(x)$ to signify which term is requested and $prop(y)$ to signify which clauses are propagated.

The example program in figure 8.1 shows that servicing requests eagerly (essentially DFS) incurs a significant overhead in CFG traversal. In our example, one possible order of evaluation traverses the CFG in topological order, and services requests eagerly - we get the following order of operations (we assume scoping is enforced to limit some requests - e.g. n_3 never requests z as it is not

```
$n_0$: x := 0
$n_1$: y := 1
$n_2$: z := 2
if (*)
   $n_3$: assert y!=z //negated $y = z$
else
   $n_4$: assert x!=y //negated $x = y$
```

Figure 8.1: Example for order of evaluation
We consider the fragments of ground superposition
where $z \succ y \succ x \succ 2 \succ 1 \succ 0$

```
$n_0$: assume $P(b) \vee Q(c)$
$n_1$: assume $\neg P(a)$
$n_2$: assume $b = a$
...
$n_k$: assert $Q(c)$ //negated $\neg Q(c)$
```

Figure 8.2: Example for order of evaluation - simplifications
We assume $Q(c) \succ P(b)$ and $c \succ b \succ a$

in scope in n_2):
$n_0.sup - n_1.sup - n_2.sup - n_3.req(z) - n_2.prop(z = 2) - n_3.prop(z = 2) - n_3.sup$
now n_3 has derived $2 \neq y$.
$n_3.req(y) - n_2.req(y) - n_1.prop(y = 1) - n_2.prop(y = 1) - n_3.prop(y = 1) - n_3.sup$
Now n_3 has derived $2 \neq 1$ and is eliminated.
$n_4.req(y) - n_2.prop(y = 1) - n_4.prop(y = 1) - n_4.sup-$
$n_4.req(x) - n_3.req(x) - n_2.req(x) - n_1.req(x)-$
$n_0.prop(x = 0) - n_1.prop(x = 0) - n_2.prop(x = 0) - n_4.prop(x = 0) - n_4.sup$
Note that n_2 has cached the result for y and did not request it again from n_1.
We can see that CFG traversal can cause a lot of overhead even with caching of propagation results.

Another possible evaluation order that tries to reduce the overhead of CFG traversal traverses the CFG in reverse topological order only sending requests to predecessors, and then in topological order propagating responses and saturating. In the above example, the assertions are proven by two passes of request-propagate-saturate. While this evaluation order reduces the overhead of CFG traversal, it does not combine very efficiently with simplification derivations.

For example, consider the program in 8.2, here, n_k requests $Q(c)$ and the response $\{P(b) \vee Q(c)\}$ is propagated from n_0 through n_1,n_2, where it is simplified to $P(\mathbf{a}) \vee Q(c)$. If we follow the evaluation order of full CFG request-response cycles, we propagate this clause to n_k, perform resolution and then we need another request-response cycle (requesting $\neg P(a)$) to get a refutation.
However, if we allow eager servicing of requests, n_3 will request $Q(c)$, and when this request reaches n_2 the clause is simplified as above to $P(a) \vee Q(c)$, but now n_2 sends a request for $P(a)$ (both positive and negative) as it is a valid simplification literal. If the request is serviced immediately n_2 can simplify the clause to $Q(c)$ which is then propagated to n_k - we have reduced the number of CFG-traversals and also the size of propagated clauses. Similarly, as we have

```
$n_0$: assume f≠g
    $f \neq g$
$n_1$: x.f:=5
    $H_1 = H_0[x.f := 5]$
$n_2$: x.g:=6
    $H_2 = H_1[x.g := 6]$
$n_3$: y.f:=7
    $H_3 = H_2[y.f := 7]$
if (x=y)
    $n_4$: assert y.g==6
    $x = y$
    $H_3[y.g] \neq 6$
else
    $n_5$: assert y.g==old(y.g)
    $x \neq y$
    $H_3[y.g] \neq H_0[y.g]$
```

Figure 8.3: Example for order of evaluation - heaps
We use the DSA versions of the heap H_0, H_1, H_2, H_3
We use the fragment of UGFOLE plus the fragment of ground heaps
For superposition, the ordering is
$H_1 \succ g \succ| H_2 \succ y \succ| H_3 \succ H_0 \succ f \succ x$

seen in chapter 4, we can avoid the propagation of some clauses entirely if we check for subsumption when propagating.

Heaps

Another fragment that requires a different strategy is the heaps fragment, and, more generally, any fragment that relies on pattern based instantiation.

Consider the example in figure 8.3. Assume we use superposition, with the following heap axioms:
$\forall h, r, u, r', u' \cdot (r \neq r' \vee u \neq u') \Rightarrow \underline{h[r'.u' := v][r.u]} = h[r.u]$ and
$\forall h, r, u, v \cdot \underline{h[r.u := v][r.u]} = v$

n_1 derives the following clauses:
$\forall r, u \cdot r = \mathrm{x} \vee \underline{\mathrm{H}_1[r.u]} = \mathrm{H}_0[r.u]$
$\forall r, u \cdot u = \mathrm{f} \vee \underline{\mathrm{H}_1[r.u]} = \mathrm{H}_0[r.u]$
$\underline{\mathrm{H}_1[\mathrm{x.f}]} = 5$
Similarly for n_2:
$\forall r, u \cdot r = x \vee H_2[r.u] = \underline{H_1[r.u]}$
$\forall r, u \cdot u = g \vee H_2[r.u] = \underline{H_1[r.u]}$
$\underline{H_2[x.g]} = 6$
And, together with the clauses of n_1 (after simplifications):
$\forall r, u \cdot r = x \vee \underline{H_2[r.u]} = H_0[r.u]$
$\forall r, u \cdot u = f \vee u = g \vee \underline{H_2[r.u]} = H_0[r.u]$
$H_2[x.f] = 5$
At n_3 we get:
$\forall r, u \cdot r = y \vee H_3[r.u] = \underline{H_2[r.u]}$
$\forall r, u \cdot u = f \vee H_3[r.u] = \underline{H_2[r.u]}$

$H_3[y.f] = 7$
And together with the propagated clauses:
$\forall r, u \cdot r = \mathrm{y} \lor r = \mathrm{x} \lor \underline{\mathrm{H}_3[r.u]} = \mathrm{H}_0[r.u]$
$\forall r, u \cdot u = \mathrm{f} \lor u = \mathrm{g} \lor \underline{\mathrm{H}_3[r.u]} = \mathrm{H}_0[r.u]$
$\forall r, u \cdot r = \mathrm{x} \lor u = \mathrm{f} \lor \underline{\mathrm{H}_3[r.u]} = \mathrm{H}_0[r.u]$
$\mathrm{x} = \mathrm{y} \lor \underline{\mathrm{H}_3[\mathrm{x.f}]} = 6$
$\underline{\mathrm{H}_3[x.g]} = 6$

We can see that these derivations are not goal oriented - they are performed regardless of the assertions and entail the indexing and propagation of non-ground clauses. For example, the clause $\mathrm{H}_3[\mathrm{x.g}] = 6$ is not useful as the program never references the heap location x.g at any point after H_3 is defined. In contrast, our heap fragment instantiates heap axioms only on demand - when a heap-read term occurs at a later CFG-node.

For the above example:
n_4 requests $\mathrm{H}_3[\mathrm{y.g}], \mathrm{H}_0[\mathrm{y.g}]$, which causes n_3 to instantiate:
$\mathrm{H}_3[\mathrm{y.g}] = \mathrm{H}_2[\mathrm{y.g}]$
Now n_3 requests $\mathrm{H}_2[\mathrm{y.g}], \mathrm{H}_0[\mathrm{y.g}]$ and n_2 instantiates:
$\mathrm{x} = \mathrm{y} \lor \mathrm{H}_2[\mathrm{y.g}] = \mathrm{H}_1[\mathrm{y.g}]$
$\mathrm{x} \neq \mathrm{y} \lor \mathrm{H}_2[\mathrm{y.g}] = 6$
And then n_2 requests $\mathrm{H}_1[\mathrm{y.g}], \mathrm{H}_0[\mathrm{y.g}]$ and n_1 instantiates:
$\mathrm{H}_1[\mathrm{y.g}] = \mathrm{H}_0[\mathrm{y.g}]$
Now n_2 derives, by ground superposition:
$\mathrm{x} = \mathrm{y} \lor \mathrm{H}_2[\mathrm{y.g}] = \mathrm{H}_0[\mathrm{y.g}]$
And n_3 derives:
$\mathrm{x} = \mathrm{y} \lor \mathrm{H}_3[\mathrm{y.g}] = \mathrm{H}_0[\mathrm{y.g}]$
$\mathrm{x} \neq \mathrm{y} \lor \mathrm{H}_3[\mathrm{y.g}] =$

Note that without the above-mentioned requests, no instantiations occur, whether we use our heap domain or pattern based instantiation - e.g. the term $\mathrm{H}_2[\mathrm{y.f} := 7][\mathrm{y.g}]$ does not exist at n_3 and cannot exist there because of scoping. Note also the difference in complexity between the two approaches.

In terms of order of execution, it is easy to see that the above program can be verified in one backward-forward pass if the backward pass instantiates heap axioms and sends requests and the forward pass performs ground superposition (as implemented in our heap fragment). If we were to use only forward passes, even with eager request servicing, we would need four passes to prove the program, as each CFG-node requires instantiation in order to produce the requests for its predecessors.

The above example suggests two modifications to eager request servicing - the first modification instantiates heap-axioms when propagating a request backwards and the second modification performs ground superposition on the propagated clauses when propagating the response to a request. With both modifications the above program can be proved in one forward pass.

We have implemented a dedicated heap fragment and experimented both with a single backward-forward pass and with the variants of eager request servicing. In our experience, eager request servicing with instantiation on request-propagation greatly reduces the number of passes needed to prove a program, but the superposition step on response propagation is often expensive. Instead, we perform the forward pass that includes e.g. the ground equality fragment and a quantifier instantiation fragment with eager request servicing, and then perform one backward-forward pass of the entire CFG where the backward pass instantiates map axioms and propagates requests and the forward pass propagates responses and performs ground superposition only on equalities on map read terms.

Sub-DAG saturation: A more DFS oriented evaluation order we have experimented with saturates the CFG in topological order, but when saturating a goal node it repeats saturating the sub-CFG of the node's transitive predecessors (in topological order) until the whole sub-CFG is saturated, and only then moves on to the next CFG-node. In the program in figure 8.1 this means that we saturate the sub-CFG $\{n_0, n_1, n_2, n_3\}$ before we even look at n_4. In order to facilitate this order the traversal mechanism has to keep track of which CFG-nodes are not saturated, which is not trivial in cases such as the heap fragment above, where a request from a successor can enable more derivations. We have found that, while this order allows the elimination of many goals on the first pass and hence produces a simpler CFG, the cost of CFG traversal dominates the execution time and so the overall performance is much worse.

Our experience: We have experimented with several combinations of the above evaluation orders, where we saturate different combinations of fragments in different orders.

We have found that, in most cases, CFG traversal dominates the run-time and furthermore, the vast majority of requests have empty responses. The evidence we have seen for this is that, when we turn off the caching mechanism for any of the fragments, the algorithm grinds to a halt and usually does not finish even a single pass.

However, we have also found that sometimes servicing one simple request eagerly can allow a whole branch to be simplified or a goal node to be eliminated, rather than waiting until the whole CFG has performed one step. In light of the above we traverse the CFG in topological order and for each CFG-node collect all the requests for all fragments and service them eagerly, we continue by saturating and servicing eagerly only the simplification fragments, and propagate and cache an over-approximation of the requests, in order to reduce CFG traversal.

CFG-nodes responding to requests saturate the propagated clauses only w.r.t. simplifications (for which eager request servicing is applied). Each CFG-node remembers all the previous requests and if a new clause is derived anywhere that matches a previous request it is propagated in the next forward pass. We apply one forward pass of instantiation or superposition (the more expensive fragments) followed by a pass of the cheaper fragments and then a garbage collection pass. A pass of garbage collection performs mark-sweep in reverse topological order where each CFG-node marks for its predecessor which of the previous requests is still relevant - this allows us to eliminate lemmas that were produced for a goal node that has been eliminated and not used by any other CFG-node.

8.2 Implementation

We have implemented the verification algorithm in C#. We have implemented the fragments of UGFOLE, GFOLE, simple instantiations by E-matching (with a very simple and partial inference of patterns), heaps and, partially, superposition. We have also implemented some simple and incomplete integer handling (mostly by simplifying terms). We have implemented scoping and depth and width bounds at the construction level (that is, for all EC-graph operations and for unification and E-unification). We have not implemented interpolation but we calculate the ordering for superposition (used also for heaps) as described for interpolation and hence, while not complete, superposition is geared more towards eliminating terms that are not in scope in successors. We have also used the ordering for superposition as a hint when inferring patterns for instantiation, by preferring maximal (in $\succ$) terms as patterns.

Pre-processing

Our implementation takes a Boogie program after it has been simplified by Boogie to include only `assume` and `assert` statements (not in CNF form, and the CFG has non-binary joins, but no loops) and then performs several transformations and simplifications to bring the program to our required shape - including:

- Type grounding - Boogie uses polymorphic types with type quantifiers, but in our experience it is often the case that the relevant type instances can be determined a-priori, so we instantiate only the relevant type-instances of axioms.
- Function inlining and elimination of unused or useless declarations.
- Branch determinization - by adding a branch condition propositional literal for non-deterministic branches.
- Extraction of Boolean functions, similar to [KKRV16] (Boogie supports Boolean typed expressions in any position, as in SMT-lib, while standard FOL supports Boolean functions only under the predefined Boolean connectives).
- CFG transformation to enforce a well branching program (binary branches and matching joins) and also several CFG simplifications
- CFG unit propagation - a simple analysis that treats each literal as its (unique) string representation, and performs unit-propagation along the CFG, calculating the set of unit clauses that hold at each CFG-node and simplifying clauses accordingly - this simple analysis can often remove up to half of the assertions. While the assertions removed are mostly simple ones, removing them allows us to simplify the CFG.
- Scope calculations - we take the initial set of clauses at each CFG-node and calculate the minimal scope needed to ensure completeness even with quantifiers, as described in chapter 5.2.
- DSA map calculations - for Boogie maps (often used to represent heaps) we track the DSA chain for each map typed term that occurs in the program all the way to the root-node. We use the unit clauses at each node to find heap updates and also detect CFG-nodes where it is likely that a frame rule has been applied (that is, when a DSA chain is broken for a map but there is another map of the same type that has a DSA chain), we then detect possible frame axioms (that equate two map-reads of the same type

with free variables) and optionally instantiate the potential frame rule for the relevant pair of maps if found. We can also optionally track each map-read in the initial program through its DSA chain and add the relevant terms that form its DSA chain, in order to reduce CFG-traversals later.

Main algorithm

The data structure at each CFG-node is based on an EC-graph (as in chapter 3) and a set of clause-ECs (as in section A.2) over the nodes of the EC-graph. In order to enable the sharing of EC-nodes between different CFG-nodes, each EC-node only includes references to EC-nodes in its sub-graph (that is, only for sub-terms), and any references for super-terms (such as set of super-terms with a given function) is kept per CFG-node. For example, if $a = b$ holds in two consecutive CFG-nodes n_1, n_2 and both contain the EC-node for the terms $f(a), g(f(a))$ but at n_2 also $g(f(a)) = c$ holds, they will share the EC-sub-graph $[a, b], [f(a), f(b)]$ but not the EC-nodes for $g(f(a)), c$.
Each EC-node caches information such as depth, tkbo ordering information and transitive sub-terms.
We keep a central repository of all EC-nodes to ensure that there is exactly one instance of each EC-sub-graph.

Equality: The handling of equality is by the algorithm from chapter 3, which interleaves congruence and transitive closure with equality propagation. This algorithm is used to add term-ECs to the EC-graph, propagate term-ECs from predecessors, synchronize with a predecessor after it had been updated and adding new equalities (e.g. derived by other fragments).

The algorithm, as described, does not ensure a unique object (meaning the same EC-sub-graph is represented by the same object in all EC-nodes in which it appears). The algorithm works on a temporary copy of the sub-EC-graph that is modified, and once it is done (`mainLoop` terminates), a unique representation is found for each modified EC-node.

The search for unique objects for EC-nodes is performed by first splitting the simplified EC-graph to strongly connected components (SCCs) (with the GFA edges directed to sub-terms) and then processing them in topological order (Each not-modified EC-node is included in the first connected component).
For each component, we define a total order on its EC-nodes and number them accordingly, and then produce a unique string that represents the SCC - we keep a map of these strings for all existing SCCs in the system. For example, the simplified EC-graph $\{[\mathrm{f(a), g(h(b))}], [\mathrm{h(b)}]\}$ which references the unmodified EC-nodes $\{[\mathrm{a}], [\mathrm{b}]\}$ has one SCC that includes the entire graph. In order to find an order between the ECs $[\mathrm{f(a), g(h(f(a)))}], [\mathrm{h(f(a))}]$, we first select the GFAs that only reference EC-nodes from previous SCCs - in this case this is only f(a). On this GFA we define a total order as we have a total order on existing EC-nodes by order of creation, which always respects the topological order on SCCs. We order the ECs with such GFAs by their minimal GFA and number them accordingly - so $[\mathrm{f(a), g(h(b))}]$ is assigned the number 0. We proceed by ordering the next set of GFAs that reference only unmodified *and numbered* ECs - in our case the GFA h(f(a))assigns the number 1 to $[\mathrm{h(f(a))}]$. We continue assigning numbers and ordering GFAs until all ECs in the SCC are assigned a number. We are guaranteed to assign numbers to all ECs as each EC is guaranteed to represent at least one term (we do not search for a unique representative for

the empty GTs - they are only used as witnesses for completeness). We are now ready to produce our unique string - we represent the ECs in order and within each EC the GFAs in order. For each unmodifed EC-node we have a pre-calculated unique string (we use an integer index) for each new EC we use the numbers - so in our case the string is `[f(s`$_{[a]}$`),g(1)],[h(0)]` where `s`$_{[a]}$ is the pre-calculated string for the unmodified EC [a].

Clauses: Clauses are represented as clause-ECs over the EC-graph of the EC-node in which they occur (so a clause may have more than one instance if it occurs in two CFG-nodes with a different EC-graph). Clauses are simplified eagerly (e.g. unit clauses are propagated eagerly, duplicate literals are eliminated and a clause with opposing literals is eliminated). Clause-ECs are updated only once the congruence closure is saturated for the EC-graph and are indexed (for instantiation or superposition) only before a pass that saturates the relevant fragment - so a clause can be simplified or even eliminated before it is indexed.

Garbage collection

As a merged EC-nodes is replaced by a new node, and as many clauses and literals are discarded by the simplification rules, the algorithm tends to create a lot of garbage. In addition, each CFG-node maintains some caches for requests from transitive successors, which are used to propagate new clauses that match previous requests without needing to re-issue the request. When some of these successors are simplified or eliminated, some of these requests become irrelevant (e.g. if we had the clause $\mathrm{C} \vee \underline{\mathrm{b}} = \mathrm{a}$ and requested b, and later we derived C and the clause is subsumed). We perform a mark-sweep pass that traverses the CFG in reverse topological order and removes unnecessary elements in each CFG-node, marking the still relevant requests in direct predecessors. We also mark all used term- and clause-ECs and discard those that are no longer used. We have experimented also with reference counting but the overhead is quite significant and support for cyclic EC-graphs is only partial.

8.3 Experimental results

As we did not implement support for rationals and only a limited support for integers, and as our implementation is written more as a test-bed for experiments than as an optimized verifier, we do not expect to be competitive with mature tools, such as Z3, for smaller examples, as optimizations and especially the optimization of the pre-processing stages dominates performance on such examples. Hence, we looked for examples that take Boogie+Z3 long time to prove, and that are not arithmetic intensive - which meant, mostly, heap intensive programs. We also tried to collect examples from different front-ends to Boogie as they tend to generate VCs of different styles.

We have collected Boogie programs from Dafny, Chalice, VCC. We show results only for the long running examples. For Chalice, we have written an implementation of an AVL tree with all the standard operations and full specification, where the elements of the tree are modeled as a sequence typed ghost field. Dafny and VCC programs tend to use set axioms. We have also experimented with some of the long-running VCs used in the thesis [Rud11], which encode a *stereotype*, which is a collection of reusable specifications for heap-manipulating data structures.

Experiments

As we have not implemented a sufficient integer fragment, and as most interesting programs include at least some integer reasoning, our tool cannot, in itself, verify many interesting programs. Instead, we run our tool on the Boogie program and, if we cannot completely verify the program, we dump an SMTLib version of the state at various stages of the algorithm and compare the run-time of an SMT solver on the original Boogie SMTLib output and the output from our tool (with the run-time of the tool up to that point added).

We have experimented with different orders for applying the various fragments and also with different values for the following parameters:

- Maximal term depth
- Maximal clause width (number of literals)
- For heaps: whether to perform ground superposition on instances of map axioms eagerly, or only create the instances
- For superposition: whether to derive only ground clauses

SMT solver

For programs that our tool could not handle on its own, we have used Z3 as the SMT solver back-end for the remaining (simplified) proof obligations. The logic needed for most interesting examples (examples that take a significant time for Boogie to prove) is at least UFLIA (quantifiers, uninterpreted functions and linear integer arithmetic). According to SMTCOMP (e.g. [CDW14]), the only provers that support this logic are CVC4 ([BCD+11]), CVC3, Z3 ([dMB08]), veriT ([Sch09]) and recently Vampire_smt (apparently based on [Vor14]). We have tried running the Boogie output, and the output of our tool with CVC4, CVC3, Z3, veriT and also with Beagle ([BBW15]) as it is a superposition based prover that also supports some numerical reasoning using hierarchical superposition and supports SMTLib natively.

Of the above provers, only Z3 was able to handle any of the interesting problems (with a run-time of more than a few seconds). We believe the reason is as follows: Boogie itself was built with Z3 in mind and uses in its output some Z3 specific attributes - specifically, it uses weights for quantifiers in axioms to direct the relative depth of instantiation. A common problem for pattern-based quantifier-instantiation is that of matching loops - for example, an axiom $\forall x \cdot P(x) \Rightarrow P(f(x))$ with the pattern $P(x)$, combined with the axiom $P(c)$ will produce an infinite number of instances $\{P(f^n(c)) \mid n \in \mathbb{N}\}$. A matching loop can also involve several axioms. While some of the cases of matching loops can be detected by the SMT solver, there is no way to find a maximal instantiation depth for such axioms (As FOL and FOLE are only semi-decidable), hence some prioritization of instances is needed. SMT solvers sometimes use a heuristic that tries to direct quantifier instantiation away from such matching loops, assigning to each term-EC a measure that roughly describes how many quantifier instantiations were needed to produce that term. This measure is called the weight or depth or generation of a term (there are different strategies and measures used in different provers - e.g. the depth of [DNS05]). The measure often also depends on other parts of the SMT-solver, to control the mix between decisions and quantifier instantiation. For our example, such a measure could be 0 for $P(c)$, 1 for $P(f(c))$, 2 for $P(f(f(c)))$ and so on. Prioritizing quantifier instantiation to the instances where the pattern is matched by a lower measure allows fairer instantiation of axioms and avoiding generating too many instances of axioms with such matching loops (some axioms can also produce a finite but exponential number of instances (e.g. using permutations), which some of the measures can also help to avoid).

For a given path in the Boogie VC of a program, the minimal instantiation depth needed to prove an assertion at the end of the path that contains a heap read can be up to the number of heap updates on the path - in some of our examples, this is in excess of a hundred heap-updates per-path (especially Chalice uses a shadow permission heap which is updated often). Allowing other axioms to instantiate to such a depth will make most solvers extremely inefficient.

Z3 allows annotating each quantifier with a weight that is used as a coefficient for determining the measure of terms introduced in quantification, as a function of the measure for terms used in instantiation. Boogie assigns a weight of zero to the map read-over-write axioms, allowing them to instantiate to any depth, because heap updates that are generated from Boogie heap-writes are guaranteed to be oriented to the CFG (that is, a DSA version of the heap can only be an update of an earlier DSA version), except for some very unusual cases. Without the instantiation measure, an SMT solver has little chance to guess that a specific quantified axiom has to instantiated much more than others. We have examined the number of axiom instances used by Z3 in the proof search and, as expected, it is dominated by map axioms for most heap-manipulating examples (sequence axioms also rank high in some examples).

We have tried some examples with Z3 with the weight annotation removed and Z3 could not prove any but trivial examples. CVC4 supports several quantifier instantiation strategies in addition to using patterns, including model-based quantifier instantiation ([GdM09]) and conflict guided quantifier instantiation ([RTdM14b]). We tried running several versions of CVC4 with different quantifier instantiation strategies, but none could prove any of the non-trivial examples. CVC4 is leading the UFLIA and UFNIA categories in the latest SMT-COMPs and hence we expect that it could handle many of the examples, but this would require further work with the generation of the SMTLib output.

A possible solution is to encode heaps using the native array theory of SMTLib. The problem is that Boogie uses polymorphic maps with dependent types - the type returned by a heap read depends on the type of the field. Boogie uses an encoding that adds type parameters as terms and uses type coercion functions (as described in [LR10]). Boogie supported using a native array theory of Z3, but this support is deprecated.

However, for some examples that CVC4 could not prove for the reasons above, we found that running CVC4 on the output of our tool allowed finding a refutation. This was the case only for simpler examples, and we suspect the reason is that our tool derived most of the necessary instances of heap axioms. In addition, the CVC4 performance on such example was better when we ran our tool for more passes, up to a point.

In light of the above, our results are biased towards the particulars of Z3.

Encoding the SMTLib output

We have found that the encoding of the SMTLib output has a very significant effect on the performance of Z3, which is to be expected as Z3 uses E-matching and lazy CNF conversion. We kept our encoding as similar to the Boogie encoding as possible, where the only significant parameter is which part of the information calculated by the tool is transferred to the SMT solver.

Our minimal encoding includes the simplified CFG (with any node shown infeasible removed). We use the pre-CNF-conversion form of of the formulae describing the VC and only simplify them using our derived information, as eager CNF conversion has a significant negative effect on the performance of Z3. We use the EC-graph and the calculated unit clauses at each CFG-node to simplify the formula at each CFG-node. Simplification is by rewriting terms and literals to normal forms. Normal forms for terms are the minimal representative of an EC in the EC-graph of the relevant CFG-node as per the superposition ordering - essentially if we have derived the fact that a certain ground term equals an integer or Boolean constant at some program point, any occurrence of the term at a formula at that program point is replaced by the constant, and the equality between the term and the constant is added to the set of formulae at that program point. After rewriting terms to normal form by the EC-graph we use the derived unit clauses at a CFG-node to simplify the original formula at the CFG-node, apply propositional simplifications and eliminate any formula that evaluates to true. We do not use non-unit clauses as they may contain literals added for CNF conversion, which affect Z3 performance badly. In theory, one could reverse the CNF conversion (at least for ground clauses) as we know which literals were introduced during conversion, and this remains as future work.

We have also experimented with an encoding that does not apply the above simplifications, and an additional encoding that adds equality information for all map reads (hence essentially encoding our interpretation of the map axioms) and a further encoding that includes all of our derived information (although in CNF form). The full encoding with all derived information always performs very badly - we believe that the reason is a combination of the CNF conversion and the introduction of new terms, which affect pattern based quantifier instantiation.

Configuration

The experiments were run on an Intel Core i7 2600K quad core processor with 16GB of memory, running Windows 7 64bit. All experiments use only a single core.

The first set of experiments (run as a Boogie optimizer) was run twice, with some of the experiments performed several times. We have found that the runtime of our tool is very consistent (for longer running examples) and so the other experiments were performed only once.

Use as a Boogie optimizer

The overall performance of the tool when run as a pre-process for Boogie is shown in table 8.1. We used our tool running one pass of saturating the UG-

FOLE fragment and instantiation, including the heaps fragment but without ground superposition on map clauses. We only show results for long running examples that verify in Z3. We run all examples with a time-out of 600 seconds as this captures the Boogie run-time of most examples with some margin. Most results show an improvement in the total run-time. The Dafny examples are somewhat problematic, as explained below. We note that, on one hand, we only included the simpler fragments our tool can handle, as the implementation of the more advanced fragments is not yet competitive, but on the other hand, our tool, pre-processor and back-end are not highly optimized.

Encoding of the SMTLib output: As noted above, our SMTLib back-end encodes the VC quite similar to Boogie - we use nested `let` expressions to encode control flow, and use unit clauses derived by our algorithm to simplify axioms. We do not use the CNF clauses as CNF conversion is harmful to Z3 performance.

For examples originating in Dafny, our SMTLib back-end faces a problem. Our pre-processor removes type quantification by enumerating the possible types for each quantifier, allowing us to use typed versions of the axioms rather than using type coercion as done by Boogie. The problem is that the frame axioms of Dafny are encoded using type quantification in order to apply to heap accesses of all field types, but the type variable is not part of the pattern for the frame axiom, and hence in the output of our back-end each frame axiom that is instantiated once in the Boogie output is instantiated several times in our output (in fact, in all the Dafny examples, there are five instances for five different field types). We have found that, for the Boogie output of all the Dafny examples we have inspected, the frame axioms are instantiated many times (up to millions of times) even when there are very few distinct instances - evidently Z3 fails to learn clauses constructed with instances of this axiom. The direct outcome is that our pre-processor amplifies this problem by eliminating type quantification - the output of the back-end of our tool, even just after the pre-processor, performs much worse in Z3 simply because many more (non distinct) instances of the frame axiom are generated. We have been able to mitigate this effect somewhat by removing one axiom of the Dafny preamble - this axiom encodes the transitivity of heap succession - the transitivity of the `$HeapSucc(a, b)` predicate which is used on the left hand side of all frame axioms. This axiom was not needed by any of the examples we have inspected, and has a strong negative effect on Z3 performance regardless of our tool (the original VC for the `AVLTree` for Dafny took almost 3000 seconds to prove with the axiom, and around 400 seconds without). However, for the `TreeBarrier` example, we still suffer from the elimination of type quantification. It is theoretically possible to reverse this elimination in the back-end, at least for the original axioms, but we have not implemented this.

Scoping

In order to quantify the effects of scoping on performance and completeness, we run several of the larger benchmarks both with and without scoping. We run each benchmark for four passes, in the order equalities, heaps, instantiation, heaps. In most cases, the number of assertions proven per pass was the same. The results are shown in table 8.2. The effects of scoping are almost universally beneficial, and the effect on completeness (in the limited fragment which we

	time(s)			
VC	Boogie	Optimizer	Z3	%
Chalice				
AVLTree.insert	29	9	16	86%
AVLTree.pruneMax	31	6	18	77%
AVLTree.rebalanceLR	273	21	233	93%
AVLTree.rebalanceRL	TO	21	207	
AVLTree.remove	62	17	36	85%
Dafny				
AVLTree.InsertRecursion	405	17	TO	
AVLTreeGF.InsertRecursion	79	18	41	74%
B4.Map.Remove	12	3	1	33%
TreeBarrier.Node.barrier	11	10	40	454%
Stereotypes				
PIP.AddToEnd	11	1	10	100%
PIP.Node.acquire	64	3	35	59%

Table 8.1: Results of running the verifier as an optimization pre-process
Base stands for the baseline Boogie performance with Z3 as a backend. The Opt column is the time taken for our tool run as an optimizer, the Z3 column is the time taken for Z3 on the optimized output of our tool and the last column is percentage of time taken by our tool and Z3 from the baseline time of only Z3. TO means timeout - more than 600 seconds.

used) seems less significant.

Term depth restriction

In order to quantify the effects of depth restrictions, we run our tool on VCs with many assertions that do not require arithmetic (the examples we have found come from Dafny, Chalice and VCC - other tools seem to produce less assertions per VC) and compared the number of assertions proven for each term depth bound. We did not count the assertion proven in our pre-processing steps, as they are not affected by the term depth bound. The results are shown in table 8.3. While we do not have many samples for which we could quantify the effect of term-depth, the results do hint that a gradual increase of term depth can be useful for proving a subset of the assertions, even for programs we cannot completely prove. In addition, for the vast majority of the benchmarks, running with a very low term depth (1 or 2), the tool terminated in a short time, regardless of whether it diverged for higher depths. A use case for the tool would be running very low term depth without a time-out and then attempting to verify with a higher term-depth with a time-out. Marking each proven assertion with the depth needed to prove it can help the user to understand how changes in the program affect the proof complexity for assertions.

While we need to implement arithmetic in the tool in order to get full results, and more samples are needed, the above results, and some inspection of not proven assertions, suggest that program proofs often do not need very large terms. Running the tool with increasing term depth and a time-out might produce more useful results than running a monolithic VC with the same time-out, if not all assertions are proven.

Additional experiments

We have also experimented with varying the maximal clause width, but experiments showed little difference even with no width bound - only very few

		time(s)				#proven	Δ
VC	**pass #**	1	2	3	4		
Chalice							
AVLTree.rebalanceLR	scoping	24	81	92	151	9	
	no scoping	27	TO			1	
AVLTree.insert	scoping	10	20	21	31	24	75%
	no scoping	11	57	62	122	24	
FactorialActor	scoping	41	45	65	80	80	
	no scoping	44	TO			1	
Dafny							
BFS	scoping	6	7	7	8	9	
	no scoping	TO				0	
B4	scoping	3	3	3	4	4	0%
	no scoping	3	3	4	4	4	
AVLTree.InsertRecursion	scoping	18	26	31	39	105	20%
	no scoping	21	31	39	49	105	
VCC							
vscomp2010_p1_v2.sum_max	scoping	8	9	10	11	4	0%
	no scoping	8	10	10	11	4	
vscomp2010_p3_v0.find	scoping	9	10	10	11	2	9%
	no scoping	9	10	11	12	2	
VCC3.RBTree.left_rotate	scoping	21	30	33	38	7	15%
	no scoping	22	35	39	45	7	
VCC2.RBTree.tree_insert	scoping	114	131	138	149	23	30%
	no scoping	133	167	184	211	23	
VCC2.RBTree.right_rotate	scoping	23	34	37	49	38	36%
	no scoping	29	49	59	77	39	

Table 8.2: The effect of scoping on the run time and number of assertions proven.
We show the total time taken for the first n passes.
The last column shows the improvement for four passes - empty when our tool timed out without scoping.
The `tree_insert` took about 75 seconds in the pre-processing stage that modifies the CFG-node scope so that a total order can be defined, because of an inefficiency in the implementation which we believe is fixable.

		depth				
VC		1	2	3	4	5
Chalice						
AVLTreeNode.init	time(s)	4	6	10	12	19
	#proven	7	23	38	38	38
AVLTree.insert	time(s)	3	10	18	27	35
	#proven	1	6	20	20	20
FactorialActor	time(s)	21	65	91	196	419
	#proven	0	25	77	99	111
Dafny						
BFS	time(s)	7	10	11	11	11
	#proven	2	6	7	7	7
AVLTree.InsertRecursion	time(s)	26	51	149	TO	
	#proven	74	98	99		
VCC						
VCC2RedBlackTree.left_rotate	time(s)	7	24	33	64	147
	#proven	17	36	38	39	39
VCC2RedBlackTree.tree_insert	time(s)	17	113	142	148	200
	#proven	6	23	24	24	24

Table 8.3: The effect of term depth bounds on run-time and number of assertions proven.

examples derived very wide clauses and they were often not needed for the proof. We did encounter a few examples that diverged after applying several superposition passes, when clause width was not restricted.

Our tool reports the ratio of sharing for GTs and GFAs and also clause- and atom-ECs. The sharing ratio for GTs and GFAs (the number of CFG-nodes where the GT resp. GFA occurs) was consistently between 4 and 6 for the vast majority of examples. For atom-ECs it was lower and for clause-ECs yet lower. This result supports the design of our EC-graph data structure, but does not support the additional overhead of atom- and clause-ECs.

Chapter 9

Conclusion and future work

In this thesis we have proposed a novel technique for the tighter integration of a program verifier and a theorem prover. Our aim is an incremental and reliable verifier - a tool that a programmer can use with predictable results for simpler programs and properties, but can also be extended to more complicated programs.

9.1 Main contributions

We have developed a technique that allows incremental verification of programs, applying logical reasoning in increasing strength and providing useful intermediate results. Our technique combines ideas from verification, static analysis and compilation. Using our technique we can reduce the search space of the prover using knowledge from the program and simplify the program using knowledge from the prover. A global view of the program allows us to set parameters to the prover, such as the scope, ordering and relative depth of terms, which helps direct the prover towards more efficient proof search.

We have instantiated our technique for several logical fragments, including ground equalities, superposition with bounded size and a simple heap analysis. We have shown how scoping, taken from programming languages, can be used in the verification context of the same programs to guide the prover in the search for a proof.

For the fragment of ground equalities we have shown an incremental algorithm and data structure that can deduce equality information on-demand. The algorithm can deduce equality information at program locations that is sensitive both to earlier and later parts of the program, which cannot be done by analyzers that perform a single forward or backward pass. The data structure helps derive information at specific program points lazily, and reduce the duplication of effort during the proof search. Our algorithm supports the enforcing of size and scope restrictions on construction, ensuring reliable performance.

For superposition, we have shown a complete way to use superposition for verification while avoiding many inferences that are performed with VCG, using information about the structure of the program. We have shown how to use joins to reduce the proof size and how to program structure can be used to enhance redundancy elimination. We have shown how to define a total order on ground terms that is oriented for a DAG and allows interpolation in DAG shaped programs, although we have not implemented the interpolation algorithm. We have also shown that using bounds on the size of terms can tame superposition and allow an incomplete search for smaller proofs, more so than clause size.

9.2 Experience and directions for future work

We have found that our technique works relatively well for simpler logical fragments, especially the equality and heap fragments, and that the exchange of information between the verifier part and the prover part is beneficial for both. In addition, we have seen evidence that the size bounds on proof elements, and especially terms, is an effective tool for building logical fragments of increasing strength.

As our implementation is not complete, and especially lacks integer reasoning, we have inspected many assertions that our tool could not prove to see which part of the reasoning was missing. We have not found many cases where the missing part of the proof was complicated or large, not in integer reasoning and not in the propositional part.

On the other hand, we have found that superposition, even tamed by our technique, has some weaknesses that make it less efficient in its current form for verification. The main culprit, in our experience, is the propositional part of the proof. Although much smaller in our setting than in the more common VCGs, some inferences produce many propositional variants of very similar clauses. In addition, even though the ordering we produce is predictable for most constants, there are still some arbitrary choices in determining the ordering that affect the proof depth of some assertions, making verification in bounded fragments less predictable to users.

One important direction for future work is better handling for the propositional part of the proof. A large part of the proofs for all the programs we have seen is ground and propositional. Currently the more successful provers for verification are based on SMT technology, which is very effective on ground problems but is less predictable for unbounded problems. Superposition is based on ordered resolution, which is known to be less efficient for propositional proofs. An alternative that satisfies our requirements is Stålmarck's method ([SS98]). The main attraction of the method is that it applies aggressive simplification and branches, but it performs joins on branches, so that a hierarchy of fragments of predictable strength can be defined, using the depth of branching, and in addition, the state needed at any time during the algorithm is proportional to the input size (for ground problems). Adapting this method to our setting, and ensuring that all and only necessary information is propagated to the prover at each CFG-node, is interesting future work.

Another important direction is parallelization. The verification can be split so that different threads process different program points, but also between fragments - we can let each thread perform inferences in a different fragment, sharing the verification state before the split, and only adding their conclusions at specific synchronization points. For parallelization to work efficiently, the data structures will have to be developed to allow answering queries for multiple threads, while avoiding redundant work, and accepting updates from threads only at certain points in time.

A third interesting direction is to use our technique for the inference of invariants using loop unrolling, as in e.g. [McM03b]. Once we implement superposition fully, and either our interpolation calculus or one of the alternatives ([BW13], [McM08]), our technique can be adapted to generate invariants using interpolation incrementally. Our technique allows adding program extensions at any time during the algorithm, which allows unrolling a loop incrementally. In addition, our ordering is suitable for generating invariants as all loop-body internal symbols go out of scope after the loop body - the only adjustment needed is that branch conditions for branches within the loop need to be in scope only up to the end of the loop body.

An additional possible line of inquiry is an improvement to the interpolation mechanism. Rather than make earlier DSA versions larger, which causes superposition to be somewhat less goal directed, we could make earlier DSA versions smaller. When the prover at some CFG-node tries to unify two maximal terms, if the terms only disagree on pairs of terms that are in scope in the predecessor, we can send these pairs as proof obligations to the predecessor. Each such conjunction of equalities is represented by its negation at the predecessor - a disjunction of dis-equalities. If this clause can be used to derive the empty clause at the predecessor, the predecessor can remember the derived equalities and propagate them to the successor. The predecessor may have to send such requests to the successor as well for interface terms. Such a system would be nearer to IC3/PDR in the way the information is exchanged, but still operates at the syntactic level and does not require the application of an SMT solver.

Appendix A

Appendix

A.1 Interpolation

A.1.1 Completeness proof for GFOLE interpolation

We show completeness for this calculus using the standard model generation technique, adjusted to our setting.
The presentation follows mostly the proof in [BG91] with some modifications.

Definition A.1.1. Set of clauses smaller than C - S_C
For a set of clauses S and a clause C (not necessarily in S), we use:
$S_C \triangleq \{D \in S \mid D \prec C\}$

Definition A.1.2. Redundant clause $\mathbf{red}(C, S)$
A clause C is redundant in S (C not necessarily in S) iff:
$\mathbf{red}(C, S) \overset{\triangle}{\Leftrightarrow} S_C \models C$
That is, the clause implied by a set of smaller clauses in S.

Definition A.1.3. Derivability in one step $S \vdash^1_I C$
For a calculus I and a set of clauses S we write $S \vdash^1_I C$ if there is an instance of a rule in I s.t. all the premises are in S and the conclusion is C.

Definition A.1.4. Saturation up to redundancy
We say that a set of clauses S is saturated up to redundancy by a calculus I iff:
For every derivation in I with premises in S, where C is the conclusion, either $S_C \models C$ or $C \in S$
A set of clauses S is saturated up to redundancy w.r.t. a calculus I if
$\forall C \cdot (S \vdash^1_I C \Rightarrow (C \in S \vee \mathbf{red}(C, S)))$

We assume from now on:
(A1) N is a coloured set of clauses saturated up to redundancy w.r.t. SP_I.
Hence $N = N^b \cup N^t$.
We now give two lemmas that allow us to reason about the communication between N^b and N^t.
Our algorithm is parametric in $=_\Gamma$and in **idas**, and the only property of **idas** used in the completeness proof is the lemma below.
Hence, any other definition of **idas** that satisfies the below lemma will give us a complete algorithm. Our unit algorithm essentially uses $=_\Gamma$that is equivalent to $\{(s, t) \mid N^b \models s = t\}$, while the formulation we have given here is much less precise. We discuss in the future work section some other possibilities.

Lemma A.1.1. Completeness of equality theory
If
(a1) $s, t \trianglelefteq \mathrm{maxTerms}(N^t)$
(a2) $s, t \succeq l^t$
(a3) $N^b \models s = t$
Then there is a clause B s.t.
(c1) $s =_\Gamma t$
(c2) $B = \mathbf{idas}_\Gamma(s, t)$
(c3) $B \in \mathfrak{L}^I$
(c4) $N^b \models \neg B$
The proof is straightforward from the definition of $=_\Gamma$and **idasg**.

Lemma A.1.2. Soundness of the equality theory
If
(a1) $B = \mathbf{idas}_\Gamma(s,t)$
Then
(c1) $(N^b \models B) \vee (N^b \models s = t)$
The proof is straightforward

Theorem A.1.3. Soundness of SP_I
For all S,C,I if
$S \vdash_{SP_I} C$
Then
$I \models S \Rightarrow I \models C$
The proof is straightforward using lemma A.1.2.

Model generation

We will show that either $\square \in N$ or N has a model, that we will construct.
We construct the model similarly to the standard model, except that we build separately the model for N^b using the standard method (as the standard calculus applies there) and on top of it the model for N^t.

Models, structures and interpretations

In our context we only discuss models of the following form:
Each model is characterized by a congruence relation $\cong$.
The universe (domain) of the model is the quotient of the Herbrand universe and the congruence - $\mathbf{T}(\Sigma)/\cong$, so that each element of the universe is an equivalence class of terms.
We use the following interpretation I for all models we consider, and hence discuss only models from here on:
Each term t maps to its equivalence class by $\cong$:
$[\![t]\!]_I \triangleq [t]_\cong$.
The interpretation for functions is standard:
$[\![f]\!]_I([\![\bar{s}]\!]_I) \triangleq [\![f(\bar{s})]\!]_I$.
As $\cong$ is a congruence this is well defined.
We only refer to the model from here on and assume the above interpretation.

Term rewrite systems(TRS) and normal forms

A term rewrite system (TRS) is a set of rules of the form $u \to v$ where u,v are ground terms.
A derivation in a TRS T is as follows: if $u \to v \in T$ then $s[u]_p \to_T s[v]_p$ for any term s and p a position in s.
$\to_T^*$ is the transitive reflexive closure of $\to_T$.
We define derivability in up to n steps recursively as follows:
$\to_T^0 \triangleq id$
$\to_T^{n+1} \triangleq \to_T^n \circ \to_T$.
A TRS is terminating if it only has finite derivation sequences.
A term t is reducible by a TRS T iff there is some s different from t s.t. $t \to_T s$, and otherwise it is irreducible by T.
A term t is in normal form w.r.t. T if it is irreducible by T - in general a term can have more than one normal form, or none at all.
We use $\mathbf{irred_T}$ to denote the set of irreducible terms in T.
All the TRSs we discuss are oriented w.r.t. the ordering $\succ$, meaning that

$\forall u \rightarrow v \in T \cdot u \succ v$.
A well known result is that a TRS oriented w.r.t. a well-founded ordering is terminating - because the ordering is strict there are no derivation loops, and because the ordering is well-founded every derivation sequence terminates in a finite number of steps.
A TRS T is confluent iff
$\forall s \cdot s \rightarrow_{T}^{*} t \wedge s \rightarrow_{T}^{*} u \Rightarrow \exists v \cdot t \rightarrow_{T}^{*} v \wedge u \rightarrow_{T}^{*} v$.
A TRS T is convergent iff it is terminating and confluent.
In a convergent TRS T each term has a unique normal form - we will use the function r^{T} that returns the normal form of a term in T when we know T is convergent.
A TRS T is left reduced if no derivation applies to the left-hand side of any rule - formally
$\forall s \rightarrow t, u \rightarrow v \in T \cdot s \ntrianglelefteq u$.
It is a well known result that a left-reduced TRS is confluent (as it is locally confluent) - hence any oriented, left-reduced TRS is convergent.
Each convergent TRS T defines a congruence relation $=_{T}$ as follows:
$s =_{T} t \overset{\triangle}{\Leftrightarrow} r^{T}(s) = r^{T}(t)$ - that is, two terms are congruent iff they have the same normal form.
Hence each convergent TRS defines a structure with the standard interpretation.
We will use the notation
$T \models s = t$ to denote $s =_{T} t$.
A property of convergent TRSs is that, for each convergent T and terms s,u where $u = s|_{p}$:
$r^{T}(s\left[r^{T}(u)\right]_{p}) = r^{T}(s\left[u\right]_{p})$

Model for N^{b}:

We assume we have a model I^{b} generated for N^{b} in the standard method:
The model I^{b} is defined by a congruence relation on terms over $\mathfrak{L}^{b}$, we define it using the term rewrite system R^{b} that is constructed in the standard completeness proof - from which we get the following properties:
(B1) R^{b} is a TRS over $\mathfrak{L}^{b}$that is left reduced and oriented w.r.t. $\succ$ and hence convergent - formally:
(B2) Each rule $u \rightarrow v \in R^{b}$ has the following properties:
(B2.1) There is some $C = C_{0} \vee u = v \in N^{b}$ s.t.
(B2.2) $u \succ v$
(B2.3) $u = v \succ C_{0}$
(B2.4) $u \prec l^{t}$
This motivates our construction of $=_{\Gamma}$.
The congruence relation is $=_{R^{b}}$ as defined above and the domain of the model is $\mathbf{T}(\Sigma) / =_{R^{b}}$.
As R^{b} is convergent each term has a unique normal form. We define a function r^{b} that returns the normal form of the input - formally:
$r^{b}(t) \triangleq \min_{\prec}\left\{s \mid t \rightarrow_{R^{b}}^{*} s\right\}$
We extend this function to atoms, literals, clauses and clause-sets and denote $r^{b}(x)$ by x' for each of these objects.

By the properties of the TRS R^{b}:
$(I^{b} \models s = t) \Leftrightarrow s' = t'$
And by standard completeness proof of the standard calculus:

$\mathrm{I^b} \models \mathrm{N^b}$

Top interpretation
We now want to build a similar $\mathrm{I^t}, \mathrm{R^t}, \mathrm{r^t}$ for $\mathrm{N^t}$, which require some adjustments from the standard definitions.
The main difference is that, for each clause $\mathrm{C} \in \mathrm{N^t}$, we use $\mathrm{C'}$ rather than C in the construction.
We define first the partial models that we will show model all clauses smaller than a given clause - the definition is inductive and relies on the well-foundedness of the ordering $\prec$ on normal forms (by $\mathrm{r^b}$) of clauses.

Definition A.1.5. Partial top models $\mathrm{N^t_C}, \mathrm{R^t_C}, \mathrm{r^t_C}, \mathrm{E^t_C}, \mathrm{I^t_C}$
For any clause C (not necessarily in $\mathrm{N^t}$):
$\mathrm{N^t_C} \triangleq \{\mathrm{D} \in \mathrm{N^t} \mid \mathrm{D'} \prec \mathrm{C'}\}$ - (all clauses whose normal form is smaller than that of C in $\mathrm{N^t}$).
$\mathrm{R^t_C}$ is a TRS defined by $\mathrm{N^t_C}$ defined below.
As $\mathrm{R^t_C}$ is by construction oriented w.r.t. $\prec$ and left reduced, we define $\mathrm{r^t_C}$ as its normal form function $\mathrm{r^{R^t_C}}$.
The partial model $\mathrm{I^t_C}$ models all clauses smaller than C (as we show in the completeness proof).
The above symbols are defined inductively, based on the ordering on normal forms of clauses in $\mathrm{N^t}$:
Each clause in $\mathrm{N^t}$ may contribute up to one rule to the TRS that defines the final model - we use the set of rules $\mathrm{E^t_C}$ to denote the (potential) rule contributed by the clause C.
$\mathrm{E^t_C} \triangleq$

```
    If
        (1) $\mathrm{I_C} \not\models \mathrm{C}$
        (2) $\mathrm{C} = \mathrm{C_0} \vee \mathrm{l} = \mathrm{r}$ for some $\mathrm{C_0}, \mathrm{l} = \mathrm{r}$ s.t.
        (3) $\mathrm{r^t_C}(\mathrm{l'}) = \mathrm{l'}$ ($\mathrm{l'}$ is irreducible by $\mathrm{R^t_C}$)
        (4) $\mathrm{l'} \succ \mathrm{r'}$
        (5) $\mathrm{l'} = \mathrm{r'} \succ \mathrm{C'_0}$
        (6) $\forall \mathrm{s} = \mathrm{t} \in \mathrm{C_0} \cdot (\mathrm{l'} = \mathrm{s'} \Rightarrow \mathrm{I_C} \models \mathrm{r} \neq \mathrm{t})$
    then $\{\mathrm{l'} = \mathrm{r'}\}$
    else $\emptyset$
```

The TRS $\mathrm{R^t_C}$ is the partial model form all clauses whose normal form (in $\mathrm{R^b}$) is smaller than that of C:
$\mathrm{R^t_C} \triangleq \cup\{\mathrm{E^t_D} \mid \mathrm{D} \in \mathrm{N^t_C}\}$
It is easy to see that $\mathrm{R^t_C}$ is left reduced (3) and oriented (4), and hence we can define its normal form function $\mathrm{r^t_C}$:
$\mathrm{r^t_C}(\mathrm{t'}) \triangleq \min\left\{\mathrm{s'} \mid \mathrm{t'} \rightarrow^*_{\mathrm{R^t_C}} \mathrm{s'}\right\}$
We only apply $\mathrm{r^t_C}$ to terms reduced by $\mathrm{r^b}$.
A slightly delicate point is that, while $\succ$ is total, we order clauses by their normal form, so we can have $\mathrm{C} \neq \mathrm{D}$ while $\mathrm{C'} = \mathrm{D'}$.
This is not a problem in our case as $\mathrm{C'} = \mathrm{D'} \Rightarrow \mathrm{E^t_C} = \mathrm{E^t_D}$ in this case because the definition of $\mathrm{E^t_C}$ depends only on the normal form $\mathrm{C'}$, and $\mathrm{R^t_C}$ is a set.

Definition A.1.6. Global TRSs and global partial TRSs - $\mathrm{R^t}, \mathrm{R}, \mathrm{r^t}, \mathrm{r}, \mathrm{R_C}, \mathrm{r_C}$
We define now the TRS that, combined with $\mathrm{R^b}$, models N:
$\mathrm{R^t} \triangleq \cup\{\mathrm{E^t_C} \mid \mathrm{C} \in \mathrm{N^t}\}$

$\mathrm{R} \triangleq \mathrm{R}^t \cup \mathrm{R}^b$
R, R^t are by definition left reduced.
By convergence we can define the normal form functions:
$\mathrm{r}^t(\mathrm{t}') \triangleq \min\{\mathrm{s}' \mid \mathrm{t}' \to^*_{\mathrm{R}^t} \mathrm{s}'\}$
$\mathrm{r}(\mathrm{t}) \triangleq \min\{\mathrm{s} \mid \mathrm{t} \to^*_{\mathrm{R}} \mathrm{s}\}$
We only apply r^t to terms reduced by r^b.
By construction, R^t_C is also left reduced by R^b and hence we can define:
$\mathrm{R}_C \triangleq \mathrm{R}^b \cup \mathrm{R}^t_C$
$\mathrm{r}_C(\mathrm{t}) \triangleq \min\left\{\mathrm{s} \mid \mathrm{t} \to^*_{\mathrm{R}_C} \mathrm{s}\right\}$

Lemma A.1.4. Composition of normal form functions
It is easy to see that:
$\forall \mathrm{t} \cdot \mathrm{r}_C(\mathrm{t}) = \mathrm{r}^t_C(\mathrm{r}^b(\mathrm{t}))$
$\forall \mathrm{t} \cdot \mathrm{r}(\mathrm{t}) = \mathrm{r}^t(\mathrm{r}^b(\mathrm{t}))$

Definition A.1.7. $\mathrm{R}^t_{C+}, \mathrm{R}_{C+}, \mathrm{r}^t_{C+}$
We will use:
$\mathrm{R}^t_{C+} \triangleq \mathrm{R}^t_C \cup \mathrm{E}^t_C$ and similarly
$\mathrm{R}_{C+} \triangleq \mathrm{R}_C \cup \mathrm{E}^t_C$, and the corresponding function
$\mathrm{r}^t_{C+}(\mathrm{t}') \triangleq \min\left\{\mathrm{s}' \mid \mathrm{t}' \to^*_{\mathrm{R}^t_{C+}} \mathrm{s}'\right\}$

Definition A.1.8. Models - $\mathrm{I}^t_C, \mathrm{I}_C, \mathrm{I}^t, \mathrm{I}^t_{C+}, \mathrm{I}_{C+}, \mathrm{I}$
We now define the models as above:
$\mathrm{I}^t_C \triangleq \{\mathrm{s} = \mathrm{t} \mid \mathrm{r}^t_C(\mathrm{s}) = \mathrm{r}^t_C(\mathrm{t})\}$
$\mathrm{I}^t_{C+} \triangleq \left\{\mathrm{s} = \mathrm{t} \mid \mathrm{r}^t_{C+}(\mathrm{s}) = \mathrm{r}^t_{C+}(\mathrm{t})\right\}$
$\mathrm{I}_C \triangleq \{\mathrm{s} = \mathrm{t} \mid \mathrm{r}_C(\mathrm{s}) = \mathrm{r}_C(\mathrm{t})\}$
$\mathrm{I}_{C+} \triangleq \{\mathrm{s} = \mathrm{t} \mid \mathrm{r}_{C+}(\mathrm{s}) = \mathrm{r}_{C+}(\mathrm{t})\}$
$\mathrm{I}^t \triangleq \{\mathrm{s} = \mathrm{t} \mid \mathrm{r}^t(\mathrm{s}) = \mathrm{r}^t(\mathrm{t})\}$
$\mathrm{I} \triangleq \{\mathrm{s} = \mathrm{t} \mid \mathrm{r}(\mathrm{s}) = \mathrm{r}(\mathrm{t})\}$
I will be our final model (with the standard interpretation) for N.

Definition A.1.9. $\max'_t, \max'_l$
We define $\max'_t(C) = \max_{\succ}\{\mathrm{t}' \mid \mathrm{t} \lhd C\}$ the maximal normal form of a term in C, and similarly the maximal normal form of a term for a literal.
We also define the maximal normal form of a literal in a term as
$\max'_l(C) = \max_{\succ}\{\mathrm{l}' \mid \mathrm{l} \in C\}$.

Lemma A.1.5. Ordering and maximal terms and literals
By the properties of $\succ$:
Clause ordering is determined first by the maximal term:
$\forall C, D \in N \cdot C' \succ D' \Rightarrow \max'_t(C) \succeq \max'_t(D)$
$\forall C, D \in N \cdot (\max'_t(C) \succ \max'_t(D) \Rightarrow C' \succ D')$
And
$\forall C, D \in N \cdot C' \succ D' \Rightarrow \max'_l(C) \succeq \max'_l(D)$
$\forall C, D \in N \cdot (\max'_l(C) \succ \max'_l(D) \Rightarrow C' \succ D')$

Persistence lemmas:
We now give a few lemmas that will help us show that I is indeed a model for N. The lemmas are mostly the standard lemmas, except that we use the normal forms (by r^b) of clauses rather than the original C.
We look at the sequence of partial models I^t_C arranged by increasing C, and

show that for most literals $\mathrm{l} \in \mathrm{C}$, the truth of l persists in the sequence from $\mathrm{I}^{\mathrm{t}}_{\mathrm{C}}$ on, and hence also in I^{t}.
We also show some weaker properties for maximal literals and non-maximal literals that contain the maximal term.

Lemma A.1.6. Occurrences of a maximal term in a clause
If the maximal term appears in a positive literal, then it does not appear in a negative literal and the maximal literal is positive:
For all C,s,t,u,v, if:
(a1) $\mathrm{C} \in \mathrm{N}$
(a2) $\mathrm{s} = \mathrm{t} \in \mathrm{C}$
(a3) $\mathrm{s}' = \mathrm{max}'_{\mathrm{t}}(\mathrm{C})$
(a4) $\mathrm{u} \neq \mathrm{v} \in \mathrm{C}$
Then
(c1) $\mathrm{s}' \succ \mathrm{u}', \mathrm{v}'$
Proof:
By the definition of ordering for literals.

Lemma A.1.7. Persistence of the normal form for non-maximal terms
If
(a1) $\mathrm{C}, \mathrm{D} \in \mathrm{N}$
(a2) $\mathrm{s} \lhd \mathrm{C}$
(a3) $\mathrm{D}' \succeq \mathrm{C}'$
(a4) $\mathrm{max}'_{\mathrm{t}}(\mathrm{C}) \succ \mathrm{s}'$
Then
(c1) $\mathrm{r}^{\mathrm{t}}_{\mathrm{C}}(\mathrm{s}') = \mathrm{r}^{\mathrm{t}}_{\mathrm{D}}(\mathrm{s}') = \mathrm{r}^{\mathrm{t}}(\mathrm{s}')$
Proof:
The proof is immediate from the definition of $\mathrm{E}^{\mathrm{t}}_{\mathrm{C}}$ and $\mathrm{r}^{\mathrm{t}}_{\mathrm{C}}$:
if $\mathrm{u} = \mathrm{v} \in \mathrm{E}^{\mathrm{t}}_{\mathrm{D}}$ for $\mathrm{D}' \succ \mathrm{C}'$ then $\mathrm{u}' \succeq \mathrm{max}'_{\mathrm{t}}(\mathrm{C}) \succ \mathrm{s}'$.

Using the above lemmas, and the fact that $\mathrm{E}^{\mathrm{t}}_{\mathrm{C}}$ can only include a rewrite rule $\mathrm{l}' \to \mathrm{r}'$ where $\mathrm{l}' = \mathrm{max}'_{\mathrm{t}}(\mathrm{C})$, we see that, for any dis-equality appearing in a clause, the normal form for the dis-equality is the same for any R_{D} where $\mathrm{D}' \succeq \mathrm{C}'$:

Lemma A.1.8. Persistence of dis-equalities
For any C,D,s,t, if :
(a1) $\mathrm{D} \in \mathrm{N}^{\mathrm{t}}$
(a2) $\mathrm{s} \neq \mathrm{t} \in \mathrm{C}$
(a3) $\mathrm{D}' \succ \mathrm{C}'$
Then
(c1) $\mathrm{I}_{\mathrm{C}} \models \mathrm{s} \neq \mathrm{t} \Leftrightarrow \mathrm{I}_{\mathrm{D}} \models \mathrm{s} \neq \mathrm{t} \Leftrightarrow \mathrm{I} \models \mathrm{s}' \neq \mathrm{t}'$
The proof is straightforward from lemmas A.1.6 and A.1.7 :
$\forall \mathrm{C}, \mathrm{D} \in \mathrm{N}, \mathrm{s} \neq \mathrm{t} \in \mathrm{C} \cdot \mathrm{D}' \succeq \mathrm{C}' \Rightarrow \mathrm{r}^{\mathrm{t}}_{\mathrm{C}}(\mathrm{s}', \mathrm{t}') = \mathrm{r}^{\mathrm{t}}_{\mathrm{D}}(\mathrm{s}', \mathrm{t}') = \mathrm{r}^{\mathrm{t}}(\mathrm{s}', \mathrm{t}')$

Lemma A.1.9. Persistence of equalities not containing the maximal term
Following the same reasoning as for disequalities, positive equalities persist if they do not contain the maximal term, as otherwise $\mathrm{E}^{\mathrm{t}}_{\mathrm{D}}$ for some $\mathrm{D}' \succeq \mathrm{C}'$ could rewrite the maximal term:

For any C,D,s,t, if
(a1) $\mathrm{D} \in \mathrm{N}^{\mathrm{t}}$

(a2) $s = t \in C$
(a3) $D' \succeq C'$
(a4) $s', t' \prec \max'_t(C)$
Then
(c1) $I_C \models s = t \Leftrightarrow I_D \models s = t \Leftrightarrow I \models s = t$
Proof:
For each $B \in N^t$ s.t. $D' \succeq B' \succ C'$, $\max'_t(B) \succeq \max'_t(C) \succ s', t'$ and hence $r^t_C(s', t') = r^t_D(s', t') = r^t(s', t')$ by lemma A.1.7.

Positive equalities always persist in the forward direction, as
$D' \succeq C' \Rightarrow R^D \supseteq R^C$:

Lemma A.1.10. Persistence of true positive equalities
If a positive equality holds at an interpretation, it holds in all subsequent interpretations:
For all C,D,s,t, if:
(a1) $D \in N^t$
(a2) $s = t \in C$
(a3) $D' \succeq C'$
(a4) $I^t_C \models s' = t'$
Then
(c1) $I^t_D \models s' = t'$
Proof:
This holds as $R^t_D \supseteq R^t_C$ and both are convergent.

Lemma A.1.11. Backward persistence by maximal term
For all C,D,s, if:
(a1) $C, D \in N^t$
(a2) $s' = \max'_t(C) = \max'_t(D)$
(a3) $r^t_D(s') = s'$
(a4) $C' \preceq D'$ Then
(c1) $R^t_D = R^t_C$

Proof:
(s1) For any B s.t. $(C' \preceq B' \prec D')$:
(s2) $s' = \max'_t(B)$ by the definition of $\prec$.
(s3) $E^t_B \in \{\emptyset, \{s' \to t'\}\}$ for some t s.t. $s' \succ t'$ (s2) definition of E^t_B.
(s4) As $r^t_D(s') = s'$ and $E^t_B \in R^t_D$ we know that $E^t_B = \emptyset$ for any such B.
(s5) $R^t_C = \cup\{E^t_A \mid A \in N^t \wedge C \succ A\}$
(s6) $R^t_D = \cup\{E^t_A \mid A \in N^t \wedge D \succ A\}$
(s7) $R^t_D = R^t_C \cup \{E^t_A \mid A \in N^t \wedge C \preceq A \prec D\} = R^t_C \cup \emptyset = R^t_C$ (s5)(s6)

We now want to characterize persistence for non-maximal equalities that include the maximal term.
These equalities are not, in general, persistent, however we have the following weaker property:

Lemma A.1.12. Forward persistence of the non-maximal part of a productive clause
If a clause produces a rewrite rule, then all non maximal literals in the clause persist.

For all C, D, C_0, l, r, if:
(a1) $C = C_0 \vee l = r \in N^t$
(a2) $E^t_C = \{l' \to r'\}$
  (a2a) $l' \succ r'$
  (a2b) $l' = r' \succ C'_0$
  (a2c) $I_C \not\models C$
  (a2d) $r^t_C(l') = l'$
  (a2e) $\forall s = t \in C_0 \cdot (l' = s' \Rightarrow I_C \models r \neq t)$
(a3) $D \succ C$
Then
(c1) $I_D \not\models C_0$
(c2) $I \not\models C_0$

Proof:
(s1) For any $s \neq t \in C_0$ this is immediate from lemma A.1.8
(s2) Select an arbitrary $s = t \in C_0$
(s3) w.l.o.g $s' \succeq t'$
(s4) $s' \succ t'$ by (a1) (a2c) and $I_C \supseteq I^b$
(s5) $I_C \not\models s = t$ by (s2) (a2c)
(s6) We define $C_0 = C_1 \vee s = t$
(s7) $l' = s'$ by (lemma A.1.9) (a2a) (a2b) (s3)
(s8) $r' \succ t'$ by (a1)(a2b) (s6)
(s9) $I_D \supseteq I_{C+}$ by (a2)(a3) and the definition of E^t_C
(s10) $r^t_D(s') = r^t_D(l') = r^t_D(r') = r^t_C(r')$ by (a2a)(s7)(s9) as r^t_D is convergent
(s11) $r^t_D(t') = r^t_C(t')$ by (a3)(s4)(s5) (lemma A.1.7)
(s12) $r^t_D(s') = r^t_D(t')$ by (s2)
(s13) $r^t_C(t') = r^t_C(r')$ by (s10) (s11) (s12)
(s14) $r^t_C(t') \neq r^t_C(r')$ by (a2e) (s2) (s3)
(s15) (s13) contradicts (s14)
  ■

Lemma A.1.13. Persistence of clauses in generated model
For all C,D, if
(a1) $C \in N^t$
(a2) $D \in N^t$
(a3) $D' \succ C'$
(a4) $I_{C+} \models C$
Then
(c1) $I_D \models C$
(c2) $I \models C$

Proof:
(s1) $I_D \supseteq I_{C+}$ (a3)
(s2) We select a literal $s \bowtie t \in C$ s.t. $I_{C+} \models s \bowtie t$ (a4)
(s3) w.l.o.g. $s' \succeq t'$
(4) If $s \bowtie t \equiv s \neq t$
  (4.1) $I_D \models s \neq t, I \models s \neq t$ (lemma A.1.8)
(5) If $s \bowtie t \equiv s = t$
  (5.1) If $(s' \prec \max'_t(C))$
    (5.1.1) $I_D \models s = t, I \models s = t$ (lemma A.1.9)

(5.2) If $s' = t'$

(5.2.1) $I_D \models s = t, I \models s = t$ (lemma A.1.9) by $I^b \models s = t, I_D \supseteq I^b$

(5.3) Else $s' \succ t'$

(5.4) $r^t_{C+}(s') = r^t_{C+}(t')$ (s2)

(5.5) Hence $s' \succ r^t_{C+}(s')$

(5.6) There is a rule $u' \rightarrow v' \in R^t_{C+}$, for some v, s.t.

(5.6.1) $u' = s|_p$ for some position p

(5.7) Each rule $l' \rightarrow r' \in R^t_D \setminus R^t_{C+}$ satisfies $l' \ntrianglelefteq s', t'$

(5.8) Hence $r^t_D s' = r^t_{C+}(s') = r^t_{C+}(t') = r^t_D(t')$

(6) $I_D \models C$

(7) $I \models C$ as $R_D \subseteq R$ and they are both convergent

■

Completeness Proof

We are now ready to give the completeness proof:

Theorem A.1.14. Completeness of the ground interpolation calculus
If
(A1) N is a colored set of clauses.
(A2) N is saturated up to redundancy w.r.t. the calculus SP_I.
(A3) N does not include the empty clause.
Then
(C1) I, as defined above, is a model for N.

Proof. We use induction on the ordering of bottom-normalized terms to show that each clause holds in I.
We prove:
$\forall C \in N^t \cdot I_{C+} \models C$
And the induction hypothesis is:
$\forall D \in N^t \cdot D' \prec C' \Rightarrow I_{D+} \models D$

Given N, C s.t.
(a1) $\square \notin N$
(a2) N is coloured
(a3) N is saturated up to redundancy
(a4) $C \in N^t$ ($\Rightarrow C \succ l^t$)
(a5) i.h. $\forall D \in N^t \cdot D' \prec C' \Rightarrow I_{D+} \models D$
Our objective is to show:
$I_{C+} \models C$

Proof:
(1.1) $I_{C+} = \bigcup_{D \preceq C} E_D$ by (definition A.1.8)

(2) If $I_C \models C$

(2.a.1) $I_{C+} = I_C$ (definition A.1.8)

(2.a.2) $I_{C+} \models C$

(2.a.3) done

(2e) We assume from here on that $I_C \not\models C$

Maximal dis-equality

(3) If there is a maximal (normal form) literal of C that is a disequality $s \neq t \in C$

- (3.1) We define $C = C_0 \vee s \neq t$
- (3.2) $s' \neq t' \succeq C_0$ and w.l.o.g. $s' \succeq t'$
- (3.3) If $s' = t'$
  - (3.3.1) There is a clause B s.t. (lemma A.1.1)(3.3)
    - (3.3.1.1) $B = \mathbf{idas}_\Gamma(s, t)$
    - (3.3.1.2) $I^b \not\models B$
    - (3.3.1.3) $B \prec l^t$
  - (3.3.2) The following derivation is valid in N, we name the conclusion R:

$$\frac{C_0 \vee \underline{\underline{s \neq t}}}{B \vee C_0}$$

(i)	$[s \neq t \not\prec_i C_0]$	(3.2)(lemma 5.4.1)
(ii)	$[s \succ l^t]$	(a4)(3.1)(3.2)
(iii)	$[B = \mathbf{idas}_\Gamma(s, t)]$	(3.3.1.1)

  - (3.3.3) $R' \prec C'$ (a4) (3.2) (3.3.1.3)
  - (3.3.4) $I_C \not\models R$ (2e) (3.1) (3.3.1.2)
  - (3.3.5) $I_{R+} \not\models R$ (3.3.3) (3.3.4) (lemma A.1.13)
  - (3.3.6) Contradiction to (i.h.)(a3)(3.3.2)
- (3.3e) $s' \neq t'$ (3.3-3.3.6)
- (3.4) $s' \succ t'$ (3.2) (3.3)
- (3.5) $r^t_C(s') = r^t_C(t')$ (2e) (3.1)
- (3.6) There is a rule $l' \to r' \in R^t_C$ s.t. (3.4)(3.5)(def A.1.5)
  - (3.6.1) $l' = s'|_p$ for some p
  - (3.6.2) There is some B s.t. (lemma A.1.1)(3.6.1)
    - (3.6.2.1) $B = \mathbf{idas}_\Gamma(l, s|_p)$
    - (3.6.2.2) $I^b \not\models B$
    - (3.6.2.3) $B \prec l^t$
  - (3.6.3) $l' \succ r'$
  - (3.6.4) There is a clause $D = D_0 \vee l = r \in N^t$ s.t. (3.6) (def A.1.5)
    - (3.6.4.1) $D' \prec C'$
    - (3.6.4.2) $l' = r' \succ D'_0$
    - (3.6.4.3) $I_D \not\models D$
    - (3.6.4.4) $E^t_D = \{l' \to r'\}$
- (3.7) $I_C \not\models D_0$ (lemma A.1.12)
- (3.8) $I_C \models l = r$ (3.6)
- (3.9) The following instance of $\sup^i_{\neq}$ is valid in N,
  we name the conclusion R:

$$\frac{D_0 \vee \underline{l} = r \quad \underline{s} \neq t \vee C_0}{B \vee C_0 \vee D_0 \vee s\,[r]_p \neq t}$$

(i)	$[l \not\preceq_i r]$	(3.6.3)(lemma 5.4.1)
(ii)	$[l = r \not\preceq_i D_0]$	(3.6.4.2)(lemma 5.4.1)
(iii)	$[s \not\preceq_i t]$	(3.4)(lemma 5.4.1)
(iv)	$[s \neq t \not\preceq_i C_0]$	(3.2)(lemma 5.4.1)
(v)	$[s \neq t \not\preceq_i l = r]$	(3.6.1)(lemma 5.4.1)
(vi)	$[l, s \succeq l^t]$	(a4)(3.1)(3.2)(3.6.4)
(vii)	$[B = \mathbf{idas}_\Gamma(l, s\|_p)]$	(3.6.2.1)

(3.10) $R' \prec C'$

$[B']$	(3.6.2.3)
$[C'_0]$	(3.1)(3.2)
$[D'_0]$	(3.6.4)+(3.6.4.1)+(3.6.4.2)
$[s\,[r]_p \neq t]$	(3.1)(3.2)(3.6.3)

(3.11) $I_C \not\models R$

$[B]$	(3.6.2.2)
$[C_0]$	(2e)
$[D_0]$	(3.7)
$[s\,[r]_p \neq t]$	(2e)(3.8)(3.6.1)

(3.12) $I_{R+} \not\models R$ (3.11)(3.6.4.1) (lemma A.1.13)
(3.13) Contradiction to (i.h.) (a3)(3.9)
(3e) There is no maximal negative literal in C (3-3.13)
(3e.1) All maximal literals in C are positive (3e)

Maximal equality literal:
(4) We select a maximal literal $s = t \in C$
(4.1) $C = C_0 \vee s = t$
(4.2) $s' = t' \succeq C'_0$
(4.3) w.l.o.g. $s' \succeq t'$
(4.3.1) $s \succ l^t$ (4.3)(4.2)(4.1)(a4)
(4.4) If $s' = t'$
(4.4.1) $I^b \models s = t$
(4.4.2) $I_C \models s = t$
(4.4.3) Contradicts (2e)
(4.4e) $s' \succ t'$

Maximal term is reducible in R_C:
(5) If $s' \succ r^t_C(s')$
(5.1) There is a rule $l' \to r' \in R^t_C$ s.t.
(5.1.1) $l' = s'|_p$ for some p
(5.1.2) There is some B s.t. (lemma A.1.1)(5.1.1)
(5.1.2.1) $B = \mathbf{idas}_\Gamma(l, s|_p)$
(5.1.2.2) $I^b \not\models B$
(5.1.2.3) $B \prec l^t$
(5.1.3) $l' \succ r'$
(5.1.4) There is a clause $D = D_0 \vee l = r \in N^t$ s.t. (definition A.1.5)
(5.1.4.1) $D' \prec C'$
(5.1.4.2) $l' = r' \succ D_0$
(5.1.4.3) $I_D \not\models D$
(5.1.4.4) $E^t_D = \{l' \to r'\}$
(5.1.4.5) $l \succ l^t$ (5.1.4)(5.1.4.2)(5.1.3)

(5.2) $I_C \not\models D_0$ (5.1.4.4)(lemma A.1.12)
(5.3) $I_C \models l = r$ (5.1.4)
(5.4) The following instance of $\text{sup}^{\text{i}}_{=}$ is valid in N,
we name the conclusion R:

$$\frac{D_0 \vee \underline{l} = r \quad \underline{s} = t \vee C_0}{B \vee C_0 \vee D_0 \vee s\,[r]_p = t}$$

(i)	$[l \not\preceq_i r]$	(5.1.3)(lemma 5.4.1)
(ii)	$[l = r \not\preceq_i D_0]$	(5.1.4.2)(lemma 5.4.1)
(iii)	$[s \not\preceq_i t]$	(4.3)(lemma 5.4.1)
(iv)	$[s = t \not\preceq_i C_0]$	(4.2)(lemma 5.4.1)
(v)	$[s = t \not\preceq_i l = r]$	(4.2)(5.1.4)(5.1.4.1)(5.1.4.2)(lemma 5.4.1)
(vi)	$[l, s \succeq l^t]$	(4.3.1)(5.1.4.5)
(vii)	$[B = \mathbf{idas}_\Gamma(l, s|_p)]$	(5.1.2.1)

(5.5) $R' \prec C'$

$[B']$	(5.1.2.3)
$[C'_0]$	(4.2)
$[D'_0]$	(5.1.4)+(5.1.4.1)+(5.1.4.2)
$[s\,[r]_p = t]$	(4.1)(4.2)(5.1.3)

(5.6) $I_C \not\models R$

$[B]$	(5.1.2.2)
$[C_0]$	(2e)
$[D_0]$	(5.2)
$[s\,[r]_p = t]$	(2e)(5.3)

(5.7) $I_{R+} \not\models R$ (5.3) (5.4) (lemma A.1.13)
(5.8) Contradiction to (i.h.) (a3)(5.4)
(5e) $r^t(s') = s'$ (5)-(5.8)

Maximal term occurs in a non-maximal equality literal

(6) If there is a literal $l = r \in C_0$ s.t.
(6.0.1) $s' = l'$
(6.0.2) $I_C \models t' = r'$
(6.1) We define $C_0 = C_1 \vee l = r$
(6.3) $t' \succeq r'$ (4.2)
(6.4) $l' \succ r'$ (2e)
(6.5) There is some B s.t. (3.3) (lemma A.1.1)
(6.5.1) $B = \mathbf{idas}_\Gamma(l, s)$
(6.5.2) $I^b \not\models B$
(6.5.3) $B \prec l^t$
(6.6) The following instance of $\text{fact}^{\text{i}}_{=}$ is valid in N,
we name the conclusion R:

$$\frac{C_1 \vee l = r \vee \underline{s = t}}{B \vee C_1 \vee r \neq t \vee s = r}$$

(i)	$[s \not\preceq_i t]$	(4.3)(lemma 5.4.1)
(ii)	$[s = t \not\preceq_i C_1 \vee l = r]$	(4.1)(4.2)(6.1)(lemma 5.4.1)
(iii)	$[l \not\preceq_i r]$	(6.4)(lemma 5.4.1)
(iv)	$[s \succ l^t]$	(4.3.1)
(v)	$[B = \mathbf{idas}_\Gamma(l, s)]$	(6.5.1)

(6.6) $R' \prec C'$

$[B]$	(6.5.3)
$[C_1]$	(6.1)(4.2)
$[r \neq t]$	(4.2)(6.0.1)(6.4)
$[s = r]$	(4.2)(6.0.1)(6.3)(6.4)

(6.7) $I_C \not\models R$

$[B]$	(6.5.2)
$[C_1]$	(2e)(6.1)
$[r \neq t]$	(6.0.2)
$[s = r]$	(2e)(6.0.2)

(6.8) $I_{R+} \not\models R$ (6.6) (6.7) (lemma A.1.13)

(6.9) Contradiction to (i.h.)(a3)(6.6)

(6e) $\forall l = r \in C_0 \cdot (l' = s' \Rightarrow I_C \not\models t' = r'$

(6e.1) $s' = t' \succ C_0$

(7) $E^t_C = \{s' = t'\}$ (2e) (3e) (4.4e) (5e) (6e) (6e.1) (definition A.1.5)

(7.1) $I_{C+} \models C$ (4.1)(7)(definition A.1.5)

$\square$

Interpolation corollary:

We can get a version of the (reverse) interpolation theorem for ground clausal first order logic with equality as a corollary:

For two sets of clauses A, B where $A \not\models \square$, $B \not\models \square$ but $A, B \models \square$, we can show a set of clauses S s.t. $A \models S$, $S, B \models \square$, and S includes only constants that appear in both A and B:

We give the weight $2\omega + 1$ to each constant in $\mathrm{consts}(A) \setminus \mathrm{consts}(B)$,

$1\omega + 1$ to each constant in $\mathrm{consts}(A) \cap \mathrm{consts}(B)$ and

1 to each constant in $\mathrm{consts}(B) \setminus \mathrm{consts}(A)$.

We set $l^b = \omega$, $l^t = 2\omega$.

As the calculus is complete, if we generate all valid inferences fairly, we will derive the empty clause eventually.

We define the interpolant J as $J = \{C \in SP_I(A) \mid C \prec l^t\}$ - where $SP_I(A)$ is the set of clauses we have derived with premises in A that are in N^b.

By the soundness of SP_I $A \models J$ and by definition $\square \in SP(J \cup B)$.

Similarly we can build sequence interpolants using a separate l^t, l^b for each pair of nodes arranged accordingly, and, as we will show later, with some reworking of the ordering we can also construct DAG interpolants.

A.1.2 Improved justification extraction

In this section we show an alternative encoding of the approximation Γ of the equivalence relation of successor CFG-nodes. This alternative encoding uses a hierarchical structure of EC-graphs rather than a single EC-graph for approximation, and hence can help us reduce the size of the interpolants we produce.

In this sub-section we describe an improvement of the part of the algorithm that over-approximates Γ - it mostly replaces g^i_n in deciding $=_\Gamma$ and especially in generating interpolation clauses (ensuring optimal clauses) and justifications. We have seen sub-optimal interpolation clauses in the example in figure 5.18. We assume a given EC-graph g^t and the set of equations Γ over I^t, which over-approximates the equations of all suffix paths restricted to I^t.

Basic idea:

Instead of the graph g^i_n, we create a tree of EC-graphs, where the root is g^t and each tree-node is an EC-graph obtained from its parent by `assuming` one relevant equation from Γ - this equation is the label of the parent edge. The leaves are the EC-graphs with no relevant equalities or that are in conflict (that is, an EC-node has a dis-equality self-loop). At each tree-node we have an EC-graph g obtained from its tree-parent EC-graph by `assuming` the edge label of its parent edge (the edge label is an equation of two EC-nodes of g^t, we follow the source-edges backwards from g^p to g^t (selecting one representative when there are several options) in order to translate the equation to be over two EC-nodes of g^p). Each such tree-node g is associated with the sequence S of interface equations on the path leading to it from the tree-root. We establish the source invariant between each EC-graph g and the EC-graph g^p of its parent tree-node.

Extracting interpolants:

For each such tree-node g we may output some interpolation clause-ECs: The body is always the disjunction of the negation of all equations on the DAG-path leading to g. If g is in conflict, we output only the body. Otherwise, we basically want to output one interpolation clause for each pair of interface terms $s, t \in \mathbf{terms}(g^t)$ s.t. $g \models s = t \wedge g^p \not\models s = t$. This set of pairs of terms can be infinite if g has cycles (e.g. a=f(a)), but we can output a sufficient number of clauses that imply any such equality.

The set of heads of Horn clauses for g is defined as follows:

The simple case is when two interface nodes $u, v \in g^p$ are merged into one interface node $w \in g$ by congruence closure - for example, in 5.21, the tree has one tree-node beside the root - $g_{a=b}$ (as a=b is the only relevant equality - in fact it is $[a]^t = [b]^t$). At g^t, $[c]^t \neq [d]^t$ while at g_1, $[c]_{a=b} = [d]_{a=b}$, hence we output the interpolation clause $[a]^t \neq [b]^t \vee [c]^t = [d]^t$.

Looking at example 5.18, out tree will have four non-root nodes -

$g_{a=b}, g_{a=b,c=d}, g_{c=d}, g_{c=d,a=b}$

We only consider here $g_{a=b}, g_{c=d}$ as we will show below how to avoid adding the other two.

At $g_{a=b}$ we output $[a]^t \neq [b]^t \vee [e_1]^t = [e_2]^t$ while at $g_{c=d}$ we output

$[c]^t \neq [d]^t \vee [e_3]^t = [e_4]^t$ - so we have produced the optimal interpolation clauses.

The second case for the head of an interpolation clause is when, for some EC-node $n \in g$, we have a pair of interface GFAs $f(\overline{s}), g(\overline{t}) \in n$, s.t. at least one of the GFAs has no source in g^p that is an interface gfa.

In example 5.12, at the graph $g_{a=b}$ with the parent g^t, the EC-node $[e]_{a=b}$ has two interface gfas, $\{f([e, y]), e()\}$ where the first GFA has just one source in g^t

- $f([y])$, which is not an interface gfa, and hence we output the interpolation clause $[a]^t \neq [b]^t \vee [e]^t = [f(e)]^t$. Note that $[f(e)]^t$ does not exist in g^t and has to be added.
The way we choose which EC-nodes to add to g^t is as follows - we define a total order on ground terms s.t. all interface terms are smaller than all non-interface terms (described later), each GFA is represented by its minimal term and each EC-node is represented by the minimum on terms representing its gfas. If the representatives of our interface GFAs at g do not appear in g^t, we add them.
In the above example,
$\mathbf{terms}([e]_{a=b}) = \left\{f^k(e), f^k(y) \mid k \geq 0\right\} \cup \{h(a, x), h(b, x)\}$ - the smallest interface term is e.
For the GFA $f([e])$ the representative is $f(\mathbf{rep}([e])) = f([e])$, which has to be added to g^t.

Reducing redundancy using a DAG

Evidently, as described above, the algorithm is very inefficient, as each set of interface equalities can be reached on several paths - one for each order of `assuming` the equalities - we could use sets instead of sequences to characterize tree-nodes.
Hence we use a DAG rather than a tree, where each DAG-node that represents the set of equalities S will have, for each $u = v \in S$, a parent DAG-edge to the DAG-node that represents $S \setminus \{u = v\}$. We avoid creating a child node if *any* of its parents are in conflict, as it will not produce any new interpolation clauses. Additionally, we only output interpolation clauses as above if the conditions for generating the clause hold for *all* parents.
For example, in figure 5.19, the root of the DAG is the graph g^t.
The root has two depth-1 children - $g_{a=b}, g_{c=d}$, as per the relevant equalities - producing the interpolation clauses $a \neq b \vee e_1 = e_2, c \neq d \vee e_3 = e_4$ respectively.
Each of these children has a relevant equality, producing the DAG-node
$g_{a=b,c=d}$. However, we will not produce the interpolation clauses
$a \neq b \vee c \neq d \vee e_1 = e_2, a \neq b \vee c \neq d \vee e_3 = e_4$ because $g_{a=b} \models e_1 = e_2$ and
$g_{c=d} \models e_3 = e_4$.

Reducing redundancy using disjoint parts

Looking at example 5.18, we see that the equalities $a = b, c = d$ are independent in the following sense: If we `assume(a=b)`, we have not modified the part of the EC-graph which is the (GFA) downward closure of $[e_3], [e_4]$ and vice versa, where `assume(c=d)` does not affect $[e_1], [e_2]$.
In this case, although $g_{a=b}$ has a relevant equality $c = d$ and $g_{c=d}$ has a relevant equality $a = b$, we will not gain any new interpolation clause by calculating $g_{a=b,c=d}$ because each interpolation clause is subsumed by one of its parents. In order to take advantage of this fact, for each EC-graph $g_{u=v}$ of depth one in the DAG, we trim any EC-node that is not in the down-closure of the up-closure of u,v - that is any term that is not needed to represent any super-term of u,v.
In our example the sub-term closure of the super-term closure of $\{[a], [b]\}$ is $\{[a], [b], [e_1], [e_2], [x]\}$ and similarly for $\{[c], [d]\}$ it is $\{[c], [d], [e_3], [e_4], [x]\}$.
After trimming $g_{a=b}, g_{c=d}$, they both have no more relevant equalities, and hence we avoid adding the meet DAG-node $g_{a=b,c=d}$.

In order to maintain completeness, we must, besides super-term closure, also maintain closure w.r.t. $=_\Gamma$- consider the example in figure A.1:

$N^t = \{y_0 = g(a, x), y_1 = g(b, x) = h(c, x), y_2 = h(d, x), y_0 \neq y_2\}$

Figure A.1: Example for different upward closures
Without $=_\Gamma$closure:
The graph $g_{a=b}$ will include $\{[a], [b], [c], [x], [y_0, y_1]\}$
The graph $g_{c=d}$ will include $\{[b], [c], [d], [x], [y_1, y_2]\}$
Both graphs will be leaves - we are missing the closure with $y_0 =_\Gamma y_2$.

It is also sufficient to include in each DAG-node only the EC-nodes that can be merged by one application of congruence closure, which are sufficient for calculating the relevant equalities, and only transitive super-terms once congruence closure is performed.

Reducing redundancy using path compression:
One case we have not yet covered is an asymmetric meet - consider the example in figure A.2

$N^t = \{b_0 = g(a_0, x), b_1 = g(a_1, x), y_0 = h(b_0, c, x), y_1 = h(b_1, d, x), y_0 \neq y_1\}$

Figure A.2: Example for an asymmetric meet
The graph $g_{a_0=a_1}$ will have a node $[b_0, b_1]$ and hence is effectively also $g_{a_0=a_1,b_0=b_1}$
The graph $g_{b_0=b_1}$ will have $g_{a_0=a_1}$ as a child.
The interpolation clauses will be:
$\{a_0 \neq a_1 \vee b_0 = b_1, b_0 \neq b_1 \vee c \neq d\}$

Here we can compress the DAG-path $a_0 = a_1.b_0 = b_1$ to $a_0 = a_1$ because of the implication.
When extracting interpolation clauses we can skip $a_0 \neq a_1 \vee c \neq d$ because it is derivable from the other two - this is detected because $g_{b_0=b_1} \models c = d$ and $g_{b_0=b_1}$ is a parent to $g_{a_0=a_1}$ which is, in fact, $g_{a_0=a_1,b_0=b_1}$. We do not reject the interpolation clause $a_0 \neq a_1 \vee b_0 = b_1$ although $g_{b_0=b_1} \models b_0 = b_1$ because $[b_0] = [b_1]$ is a consequence of $[a_0] = [a_1]$ in $g_{a_0=a_1}$ and *directly* assumed in $g_{b_0=b_1}$.

Another instance of path compression is shown in figure A.3:

$N^t = \{c = g(a_0, b_0, x), d = g(a_1, b_1, x)\}$
$N^b = \{a_0 = a_1, b_0 = b_1, c \neq d\}$

Figure A.3: Example for path compression
There is no need for the graphs $g_{a_0=a_1}, g_{b_0=b_1}$, only $g_{a_0=a_1,b_0=b_1}$.

We can skip any DAG-node where no congruence closure has been performed in comparison with *all* parents - any interpolation clause derived by transitive closure is not needed (e.g. if $g^t \models a = b, c = d$ and $g^b \models b = c$ then there is no need for the clause $b \neq c \vee a = d$ because we use the clauses $a = b, c = d$ which are on the interface (which is transitively closed).
In the incremental case, we might encounter a non-incremental interpolant - for example, in A.3, if we were to remove $c \neq d$ from N^b, derive the interpolation clause $a_0 \neq a_1 \vee b_0 \neq b_1 \vee c = d$, and later derive in N^t (in another fragment)

$c = f(a_0, x), d = f(a_1, x)$, then our interpolation clause is obsolete and should be replaced (subsumed) by $a_0 \neq a_1 \vee c = d$ - as we have stated before, removing the old clause from all successors is not very efficient, but, in the case where we have several successors, we cannot skip adding the smaller clause, as some successors might only entail $a_0 = a_1$ and not $b_0 = b_1$ e.g. $N^t = \{a_0 = a_1, c \neq d\}$. Hence, we add and propagate the shorter clause as well and rely on subsumption to eliminate the unnecessary clauses (although at the price of some redundant derivations).
We can, however, eliminate the DAG node $g_{a_0=a_1,b_0=b_1}$ as it has been subsumed. Using the above described path compression we ensure that, for each DAG-node, at least one pair of non-interface GFAs is merged that was not merged in *all* parents.

Complexity

The DAG can have, in the worst case, a single DAG-node per any subset of the possible interface-node equalities - hence $O(2^n)$ complexity for the set of interpolation clauses.
An example that achieves exponential complexity:

$$N^t = \left\{ \bigwedge_{i \in I, j \in J} (f(a_{i,j}, x) = y_i \wedge f(b_i, x) = z_i), c = f(\overline{z}), d = f(\overline{y}) \right\}$$

Here the non-redundant interpolants are:

$$\left\{ \bigwedge_{\overline{k} \in J^{|I|}} (c = d \vee \bigvee_{i \in I} a_{i,k_i} \neq b_i) \right\}$$

A total of $\theta(|J|^{|I|})$ clauses for an input size of $O(|I| \times |J|)$.

Worst case lower space bound, non-unit case:
The worst case complexity for any interpolant between two sets of ground (not necessarily unit) clauses is exponential, as expected from the propositional case (e.g. pigeon-hole principle). Even if the first set of clauses includes only unit ground clauses, the worst case minimal CNF interpolant is still exponential - for example:

$$N^t = \left\{ \bigwedge_{i \in I, j \in J} (f(a_{i,j}, x) = y_i \wedge f(b_i, x) = z_i), c = f(\overline{z}), d = f(\overline{y}) \right\}$$

$$N^b = \left\{ \bigwedge_{i \in I} (\bigvee_{j \in J} a_{i,j} = b_i), c \neq d \right\}$$

Where the minimal CNF interpolant is:

$$\left\{ \bigwedge_{\overline{k} \in J^n} (c = d \vee \bigvee_{i \in I} a_{i,k_i} \neq b_i) \right\}$$

As above, the minimal CNF interpolant is exponential in the input size.

Worst case upper space bound, unit case, binary:
The maximal space complexity of a minimal interpolant between two sets of unit ground clauses is polynomial, based on our above algorithm (the basic algorithm without the EC-graph DAG):
We let k be the number of EC-nodes in the larger between g^t and g^b. At each step we transfer at least one equality from g^b to g^t, hence decreasing the number of nodes by at at least one, and at least one equality from g^t to g^b - hence the number of literals of each interpolation clause (Horn clause) is at most k and

we have at most k Horn clauses - hence the interpolant (conjunction of above Horn clauses) is of at most k^2 size as a function of interface nodes (clause ECs). An interpolant over terms rather than EC-nodes could be exponential - for example:

$N^t = \left\{ \bigwedge_{i \in 1..n} x_{i+1} = f(x_i, x_i), x_0 = a, x_n \neq b \right\}$

$N^b = \left\{ \bigwedge_{i \in 1..n} m_{i+1} = f(m_i, m_i), m_0 = a, m_n = b \right\}$

Here, the size of the minimal representative term of $[x_n]^t$ is exponential in n, and similarly for $[m_n]^b$. However, if we allow shared representation (so that we count only distinct sub-terms), then the representation is polynomial.

Worst case lower space bound, unit case, CFG:
The problem of calculating an interpolant between a CFG node and all paths from it to assertions is equivalent to the non-unit case, even if each CFG-node only contains unit clauses - we can simulate an n-literal CNF clause by an n-branch with each literal assumed in one arm of the branch, which is immediately joined. Hence in this case the complexity is exponential.

Bounded fragments:
In realistic scenarios, we do not expect the above worst case example to be common, as e.g. examples of the pigeon-hole principle are not common as a verification condition for propositional programs.
In order to ensure a time and space complexity bound we simply limit the depth of any path in the DAG - as the out-degree of each DAG-node of depth k is at most $\binom{|g^t|-k}{2}$ (quadratic), it is easy to calculate the maximal size of the DAG up to any given depth n, with path compression we would not expect the necessary depth to be high for most realistic examples.

Unit disequalities

The algorithm for verification with only ground unit equalities presented in chapter 3 assumed (for simplicity) that all dis-equalities occur at assertion nodes. We have seen how dis-equalities can be represented in the EC-graph using dis-equality edges between EC-nodes. Propagating dis-equalities using sources is simple - for a CFG-node n, and two EC-nodes u,v, if in all predecessors there is a dis-equality edge between a source of u and a source of v, we can add the edge also between u and v. We also need to propagate dis-equalities on super-terms of EC-nodes with more than one source - but we omit the details here.
In this section we are interested in the effect of scoping on the propagation of dis-equalities. We show that our technique for generating interpolants can be used to generate dis-equality interpolants that strengthens the EC-graph based verification algorithm by allowing it to propagate more dis-equalities.

For a unit dis-equality at a CFG-node n that includes a sub-term that is not in scope in a successor CFG-node s, we can have several options for a unit interpolant - for example:
$\{g(f^7(a), x) \neq g(f^6(a), x)\}$
implies each of $\{f^{i+1}(a) \neq f^i(a) \mid 0 \leq i < 7\}$
While
$\{g(f^7(a), x) \neq g(f(a), x)\}$
implies the following (direct implication graph):

$$\begin{tikzcd}
 & & \mathrm{f}^7(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}) \arrow[dll, Rightarrow] \arrow[d, Rightarrow] \arrow[drr, Rightarrow] & & \\
\mathrm{f}^4(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}) \arrow[dr, Rightarrow] \arrow[drr, Rightarrow] & & \mathrm{f}^6(\mathrm{a}) \neq \mathrm{a} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \mathrm{f}^3(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}) \arrow[dll, Rightarrow] \arrow[dl, Rightarrow] \\
 & \mathrm{f}^3(\mathrm{a}) \neq \mathrm{a} \arrow[dr, Rightarrow] & \mathrm{f}^2(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}) \arrow[d, Rightarrow] & \mathrm{f}^2(\mathrm{a}) \neq \mathrm{a} \arrow[dl, Rightarrow] & \\
 & & \mathrm{f}(\mathrm{a}) \neq \mathrm{a} & &
\end{tikzcd}$$

Figure A.4: Unit disequalities

Intuitively, the simplest inteprolant is $\mathrm{f}^7(\mathrm{a}) \neq \mathrm{f}(\mathrm{a})$. In the set of unit dis-equalities implied by one top unit dis-equality, the *top disequalities* are those that are not implied by any other dis-equality in the set - in the above $\left\{\mathrm{f}^4(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}), \mathrm{f}^3(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}), \mathrm{f}^6(\mathrm{a}) \neq \mathrm{a}\right\}$

Consider the following example:
$\mathbf{C}_{\mathrm{p}} = \left\{\mathrm{f}^7(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}), \mathrm{f}^4(\mathrm{a}) = \mathrm{y}, \mathrm{f}^7(\mathrm{b}) \neq \mathrm{f}(\mathrm{b}), \mathrm{f}^4(\mathrm{b}) = \mathrm{z}\right\}$
$\mathbf{C}_{\mathrm{n}} = \left\{\mathrm{f}^3(\mathrm{a}) = \mathrm{a} \vee \mathrm{f}^2(\mathrm{b}) = \mathrm{b}\right\}$
Here, the interpolants $\left\{\mathrm{f}^7(\mathrm{a}) \neq \mathrm{f}(\mathrm{a}), \mathrm{f}^7(\mathrm{b}) \neq \mathrm{f}(\mathrm{b})\right\}$ imply all the others.
For EC-graphs with a depth bound of 1 and relative depth (as described in chapter 6), we cannot represent $\mathrm{f}^7(\mathrm{a}), \mathrm{f}^7(\mathrm{b})$ at n, but we can represent $\mathrm{f}^3(\mathrm{a}) \neq \mathrm{a}$ and $\mathrm{f}^2(\mathrm{b}) \neq \mathrm{b}$ - hence we want to generate the top dis-equalities for each dis-equality that is too deep for the successor.

Algorithm:
We can easily use our DAG-based algorithm to find the unit disequalities as follows:
We develop our DAG for a depth of 1, using only dis-equalities as goals (that is, we do not use pairs in I^{Γ} in order to calculate G).
Each conflicting EC-graph at a DAG-node represents one unit disequality.
This can be done incrementally, using any variant of Γ (over-approximation of equalities in transitive successors).

A.2 EC-graphs and clauses

In this section we show a combination of the verification algorithm for the non-unit ground fragment from section 4.1 with the EC-graph algorithm for the unit ground fragment presented in chapter 3. The main motivation to combine these fragments comes from the need for joins, and as an attempt to keep the representation in each CFG-node small by keeping all elements (term, clauses etc.) in normal form. As we have seen in section 3.4 regarding joins in the unit ground fragment, and in 4.3 regarding joins in the non-unit ground fragment, more equalities known at the join often translate to a better join. Using EC-graphs for the ground unit fragment, as described in chapter 3, allows us to propagate more equalities for each term at each program point. EC-graphs also allow us to perform more aggressive simplifications as they encode fully reduced rewrite systems, while superposition produces left reduced rewrite systems.

Overview: The main idea is as follows:

State: At each CFG-node n we maintain an EC-graph (g_n) as in chapter 3, and two sets of clauses ($\mathtt{done}_n$ and $\mathtt{todo}_n$) as in the previous sub-sections.

Unit equalities: Unit equalities are represented by the EC-graph and hence, instead of being added to $\mathtt{done}_n$ or $\mathtt{todo}_n$ they are `assumed` in g_n. We also maintain a queue $\mathtt{todoeq}_n$ whose equations are not saturated w.r.t. $\mathbf{SP_g}$. We propagate ground unit equalities as in chapter 3, and additionally, as alluded to before, we fall-back to non-unit propagation at joins when unit propagation is incomplete or inefficient.

Clauses: For each CFG-node n, we `add` all terms in all clauses to g_n, and represent all clauses in $\mathtt{done}_n$ and $\mathtt{todo}_n$ as clause-ECs - clauses over term-Ecs (GTs) rather than over ground terms. Each clause-EC represents an equivalence class of clauses. In the implementation we represent unit dis-equalities as edges in g_n, but we treat them as other non-unit clauses and hence we treat them here as if they are in the $\mathtt{done}_n$ and $\mathtt{todo}_n$ sets.

Requests: In section 3.3.3 we have introduced source edges, which connects an EC-node in a CFG-node with an EC-node in its direct predecessor if they share a term. Our EC-graph invariants ensure that two EC-nodes on a path that share a term on the path is connected by the transitive closure of this function (we say there is a source-chain between the two), as long as there are no joins on the path. Propagation requests are composed of GFAs rather than terms (so a request is an over-approximation) and, in addition, a request is translated using the source edges.

The conceptual difference from the algorithm presented in section 4.1 is that representing clauses as clause-ECs means that all clauses are fully reduced using rewrite by unit equalities, and using EC-nodes for requests allows us to reduce the overall number of requests both because of the sources function and because several requests are collected into one.

Representation

Each clause at a CFG-node is represented by a clause-EC - this is essentially a clause where each top-term in a literal is represented by an EC-node - formally:
An atom-EC is $s = t$ where s,t are EC-nodes.
A literal-EC is A or $\neg$A where A is an atom-EC.

A clause-EC is a (potentially empty) set of literal-ECs.
Conceptually, each EC-node t represents its minimal (by $\prec$) member - $\min_{\prec}(\mathbf{terms}(t))$, along with a set of rewrite rules (positive unit equalities) that define the equivalence class - one rule per non-minimal gfa.

In all of the above definitions, all EC-nodes come from the same EC-graph - we do not mix nodes from different EC-graphs. Different EC-graphs in different nodes are ECs for different congruences, and hence may overlap without being equal, making the check for intersection expensive, while two ECs of the same congruence are either identical or disjoint.

Representatives: As described in chapter 3, an EC-node is a non-empty set of GFAs, where a GFA $f(\bar{s})$ is a function symbol f and a tuple $\bar{s}$ of EC-nodes of the same arity as f. We define representatives using *prec* rather than depth. We denote the representative term of a node t by $\mathbf{rep}_{\prec}(t)$ - formally:
$\mathbf{rep}_{\prec}(t) \triangleq \min_{\prec}(\{\mathbf{rep}_{\prec}(f(\bar{s}))\} \mid f(\bar{s}) \in t)$
$\mathbf{rep}_{\prec}(f(\bar{s})) \triangleq f(\mathbf{rep}_{\prec}(\bar{s}))$
$\mathbf{rep}_{\prec}(\bar{s})_i \triangleq \mathbf{rep}_{\prec}(s_i)$
The representatives of literal-, atom-, and clause-ECs are defined analogously.
We compare EC-nodes using $\prec$ where:
$s \prec t \triangleq \mathbf{rep}_{\prec}(s) \prec \mathbf{rep}_{\prec}(t)$
And similarly for GFAs, atom-, literal- and clause-ECs. We only compare term-, literal- and clause-ECs composed of EC-nodes of the same EC-graph. The representative GFA - $\mathbf{repgfa}_{\prec}(t)$ of an EC-node t is the GFA that includes the minimal term:
$\mathbf{repgfa}_{\prec}(t) \triangleq \min_{\prec}(t)$
The non-representative GFAs -$\mathbf{nrepgfas}_{\prec}(t)$ are all the rest of the gfas:
$\mathbf{nrepgfas}_{\prec}(t) \triangleq t \setminus \mathbf{repgfa}_{\prec}(t)$
Note that $\mathbf{rep}_{\prec}(t)$ is different from $\mathbf{rep}(t)$ as it may not be of minimal depth.

Each atom-EC represents an equivalence class of atoms where the atoms represented are the atoms derived by replacing each top EC-node that appears in the atom with one of the terms it represents:
$\text{atoms}(s = t) \triangleq \{u = v \mid u \in \mathbf{terms}(s) \wedge v \in \mathbf{terms}(t)\}$
And similarly for literal- and clause-ECs.
The set of equalities represented by an EC-node is defined formally as follows:
$\mathbf{eqs}_{\prec}(t) \triangleq \{\mathbf{rep}_{\prec}(s) = \mathbf{rep}_{\prec}(t) \mid s \in \mathbf{nrepgfas}_{\prec}(t)\}$
The set of equalities represented by an EC-graph is the union of equalities for all EC-nodes:
$\mathbf{eqs}_{\prec}(g) \triangleq \cup_{t \in g} \mathbf{eqs}_{\prec}(t)$
Note that $\mathbf{eqs}_{\prec}(g)$ is always fully left and right reduced, and does not include equalities implied by $\texttt{done}_n$ and $\texttt{todo}_n$. The representative of a clause-EC is the clause that would be derived from any member of its EC by exhaustive application of the simplification by unit-equality rule from all equalities in the EC-graph - a normal form under the EC-graph of the CFG-node. A derivation between clause-ECs represents the derivation between their representatives.

Because clauses are simplified relative to the set of equalities of a certain CFG-node, if the clause is propagated over a join some of the equalities used to simplify it might not hold after the join.
The standard unit rewrite rule is:

$$\text{simp}_{=}\ \frac{\mathrm{l = r} \quad \cancel{\mathrm{C}}}{\mathrm{C}\,[\mathrm{r}]_{\mathrm{p}}} \quad \begin{array}{l} \mathrm{l} = \mathrm{C}|_{\mathrm{p}} \\ \mathrm{l} \succ \mathrm{r} \\ \mathrm{C} \succ \mathrm{l = r} \end{array}$$

While we apply a slightly more general version:

$$\text{simp}_{=}\ \frac{\mathrm{D} \vee \mathrm{l = r} \quad \cancel{\mathrm{D} \vee \mathrm{C}}}{\mathrm{C}\,[\mathrm{r}]_{\mathrm{p}}} \quad \begin{array}{l} \mathrm{l} = \mathrm{C}|_{\mathrm{p}} \\ \mathrm{l} \succ \mathrm{r} \\ \mathrm{C} \succ \mathrm{l = r} \end{array}$$

Where D is in fact always a path-clause-prefix.
The reason we can discard the right premise is that it satisfies the redundancy condition:
$\mathrm{D} \vee \mathrm{l = r}, \mathrm{D} \vee \mathrm{C}\,[\mathrm{r}]_{\mathrm{p}} \models \mathrm{D} \vee \mathrm{C}$ (as $\mathrm{l} = \mathrm{C}|_{\mathrm{p}}$)
And
$\left\{\mathrm{D} \vee \mathrm{l = r}, \mathrm{D} \vee \mathrm{C}\,[\mathrm{r}]_{\mathrm{p}}\right\} \prec \mathrm{D} \vee \mathrm{C}$
This allows us to propagate a simplified clause over a join with a clause-path-prefix as long as all branch literals are smaller than all other literals, and as long as the equality is also represented at the join.

Notation

For a node n, we use $[\mathrm{t}]_{\mathrm{n}}$ to represent the EC-node in g_{n} for the term t when $\mathrm{t} \in \mathbf{terms}(\mathrm{g}_{\mathrm{n}})$. In some cases we write $[\mathrm{s}, \mathrm{t}]_{\mathrm{n}}$ when we know that
$\mathrm{s}, \mathrm{t} \in \mathrm{g}_{\mathrm{n}} \wedge [\mathrm{s}]_{\mathrm{n}} = [\mathrm{t}]_{\mathrm{n}}$. We extend the notation for atom-, literal- and clause-ECs - for example: $[\mathrm{s} \neq \mathrm{t}]_{\mathrm{n}}$ denotes the atom-EC $[\mathrm{s}]_{\mathrm{n}} \neq [\mathrm{t}]_{\mathrm{n}}$ and $[\mathrm{C} \vee \mathrm{s} = \mathrm{t}]_{\mathrm{n}}$ can be written as $[\mathrm{C}]_{\mathrm{n}} \vee [\mathrm{s} = \mathrm{t}]_{\mathrm{n}}$ or $[\mathrm{C}]_{\mathrm{n}} \vee [\mathrm{s}]_{\mathrm{n}} = [\mathrm{t}]_{\mathrm{n}}$.

The bluesub-term-closure of an EC-node is the downward closure of the EC-node w.r.t. GFAs - formally (using the relation transitive closure TC):
$\mathbf{stc}(\mathrm{t}) \triangleq \{\mathrm{s} \mid \mathrm{TC}(\{(\mathrm{u}, \mathrm{v}) \mid \mathrm{u} = \mathrm{v} \vee \exists \mathrm{f}(\overline{\mathrm{w}}) \in \mathrm{u}, \mathrm{i} \cdot \mathrm{v} = \mathrm{w}_{\mathrm{i}}\})(\mathrm{t}, \mathrm{s})\}$ And similarly for GFAs and atom-, literal-, and clause-ECs. For two EC-nodes s,t of the same EC-graph, we use $\mathrm{s} \trianglelefteq \mathrm{t}$ as follows: $\mathrm{s} \trianglelefteq \mathrm{t} \triangleq \mathrm{s} \in \mathbf{stc}(\mathrm{t})$.

CFG-Node state

The state we maintain at each CFG-node n is as follows:

- An EC-graph g_{n} that includes all terms in all clauses in n, as described in chapter 3, including the sources function that connects it to the EC-graphs of direct predecessors.
- The sets $\texttt{done}_{\mathrm{n}}$ and $\texttt{todo}_{\mathrm{n}}$ of all non-unit and dis-equality clause-ECs derived at n (unit equalities are represented solely by g_{n}), as described in the current chapter, except that we use clause-ECs rather than clauses.
- The set $\texttt{todoeq}_{\mathrm{n}}$ of equalities derived at n to be `assumed` in g_{n} - we try to collect as many equalities as possible before performing congruence closure, in order to reduce the number of congruence closure operations. Rather than perform `assumeEqual` for each pair in the queue, we have one operation that uses `enqueueMerge` from figure 3.3 to enqueue all equalities from $\texttt{todoeq}_{\mathrm{n}}$ and then we call `mainLoop`. When calling `update` we also optionally enqueue all equalities from the queue.

- A cache for clause requests as described in section 4.1, except that requests are sets of term-EC-nodes rather than sets of terms and hence the cache is also sets of EC-nodes (for simplification requests also a set of atom-ECs)

Main changes from the basic algorithm

Our combined EC-graph with non-unit algorithm is based on the non-unit algorithm from section 4.1, using clause-ECs and term-ECs rather than clauses and terms, with the following differences:

- Unit equalities are represented using GTs (term-ECs) and hence there is no need to perform unit superposition. We do perform superposition where the right premise is an equality encoded in a GT.
- The `todoeq` queue is maintained seprately and all the equalities in the queue are added to the EC-graph together.
- Clause-ECs are updated after each `assumeEqual` and `update` operations, so they are always in normal form w.r.t the rewrite system represented by the EC-graph.
- Requests are represented using GFAs rather than terms, are translated using source edges and hence they are an over-approximation.
- At joins we complete missing equalities that the EC-graph cannot handle as described in section A.2.1

A.2.1 Non-unit fall-back at joins

As we have seen in chapter 3, some equality information is not representable using the unit-ground fragment and some information is representable in that fragment but the representation is inefficient. As our requests are composed of GFAs and GTs rather than terms, if a source chain is broken because of a join, a request might not reach its destination.

Our solution to the problem is to user the ordering $\succ$ to orient the choice of which terms to add at a join, so that we allow adding a term to the join not only if it is represented in both predecessors, but also if it is the minimal representation of a term in one predecessor. For non-minimal terms we add a relativized equality with the minimal term.

Consider the example in figure A.5 - the source chain between $[f(f(c))]_4$ and $[f(f(c))]_0$ is broken.

Our idea is as follows: the join CFG-node n selects a join representative - $\mathbf{jrep}_i(t)$ for each GT t in its EC-graph and each joinee p_i, so that the join representative is a GT in g_n which represents the minimum of the EC of t in the congruence $\mathbf{eqs}(n) \cup \mathbf{eqs}(p_i)$. In the above example, the join representative for $[c]_3$ in n_1 is $[b]_3$ (written $\mathbf{jrep}_1([c]_3) = [b]_3$), while for n_2, $\mathbf{jrep}_2([c]_3) = [c]_3$.

Remember that the invariant for the EC-graph for a join requires that if a term is represented at both joinees, it cannot be an RGFA in the join - in the above example, c is represented in both joinees and hence, when requested by n_4, we must add it to n_3. However, when n_4 requests $f([c]_3)$, n_3 is allowed to

```
if (P)
    $n_0$ :
    assume $C \vee \underline{f(f(c))} = a$
    $n_1$ :
    assume $\underline{c} = b$
else
    $n_2$ :
    assume $\underline{g(c)} = a$
$n_3$ :
$n_4$ :
assume $C \vee \underline{f(f(c))} \neq a$
assume $C \vee \underline{f(g(c))} \neq a$
```

Figure A.5: Example for non-unit fall-back for equality propagation
The ordering is $f(f(c)) \succ f(c) \succ f(b) \succ c \succ b \succ a$ The terms f(c),g(c) each occur on only one side of the join n_3, hence not added to the EC-graph of the join by our join algorithm.
The terms $f(c), f(f(c))$ are not represented at n_3.
The request for $f([f(c)]_4)$ is not propagated to n_3.

reject it, as the term f(c) is not represented in n_2. Remeber that this ability to reject requested terms is crucial to ensure termination and also performance.

We modify this invariant as follows - a join must add a requested term (as a GT) if it is represented in both joinees *or if it is a join representative for one of the joinees.* In our example, $[c]_3$ is not a join representative for n_1, and hence we reject $f([c]_3)$. Instead, for each non-join-representative GT, we add a (relativized) equality with the join representative - in our case, we add the clause-EC $[\neg P]_3 \vee [c]_3 = [b]_3$. For $g([c]_3)$, $[c]_3$ is the join representative for n_2 and g(c) is represented in n_2 and hence the GT $[g(c)]_2$ is added to the join EC-graph g_3.

Now, n_4 requests the GFA $f([c]_4)$ and receives the propagated clause-EC $[\neg P]_4 \vee [c]_4 = [b]_4$ to derive $[\neg P]_4 \vee [C]_4 \vee [f(f(b))]_4 \neq [a]_4$. Now n_3 must add $[f(b)]_3$ as $[b]_3$ is a join representative for n_1 and as f(b) is represented at n_1. Similarly, $[f(f(b))]_3$ is added to the join and now we have a source chain between $[f(f(b))]_4$ and $[f(f(b))]_0$.

Our solution ensures that we do not miss any inference from the superposition calculus because of broken source chains, at the price of generating more clauses in some cases as above - for f(g(c)) we did not generate extra clauses while for f(f(c)) we did.

The source chain invariant: Using the above algorithm, we ensure an invariant for source chain in the presence of joins which is weaker than the sequential one, but sufficient for completeness - if a term is represented in two CFG-nodes that share a path, they are either connected by a source chain or one of their sub-term is connected by an extended source chain - a chain of source edges and oriented relativized equalities at joins. In the above example, for the pair $[f(f([c]))]_4$ and $[f(f([b, c]))]_0$, the sub-term is c and the extended source chain is the source edge $[c]_4$ to $[c]_3$, then the equation $\neg P \vee [c]_3 = [b]_3$, then the source edges $[b]_3$ to $[b, c]_1$ to $[c]_0$.

Finding join representatives: We find join representatives in a bottom-up manner - we look at **stc**(t) bottom up (from constants) and for each GT find a minimal representative and add the relevant relativized equations to the join.

As opposed to ordering based on depth, we have to process GTs according to the order $\prec$ and not simply bottom up by depth - if $c \succ f(f(a)) \succ f(a) \succ a$, and at the joinee $c = f(f(a))$ and we are looking for the join representative of g(c), we must process GTs in the order [a], [f(a)], [f(f(a))], [c] and then we can see that the join representative for [c] is in fact [f(f(a))] hence the join representative for g(c) is [g(f(f(a)))].

Incrementality:

When the EC-graphs of the join and joinees are modified (e.g. nodes added or merged), the normal form of each join EC-node might change (only to a smaller normal form). We have not implemented our algorithm incrementally, so that if a GT or one of its sub-terms is merged, or if one of its sources is modified, we recalculate the join representatives for the new terms. As we remember the join representatives for sub-terms, this recalculation is often not very expensive - the bottom up traversal only processes modified ECs, but does not re-use the previously calculated join representatives for modified ECs.

On the other hand, adding terms or clauses, or merging super-terms anywhere in the CFG does not change the definition of a join representative, hence our choice is relatively stable during the execution of the algorithm.

Bibliography

[AGC12a] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2012.

[AGC12b] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2012.

[BB14] Armin Biere and Roderick Bloem, editors. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[BBG$^+$60] John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the algorithmic language ALGOL 60. *Commun. ACM*, 3(5):299–314, 1960.

[BBW15] Peter Baumgartner, Joshua Bax, and Uwe Waldmann. Beagle - A hierarchic superposition theorem prover. In Felty and Middeldorp [FM15], pages 367–377.

[BCD$^+$05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[BCD$^+$11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[BDdM08] Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura. Accelerating lemma learning using joins. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'2008)*, November 2008. Short paper.

[BDS02a] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2002.

[BDS02b] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of shostak's method for combining decision procedures. In Alessandro Armando, editor, *Frontiers of Combining Systems, 4th International Workshop, FroCoS 2002, Santa Margherita Ligure, Italy, April 8-10, 2002, Proceedings*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2002.

[BEL$^+$01] Matthias Baaz, Uwe Egly, Alexander Leitsch, Jean Goubault-Larrecq, and David Plaisted. Chapter 5 - normal form transformations. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 273 – 333. North-Holland, Amsterdam, 2001.

[BG91] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. Research Report MPI-I-91-208, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, September 1991. Revised version in the Journal of Logic and Computation 4, 3 (1994), pp. 217–247.

[BG94a] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[BG94b] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[BG15] Nikolaj Bjørner and Arie Gurfinkel. Property directed polyhedral abstraction. In Deepak D'Souza, Akash Lal, and Kim Guldstrand

Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 263–281. Springer, 2015.

[BGKK13] Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. Tree interpolation in vampire. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 173–181. Springer, 2013.

[BGKL13] Nikolaj Bjørner, Arie Gurfinkel, Konstantin Korovin, and Ori Lahav. Instantiations, zippers and EPR interpolation. In Kenneth L. McMillan, Aart Middeldorp, Geoff Sutcliffe, and Andrei Voronkov, editors, *LPAR 2013, 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, December 12-17, 2013, Stellenbosch, South Africa, Short papers proceedings*, volume 26 of *EPiC Series*, pages 35–41. EasyChair, 2013.

[BGW94] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Refutational theorem proving for hierarchic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.

[BJ15] Maria Paola Bonacina and Moa Johansson. Interpolation systems for ground proofs in automated deduction: A survey. *J. Autom. Reason.*, 54(4):353–390, April 2015.

[Bjö09] Magnus Björk. First order stålmarck. *Journal of Automated Reasoning*, 42(1):99–122, 2009.

[Bon13] Maria Paola Bonacina, editor. *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*. Springer, 2013.

[BPT12] Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model evolution with equality - revised and implemented. *J. Symb. Comput.*, 47(9):1011–1045, 2012.

[Bra11] Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[Bra12] Aaron R. Bradley. Understanding IC3. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference,* *Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2012.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [RV01b], pages 445–532.

[BT00] Leo Bachmair and Ashish Tiwari. Abstract congruence closure and specializations. In David A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2000.

[BT12] Peter Baumgartner and Evgenij Thorstensen. Instance based methods — A brief overview. *CoRR*, abs/1202.6148, 2012.

[BVB$^+$13] Sam Bayless, Celina G. Val, Thomas Ball, Holger H. Hoos, and Alan J. Hu. Efficient modular SAT solving for IC3. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 149–156. IEEE, 2013.

[BW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2):149–169, 2001.

[BW13] Peter Baumgartner and Uwe Waldmann. Hierarchic superposition with weak abstraction. In Bonacina [Bon13], pages 39–57.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CCKL08] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): semantic combination of congruence closure with solvable theories. *Electr. Notes Theor. Comput. Sci.*, 198(2):51–69, 2008.

[CDW14] David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2014.

[CGJ$^+$03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CL05] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 147–163, Berlin, Heidelberg, 2005. Springer-Verlag.

[CP07] Byron Cook and Andreas Podelski, editors. *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*. Springer, 2007.

[Cra57] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[DCKP12] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with triggers. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2012.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DKPW10] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In Ramakrishnan and Rehof [RR08], pages 337–340.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 125–134. FMCAD Inc., 2011.

[FGG$^+$12] Alexander Fuchs, Amit Goel, Jim Grundy, Sava Krstic, and Cesare Tinelli. Ground interpolation for the theory of equality. *Logical Methods in Computer Science*, 8(1), 2012.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In Jr. and Somenzi [JS03], pages 355–367.

[FM15] Amy P. Felty and Aart Middeldorp, editors. *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*. Springer, 2015.

[Fuc04] Alexander Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau, 2004.

[GCS11] Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient predicate abstraction of program summaries. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 131–145. Springer, 2011.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[GHW03] Harald Ganzinger, Thomas Hillenbrand, and Uwe Waldmann. Superposition modulo a shostak theory. In Franz Baader, editor, *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2003.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 235–246. ACM, 2008.

[GN07] Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. *Sci. Comput. Program.*, 64(1):97–114, 2007.

[GOR99] Erich Grädel, Martin Otto, and Eric Rosen. Undecidability results on two-variable logics. *Arch. Math. Log.*, 38(4-5):313–354, 1999.

[GT07a] Sumit Gulwani and Ashish Tiwari. Assertion checking unified. In Cook and Podelski [CP07], pages 363–377.

[GT07b] Sumit Gulwani and Ashish Tiwari. Assertion checking unified. In Cook and Podelski [CP07], pages 363–377.

[GTN04] Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2004.

[HBPG08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 283–290. AAAI Press, 2008.

[HJL99] Th. Hillenbrand, A. Jaeger, and B. Löchner. System description: WALDMEISTER – improvements in performance and ease of use. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632 of *LNAI*, pages 232–236. Springer-Verlag, 1999.

[HKMS13] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 451–476. Springer, 2013.

[IRR$^+$04] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2004.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006.

[JS03] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[KKRV16] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The vampire and the fool. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 37–48, New York, NY, USA, 2016. ACM.

[KMV11] Laura Kovács, Georg Moser, and Andrei Voronkov. On transfinite knuth-bendix orders. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 384–399. Springer, 2011.

[Kor08] Konstantin Korovin. iprover – an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning: 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, pages 292–298, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Koz77] Dexter Kozen. Complexity of finitely presented algebras. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 164–177. ACM, 1977.

[KS10] Konstantin Korovin and Christoph Sticksel. iprover-eq: An instantiation-based theorem prover with equality. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 196–202. Springer, 2010.

[KV07] Konstantin Korovin and Andrei Voronkov. Integrating linear arithmetic into superposition calculus. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2007.

[KV09] Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Schmidt [Sch09], pages 199–213.

[KW07] Daniel Kroening and Georg Weissenbacher. Lifting propositional interpolants to the word-level. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD*

2007, Austin, Texas, USA, November 11-14, 2007, Proceedings, pages 85–89. IEEE Computer Society, 2007.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, March 2005.

[LP16] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016.

[LR10] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[LW07] Michel Ludwig and Uwe Waldmann. An extension of the knuth-bendix ordering with lpo-like properties. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2007.

[Lyn59] Roger C. Lyndon. An interpolation theorem in the predicate calculus. *Pacific J. Math.*, 9(1):129–142, 1959.

[McM03a] Kenneth L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [JS03], pages 1–13.

[McM03b] Kenneth L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [JS03], pages 1–13.

[McM04] Kenneth L. McMillan. An interpolating theorem prover. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2004.

[McM05] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM08] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In Ramakrishnan and Rehof [RR08], pages 413–427.

[MS04] Markus Müller-Olm and Helmut Seidl. A note on karr's algorithm. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 1016–1028. Springer, 2004.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. Automatic verification of iterated separating conjunctions using symbolic execution. In *Computer Aided Verification (CAV)*, LNCS. Springer-Verlag, 2016. To appear.

[NO77] G. Nelson and D. C. Oppen. Fast decision algorithms based on union and find. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 114–119, Oct 1977.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[NO03] Robert Nieuwenhuis and Albert Oliveras. Congruence closure with integer offsets. In MosheY. Vardi and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *Lecture Notes in Computer Science*, pages 78–90. Springer Berlin Heidelberg, 2003.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[NR92a] Robert Nieuwenhuis and Albert Rubio. Basic superposition is complete. In Bernd Krieg-Brückner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 371–389. Springer, 1992.

[NR92b] Robert Nieuwenhuis and Albert Rubio. Theorem proving with ordering constrained clauses. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 477–491. Springer, 1992.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In Robinson and Voronkov [RV01b], pages 371–443.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158 – 167, 1978.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Classifying and solving horn clauses for verification. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2013.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[RR08] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[RSV15] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In Felty and Middeldorp [FM15], pages 399–415.

[RTdM14a] Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura. Finding conflicting instances of quantified formulas in smt. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 31:195–31:202, Austin, TX, 2014. FMCAD Inc.

[RTdM14b] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202. IEEE, 2014.

[RTG^{+}13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Bonacina [Bon13], pages 377–391.

[Rud11] Arsenii Rudich. *Automatic verification of heap structures with stereotypes.* PhD thesis, ETH Zrich, Zürich, 2011.

[RV99] Alexandre Riazanov and Andrei Voronkov. Vampire. In Harald Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[RV01a] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[RV01b] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[Sch09] Renate A. Schmidt, editor. *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*. Springer, 2009.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[SRV01] R. Sekar, I.V. Ramakrishnan, and Andrei Voronkov. Chapter 26 - term indexing. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1853 – 1964. North-Holland, Amsterdam, 2001.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on stålmarcks's proof procedure for propositional logic. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 1998.

[SS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[Vág03] Sándor Vágvölgyi. Intersection of finitely generated congruences over term algebra. *Theor. Comput. Sci.*, 300(1-3):209–234, 2003.

[VG09] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 1–8. IEEE, 2009.

[VG14] Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. In Biere and Bloem [BB14], pages 260–276.

[Vor14] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Biere and Bloem [BB14], pages 696–710.

[WZM12] Sarah Winkler, Harald Zankl, and Aart Middeldorp. Ordinals and knuth-bendix orders. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 420–434. Springer, 2012.

Curriculum Vitae

Uri Juhasz
Born in Jerusalem, Israel on January 18, 1979
Citizen of Israel

Education

2010 – 2016	**PhD in Computer Science** ETH Zurich
2005 – 2008	**Msc in Computer Science** Tel Aviv University, Israel
2001 – 2004	**Ba in Mathematics with Computer Science** Technion, Israel

Work Experience

2003 – 2009	**Hardware Verification Engineer** Intel Development Center, Haifa, Israel