WIDENING IMPACT

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of doctor of philosophy.

By
Christopher Gregory Coppins
December 2022

Abstract

McMillan's IMPACT algorithm showed how abstract models of programs could be refined on demand using interpolants derived from refuting program paths. The algorithm is formulated in terms of unwinding graphs, where each vertex in a graph is a program location paired with a formula that abstracts the state of the program variables at that location. However, the algorithm can generate unwieldy unwinding graphs for some seemingly straightforward programs. Moreover, the graphs can grow without bound, with a run of the algorithm only ending when the memory of the host is depleted. Indeed, IMPACT is a semi-algorithm: it is not guaranteed to complete on all programs to which it is applied. This thesis proposes a novel method to curtail the growth of the unwinding graphs, based on widening (calculating the limits of) formulae at the vertices of the graph.

A program is safe if there is no sequence of states down a branch of a program that leads to an error state. Safety can be observed by showing that there is no sequence of formulae down a branch of the unwinding graph that leads to an error state. Lifting safety from states to formulae provides a vehicle for introducing summaries: a sequence of formulae can summarize many sequences of states since each formula can represent many, possibly infinite, states. The rationale behind IMPACT is to compute sequences of formulae in such a way that one sequence of formulae can be subsumed within another. Subsumption is founded on the concept of covering. Covering is a binary relation on the vertices of an unwinding graph: one vertex covers another if both vertices describe the same program location yet the formula of the former is more relaxed than that of the latter. If all branches emanating from a vertex can be shown to be safe, and the vertex covers another vertex, then there is no need to develop any of the branches at the other vertex.

This thesis develops a new tactic for increasing opportunities for covering sufficient to induce termination. During a run, formulae at vertices often develop in a regular way. These formulae are typically conjuncts of similar subformulae differing by numeric offsets that are strengthened incrementally by adding another subformula (typically on each iteration of a loop). Widening is achieved by representing the conjunctive formula as a universally quantified formula, defined in terms of a variable that ranges over an interval. Then, the interval is relaxed to obtain a stronger formula which corresponds to its limit. Having reached its limit, covering relations, which were previously unstable, hold permanently, allowing IMPACT to terminate.

To represent conjunctive formulae, it is useful to split a formula into two parts: its template and its constants, both of which can be combined to reconstitute the formula. Since the template is stable, it suffices to reason about the n-ary vectors of constants in a formula to derive a summary of the formula. To this end, this thesis introduces a new abstract domain, the march domain, March^n, which summarizes sets of vectors using an interval so as to open up opportunities for widening. This domain is itself constructed compositionally from three subdomains: Int, kInt and Range^n, the latter two of which are also new. The kInt (k-interval) domain represents integers within an interval separated by a multiple, akin to k-bit strided intervals. The Range^n (range) domain represents groups of n-dimensional vectors that all lie on a line which passes through the origin. Once a summary is derived for a set of vectors, it naturally lifts to summarize a set of conjuncts as a single quantified formula. This gives a more compact representation of a formula (which can often be reduced to the first and last conjuncts). The conjunctive formula is widened by dropping a bound on the quantified variable, thereby strengthening the formula.

The novel ideas in this thesis were not developed by pen-and-paper exercises but emerged by exploring multiple runs of IMPACT using a visualization tool called IMPACTEXPLORER, another outcome of this thesis work. The regular conjunctive nature of the formulae, which is key to the whole thesis, would have gone unnoticed if it were not for the development of IMPACTEXPLORER. To conclude, this thesis makes contributions to infinite-state model checking by providing a proof-of-concept demonstrator for extending the IMPACT algorithm with widening.

Acknowledgements

First and foremost, I would like to thank my wife, Alissa, for her immense patience during the past six and a half years. She has given me the space to work and sacrificed many opportunities whilst I have been working. She has been particularly strong in raising and caring for our lovely son, August, since January.

Alissa also supported me during lockdown when I working independently while my supervisor, Andy King, was off sick and then convalescing from his illness. I wish to sincerely thank Andy for giving me his time when he felt able to, even though he was no longer employed as an academic. I feel particularly privileged to be one of the many students Andy has supervised and developed during his academic career. I would also like to thank Andy's family for providing the time and desk space on Saturday mornings, when Andy scrutinized the final drafts of the various chapters.

I have been blessed to have been loved and cared for throughout my life by my parents: by my mother, Ruth, and by my late father, Roy. My mother nurtured my interest in learning (particularly through mathematics) from a young age. I would also like thank my brother Geoff for all his love and support.

I would like to thank Alissa's parents, Glyn and Gila, and sister, Sarah, for looking after August several times recently, particular near the end of the write-up.

Thanks to Tom Seed who was likewise working on verification and abstract interpretation. He was a natural buddy for bouncing ideas off and providing feedback on my presentations as the ideas began to crystallize.

Finally, thanks to Neil Evans and Neil Grant at AWE for their reflections and thoughts on the topic at a crucial presentation in the Blacknest Library at the beginning of my thesis.

Update Dec 2024: Following the viva voce of this thesis held in March 2024, I would like to thank the examiners for their helpful comments and suggestions.

Contents

List of Figures

Chapter 1

Introduction

This thesis makes contributions to infinite-state model checking where the fundamental problem is to mechanically consider all paths through a program to show that they do not violate some predetermined verification conditions. One approach to achieving finiteness is to represent paths through the program as a tree, while representing variable assignments symbolically as formulae. A path in this setting is a sequence of pairs, where each pair is a program location coupled with a symbolic representation of program state. Program state is considered to be a map from variables to values, which can be described as a formula. A formula however can equally represent a set of maps, which is key to inducing finiteness. The tactic amounts to using one path to subsume (cover) another so that the latter does not need to be considered separately. Covering can be introduced where two paths pass through the same location yet have differing formulae. If the formula in the first path is a relaxation of the formula in the second, then the second path does not need to be developed since the first will subsume it. To increase the opportunities for applying this tactic, the formulae down a path are stripped of detail that is not pertinent to the verification conditions. This is achieved by using interpolation which relaxes a formula while still preserving its inconsistency with another. These techniques come together in the IMPACT algorithm [52] which is considered to be a major advance in infinite-state model checking. However, IMPACT is compromised by termination problems because the formulae which arise down paths can contain conjuncts of subformulae, differing

only in numeric constants. This thesis solves this long standing and hitherto overlooked problem by combining techniques in symbolic formulae manipulation with fixpoint acceleration techniques from abstract interpretation.

1.1 Scientific context

The discipline of verification is almost as old as computer science itself. To review the developments which are pertinent to this thesis, and do so in a way that supports the reader, the review is structured in a staged fashion. First, for fluidity, an executive summary is given which emphasises how one key development flows into another. Second, the developments are discussed individually to greater depth, with pointers to seminal literature, which provides the technical background for the thesis itself.

1.1.1 Executive summary

Modern verification emerged from the work of Floyd who detailed methods to establish safety and liveness properties of programs. Hoare developed these methods to show how to verify partial correctness in a formal axiomatic system. These methods were compositional but did not lend themselves to verifying large systems.

Model checking was conceived in the seventies as an automated way of verifying systems. Languages such as LTL and CTL emerged for expressing desired properties of a system in tandem with algorithmic ways of checking the validity of a specification encoded in these languages. The mu-calculus was proposed as a very general alternative to LTL and CTL.

Symbolic model checking was developed in the nineties as a form of model checking in which states are represented symbolically as formulae. Ordered binary decision diagrams rose to prominence as a compact way of representing propositional formulae. SAT-based model checking emerged in the late nineties as a way of scaling propositional formulae from hundreds to many thousands of variables. This was particularly important in bounded model checking (BMC) where the behaviours of a program are explored to a fixed number of steps.

Interpolation methods were proposed in the noughties for BMC but is now used

as a way of relaxing constraints to obtain program invariants which are sufficient to rule out error states. Interpolation lifts model checking from finite state to infinite state systems. Contemporaneous with the development of interpolation, SMT was proposed as a generalization of SAT solving where propositional variables of a SAT instance are replaced with predicates from a theory. The classic application of SMT is in symbolic execution.

1.1.2 Technical summary

The key scientific discoveries that that informed this thesis are illustrated in a timeline given in figure 1. Although techniques to scale verification began to emerge nearly 25 years ago, and lazy abstraction soon after, there has been a dearth of work in interpolation-based model checking in the last decade. One aspect of the problem of scalability is how to compositionally reason about loops: the problem addressed in this thesis.

Modern verification as a discipline has its roots in the seminal work of Floyd and Hoare. As early as the mid-sixties, Floyd [33] detailed methods to establish safety properties; that is, properties which always hold, and liveness properties; that is, properties which eventually hold. For example, total correctness is partial correctness, a safety property, with a guarantee of termination, a liveness property. Hoare developed these ideas to show how to verify partial correctness using axioms and inference rules in a formal axiomatic system [40]. This involved asserting what properties hold before a procedure is called with those that hold afterwards. Inference rules are then used to relate the two which are now known as pre and postconditions. It is now recognized that a key advantage of Hoare logic is that it allows for compositional verification: large systems can be verified by checking the correctness of each subsystem using its pre and postconditions without reference to the system as a whole.

1967: Floyd-Hoare Logic Although this enabled large systems to be verified in principle, in practice, the method did not immediately lend itself to larger systems. This is because proofs are delicate and highly coupled: the precondition of one procedure might be discharged by the postcondition of a procedure which

is called earlier. Any small changes to the earlier procedure that would change its postcondition may necessitate reproving the correctness of a following procedure using a different precondition. The problem is made more complex again by the need to invent invariants for loops which express properties of state variables which hold over all iterations of a loop and yet are still strong enough to establish a postcondition that holds on loop exit. Theorem provers, such as those deployed in SPARK [17], can mechanize the checking of pre and postconditions and loop invariants, but nevertheless the ingenuity of the user is still required to invent the conditions themselves.

1977: Temporal logic Model checking was conceived as a way of reducing the burden of verification by providing the user with a language for expressing the desired properties of a program and an automated algorithm for validating them. Temporal logic in its many variants emerged as the language of choice since it permits a wide range of correctness properties to be captured for both sequential and concurrent programs. Whereas Burstall [14] and Kröger [46] proposed the use of temporal logic for reasoning about programs, Pneuli, in his seminal paper [60], outlines an algorithm for checking the validity of a formula for programs of finite state. Pneuli used modal operators F (sometimes), G (always), X (next-time) and U (until) in his temporal logic to encode formulae about the future of paths. In addition to these operators, the logic was built up from recursive formulae comprised from a set of propositional variables and logical operators. Pneuli showed his form of temporal logic, which is now known as LTL (Linear Time Logic) was useful for expressing concepts such as mutual exclusion, absence of deadlock and absence of starvation.

In contrast to LTL, which expresses properties which hold on a single run of a program, the logic CTL (Computation Tree Logic) can express properties which hold on all branches (all runs), or at least one branch (one run) of a program. Lamport showed [47] that LTL and CTL are incomparable in terms of expressiveness: CTL cannot state certain fairness properties that are expressible in LTL: LTL, unlike CTL, cannot express the possibility of an event occurring in the future along some computation path. LTL and CTL and their many variants do not lend themselves to compositional reasoning which is the key to scaling verification.

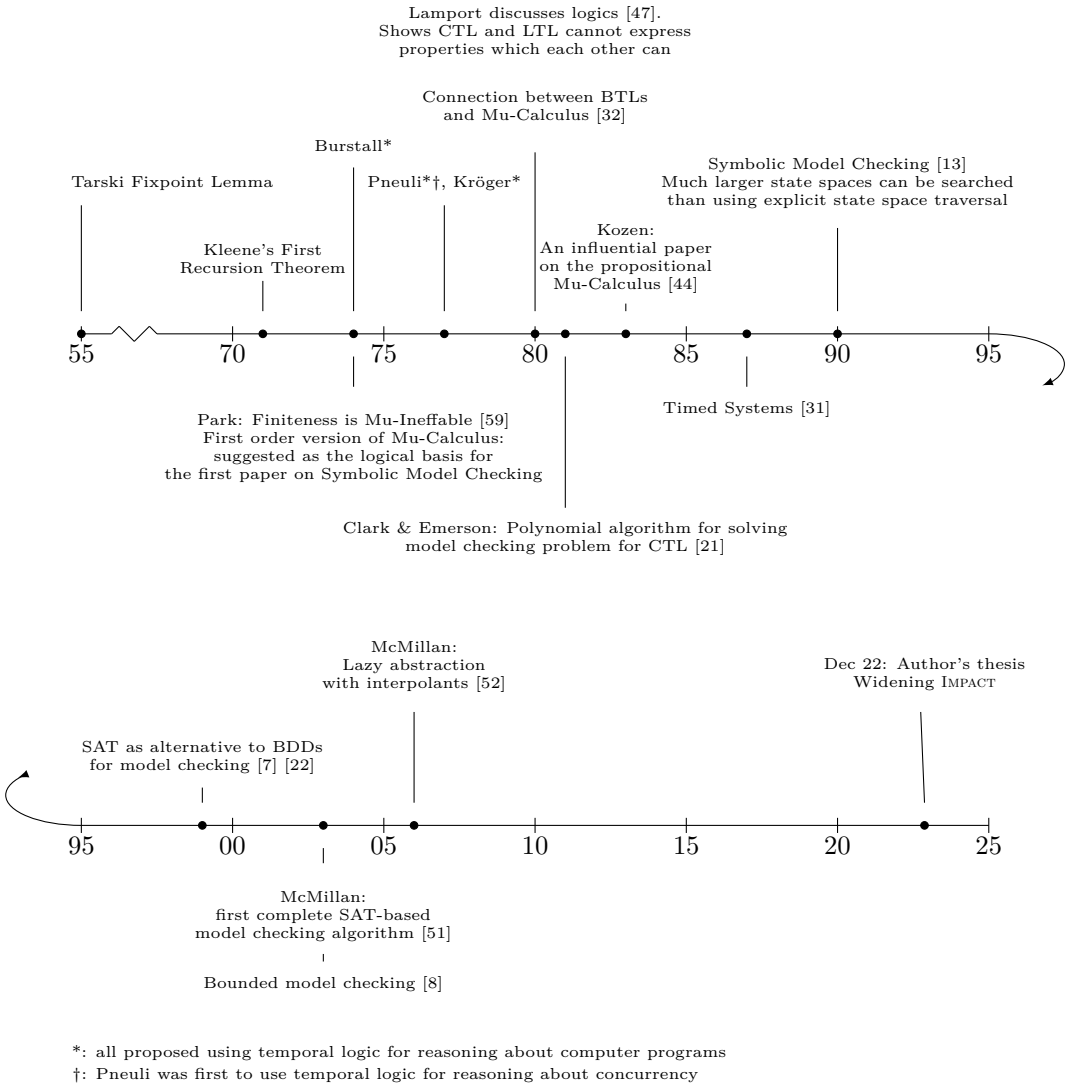


Figure 1: A timeline of key scientific discoveries

1980: The mu-calculus Rather than working in a fixed logic, the mu-calculus [59] provides a medium in which logics can themselves be expressed. Unlike temporal logics, the mu-calculus can specify properties using least fixpoints and greatest fixpoints and combinations of the two. Least fixpoints relate to terminating recursion whereas greatest fixpoints can express infinite recursion. Although mu-calculus characterizations are mathematically elegant, the intuition behind them is often less obvious than a LTL and a CTL formulation [32].

1989: Model checking finite-state concurrent systems A timed automaton [31] is an extension of a finite automaton which is enriched with a finite set of real-valued clock variables. Timed automata express systems where all clock variables progress uniformly with time until an individual clock is reset (when prescribed conditions are met). A natural representation for the state of a system is using a matrix called a difference bound matrix (DBM) which records how the value on one clock differs to the value assigned to another. When the initial state of an automaton is fixed, the state of the clocks at any node of the automaton must be drawn from a finite set of DBMs (since the entries in any cell of a DBM are themselves drawn from a finite set of differences) [31]. This is sufficient for the state space of this class of concurrent system to be finite and therefore model checking to be feasible, for small problems at least. Symbolic model checking techniques (discussed next) led to the development of more nuanced representations of differences, including clock difference diagrams [5] which represent collections of DBMs in a canonical way, in an attempt to improve efficiency of model checking by avoiding the fruitless reexploration of the search space.

1990: Symbolic model checking Symbolic model checking [13] is model checking over states which are represented symbolically as formulae. When the number of states are small, it is perfectly feasible to realise explicit state model checking by the enumeration of all transitions between all states. However, for larger systems it was not feasible to represent the transitions this way. McMillan [13] had the insight to use a symbolic representation of state transitions based on ordered binary decision diagrams (OBDDs). OBDDs allow boolean formulae to be expressed in a compact canonical form, subject to an ordering on variables, and are tractable computationally, at least when the number of variables is not excessive. OBDDs are a tree-based representation of boolean formulae where each internal node points to two sub-OBDDs according to whether the variable associated with that node evaluates to true or false. The space saving offered by OBDDs stems from the fact that sub-OBDDs which represent the same formulae can be detected thanks to the canonical representation and represented in memory only once. A dense representation of transitions can be found by encoding the transitions symbolically and then representing the formulae as a OBDD, thereby factoring out

common structure. This approach brought a whole new class of problems within reach of model checkers but also demonstrated the power of working with symbolic formulae-based representations. The dominant technique in symbolic model checking is to reason about the correctness of a program using a finite set of propositions which are chosen upfront prior to model checking. Each proposition is associated with a boolean variable so that a truth assignment to the propositions can be represented as an OBDD. OBDDs can also be used to encode transitions between adjacent points in the program, and composed to derive transitions between program points which are far apart. A truth assignment that holds on entry to a program can then be composed with a transition relation for the whole program to infer what holds on exit to the program. Termination is guaranteed because the number of propositions is finite.

1999: SAT However, even the best OBDD packages struggle to represent formulae with a support set of variables in the hundreds. The performance of OBDDs is also chaotic and unpredictable, in part because there is no polynomial algorithm for finding the smallest OBDD that represents a given formula. Consequently, Clarke [20] suggested SAT-based model checking to finesse the use of OBDDs. For example, bounded model checking unrolls the transitions of a program to derive a formula which expresses all behaviours that can be observed within a fixed number of steps. The formula is composed with the negation of a safety property and then a truth assignment to the composition represents a counterexample to the safety property under consideration. If the formula is not satisfiable then the safety property holds, at least for those unfolding steps. The number of steps is gradually increased, ruling out all possible counterexamples to that depth, until unsatisfiability checking becomes untractable. Therefore bounded model checking may miss errors beyond that level, but can generate counterexamples which uncover problematic paths through the program which otherwise would be missed. SAT-based symbolic model checking led to a step change in proving capacity beyond that of OBDDs.

2003: Interpolation Instead of exhaustive verification, the ambition of bounded model checking (BMC) [8] is to demonstrate the violation of safety properties

within a confined, finite state space. Such a state space can be obtained by unrolling a program to a prescribed depth. BMC has risen to prominence for the verification of concurrent and imperative programs since it provides counterexamples to demonstrate the existence of a bug. If all loops are bounded then BMC can even verify properties of a program. McMillan postulated the use of interpolation for BMC [51], but since then interpolation has grown to have wider impact in predicate abstraction and refinement. McMillan's insight was to show that interpolation can be used to approximate the forward image of an initial state, and do so in a way that is pertinent to discharging a verification condition. The approximation is computed using an initial state I, a final state F and $k-1$ intermediate states. Renaming (sometimes called variable priming) is applied so that these $k+1$ states do not share common variables. All constraints are partitioned into two sets A and B: A for the constraints of I and s_1 (the state following I) and B for the remainder. SAT is then used to prove (A, B) is unsatisfiable (and hence F is not violated in k steps). From its proof an interpolant P is derived which is implied by A but inconsistent with B. P is therefore implied by the conjunction of I and the first transition. Hence P approximates the forward image of I as required and moreover, with SAT, can be calculated in an efficient way [51].

After this work of forward image calculation, it was then a natural step to use interpolation to relax the constraints at intermediate states to obtain program invariants which are nevertheless sufficient to rule out error states. In the so-called lazy abstraction approach to infinite-state model checking [52], a computation tree is constructed which represents all paths through the program from an initial given state. Given an error condition, the problem is to systematically consider all paths through the tree, specifically to show they do not error, yet doing so in a way that is finitely enumerable. The innovation in lazy abstraction is to apply interpolation on the fly as the tree is constructed. Each intermediate state down a branch is relaxed whilst retaining sufficient information to rule out an error. The relaxation is calculated mirroring the construction used in BMC, but with the set A defined to be all the constraints up to and including the intermediate state. By relaxing the intermediate state, latter states in the tree are subsumed by the intermediate state, forming a cycle to finitely represent infinite branches. Interpolation-based model checking therefore extends beyond BMC to infinite-state systems.

Although McMillan's method has not been fundamentally revised until the work of this thesis, interpolation has also been proposed for interprocedural analysis. The particular setting is the verification of recursive sequential programs where a chain of calls mirror a branch through the computation tree. Interpolation is then used to produce summaries that eliminate superfluous detail yet discharge an error property and, like before, introduce cycles to finitely represent call chains. This idea is epitomised in WHALE [1] which can be seen as a direct extension of lazy abstraction to recursive programs. Since the work of this thesis strengthens lazy abstraction, the work is equally applicable to synthesizing invariants for recursive programs.

2006: SMT Satisfiability Modulo Theories (SMT) [57] is a generalization of SAT solving in which the propositional variables of a SAT instance are replaced with predicates from a given theory. SMT solvers typically support a rich set of theories so that an SMT instance can directly express many important satisfiability problems. In the dominant approach to SMT solving, the so-called lazy method, each predicate occurring in the SMT formula is mapped to a distinct propositional variable. A propositional skeleton is constructed from the SMT instance by replacing each predicate with its corresponding propositional variable. The skeleton then describes the propositional structure of an SMT instance whilst ignoring the semantics of the predicates themselves. An SMT solver deploys a SAT solver to assign truth values to the propositional variables in the skeleton. A theory solver then reinterprets the assignment to variables as an assignment to the predicates and checks for mutual consistency. If they are consistent, the SMT instance is solved, otherwise an assignment is reported which explains the root of the inconsistency. This assignment is then used to generate a clause which is added to the skeleton so that the SAT solver henceforth only generates assignments which avoid the inconsistency.

Symbolic execution is a form of model checking whose power has been multiplied by SMT solving. Symbolic execution can be seen as a generalization of testing. In testing, single paths through a program are followed by assigning concrete values to input variables. In symbolic execution the input variables are assigned symbolic (unconstrained) values. As a path is followed through a program, the

guards along the path prescribe a system of constraints on the input variables. If the constraints are unsatisfiable, the path is infeasible; if the constraints are satisfiable then the assignment to the input variables will direct execution down the path. This can be used to check whether a path can be extended to reach an error state. Examples of symbolic execution tools include EXE [16], KLEE [15] and MAYHEM [18]. Although symbolic execution is an important application of SMT, SMT dovetails with interpolation in that methods which extract an interpolant from a SAT proof of unsatisfiability can be generalized to extract interpolants from SMT proofs of unsatisfiability [52]. In fact interpolation methods have matured to the point that interpolation algorithms are now available for a rich class of theories, including: uninterpreted functors [34]; arrays, sets and multisets [43]; quantifier-free Presburger arithmetic [11] and linear Diophantine equations [42].

1.2 Objectives

The seed of the work that developed into this thesis grew from a desire to understand the IMPACT algorithm. IMPACT is applied as an automated framework to verify a program, but the first program to which the author applied it ran until the host depleted of memory. To understand why this was the case (including the possibility that the author had failed to implement IMPACT correctly), it was necessary to build a visualization tool. This would help understand how the analysis progressed and whether there was anything that would explain the memory issue.

It soon became apparent that sequences of vertices developed whose formula labels differed from another in a regular, methodical way. These vertices were generated as the program unwound an automaton loop. It was this strengthening of the labels that led the tool to keep progressing down an ever deeper path. The exhaustion of memory just reflected the non-termination of the analysis. This raises the research question whether the structure of the formulae can be exploited to obtain termination? This is addressed as the first contribution of the thesis, detailed in section 1.3.

The problem of improving termination behaviour is well known in the context of program analysis. Widening methods can finitely analyze loops without losing critical information, and loops, such as that encountered above, are the only place

where termination problems could arise. So a natural step would be to apply widening to the formulae of the vertices sufficient to prevent further rounds of strengthening and induce termination of the algorithm.

A challenge is that IMPACT does not operate over values which represent state of the store. It uses formulae to represent the relationship between the values using symbolic variables. A research question that arises is therefore how to abstract formulae to capture the properties that hold over a loop? The related question is whether such an abstraction is sufficient to induce termination of IMPACT? The second contribution of the thesis addresses this question of how to abstract formulae by introducing an abstract domain that summarizes their salient properties so as to allow multiple formulae to be represented using quantification. The third contribution is to show how quantified formulae can improve opportunities for the covering step of the IMPACT algorithm which is how termination is achieved in lazy abstraction.

Although lazy abstraction can be viewed through the lens of abstract interpretation [24], no study has shown how abstraction can actually be deployed to improve the termination behaviour of IMPACT. An important research question therefore is whether formulae abstraction is sufficient to induce termination in practice? The fourth contribution of the thesis is a case study which addresses this question head on. A wider question is to what the degree could the IMPACT algorithm be completely automated? The case study not only reflects on the need for widening, but also the extent to which widening can be automated.

The overall objective of this study is therefore to answer the question of how abstraction can be applied to the objects which arise in interpolation, namely, symbolic formulae. All contributions of the thesis collectively answer this question.

1.3 Contributions

IMPACT builds an unwinding graph where each branch in the graph represents a collection of (related) paths through a program. When the program reaches an error location by following a particular path, the sequence of formulae down the path is extracted which represents the cumulative effect of taking that path. For the error location to be unreachable, the formulae that make up the whole path

must be mutually inconsistent. This permits a sequence interpolant to be derived that explains the inconsistency. The relaxed formulae that constitute the sequence interpolant not only characterize this particular path, but other paths down the branch which are infeasible for the same reason. To derive (safety) conditions under which the error location will not be reached, the sequence interpolant is then overlaid on the unwinding graph. Then each vertex in the graph is (eventually) labelled with a formula that represents states which are safe.

IMPACT is a semi-algorithm: it is not guaranteed to terminate and the unwinding graph can grow without bound. IMPACT, however, will terminate if the program is unsafe or the program can be shown to be safe, witnessed by every leaf vertex in the unwinding graph being covered (vacuously for terminal leaves). Termination for many programs is therefore predicated on the success of covering, which in turn, depends on one formula entailing another.

As different branches in the unwinding graph are explored, new interpolants are superimposed on the unwinding graph so that formulae at vertices grow progressively stronger. This manifests itself as a conjunction of subformulae that grows monotonically, each time incorporating a requirement from the most recent interpolant. These conjunctions of subformulae often share much commonality yet typically do not entail one another. This thesis shows how this commonality can be exploited and how formulae that develop at the vertices of the unwinding graph can be compactly summarized as a single quantified formula. Furthermore, the thesis shows how the quantified formula can be strengthened by relaxing bounds on quantified variables, thereby inducing covering and accelerating IMPACT.

To the author's knowledge, few works have addressed the relationship between interpolation and abstract interpretation (widening), the only notable work being Cousot's study of interpolation [24], where he makes the technical, theoretical point that interpolation is no more than a form of narrowing (which strictly generalizes interpolation). This thesis offers not only theoretical insight but also a practical way to improve IMPACT. In summary, this work makes the following contributions:

1. This thesis reports that the formulae which arise in the unwinding graph contain conjunctions of subformulae where each subformula differs from another only by some numeric offset (reflecting the way counters increment as

loops are unfolded).

2. This thesis introduces an abstract domain that is capable of summarizing the differences between formulae, allowing multiple formulae to be compressed into a single quantified formula. The domain is first formulated on sets of integer vectors and then lifted to sets of formulae (where the vector components represent the constants at various positions in a formula).

3. This thesis shows how the termination behaviour of IMPACT can be improved by widening these quantified formulae so as to improve covering and thereby induce termination of the IMPACT algorithm. Widening in this context amounts to relaxing the ranges of quantified variables.

4. This thesis presents experimental evidence which shows how combining abstraction with interpolation can, at last, make the analysis of a case study tractable which previously manifested problematic termination behaviour.

1.4 Roadmap

This thesis is structured as follows: Chapter 2 sets out the theoretical and mathematical preliminaries that underpin the abstract domain constructions presented in its following chapter. Chapter 3 introduces the abstract domains of intervals, k-intervals, ranges, and marches which are used in summarizing and abstracting over sets of vectors of values. Chapter 4 exposes a termination problem in the classic IMPACT algorithm [52] and explains how quantified formulae can be used to induce termination. Rather than explain IMPACT with a commentary on the algorithm, a run of it is unpicked on a train controller verification problem. Only then can the termination problem can be understood and addressed. Chapter 5 formalizes the fixpoint acceleration technique that amounts to compacting conjunctive SMT formulae with quantified SMT formulae. It clarifies how quantification can be realized with the March^n abstract domain. Chapter 6 overviews the architecture and design of a tool, IMPACTEXPLORER, which applies these techniques. IMPACTEXPLORER is not merely an adjunct to the technique outlined in Chapter 4: because of the complex nature of the interactions in the original IMPACT algorithm, it is

only possible to diagnose the termination problem and therefore remedy it with the aid of sophisticated, bespoke tooling. Chapter 7 outlines promising directions for future work as well as providing a concluding discussion.

Finally, the proofs for lemmas, theorems and propositions are presented in Appendix A. The proofs underpin the results but, indeed, were instrumental in shaping the implementation, and so are an important part of the thesis, regardless of their position. The console output of running (thereby validating) all the examples in Chapter 3 through ImpactExplorer is presented in Appendix C. A set of unwinding graphs similar to those presented in Chapter 4, but where the graphs have more detail, is found in Appendix D. These graphs are designed to be viewed in the digital archive version of the thesis.

Chapter 2

Preliminaries

2.1 Introduction

To provide context for the body of the thesis, this chapter provides a gentle introduction to SMT formula solving, interpolation, abstract interpretation and the IMPACT algorithm which applies interpolation to perform abstraction in a demand-driven (lazy) fashion. For fluidity, these introductions are kept short. The chapter also introduces mathematical concepts and notation that underpin the development of the abstract domains and formulae manipulation techniques which follow.

2.2 Partial Orders

The notion of a partially-ordered set (poset) is ubiquitous. A poset is a system consisting of a set X and a binary relation $\sqsubseteq$ satisfying the following laws:

- for all x in X, $x \sqsubseteq x$ (reflexivity)
- if $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$ (transitivity)
- if $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$ (antisymmetry)

A quasi-ordered (pre-ordered) set relaxes the third requirement.

In a quasi-ordered set, if a least upper bound or a greatest lower bound of some subset exists, it may not exist uniquely, since we do not necessarily have antisymmetry for the quasi-ordering. This motivates the following:

Definition 1. A quasi-ordered set is called a pseudo-lattice iff any two elements have at least one least upper bound and at least one greatest lower bound.

Definition 2. A quasi-ordered set is called a pseudo-join-semilattice iff any two elements have at least one least upper bound.

The term pseudo-join-semilattice is of the author's own invention but is inspired by the concept of pseudo-lattice, as studied in [41].

2.2.1 Ordering by divisibility

If $a, b \in \mathbb{Z}$, and there exists $q \in \mathbb{Z}$ such that $a = qb$, then b divides a, written $b|a$. The concept of divisor is lifted here to integer vectors: if $\vec{a}, \vec{b} \in \mathbb{Z}^n$ and there exists $q \in \mathbb{Z}$ such that $\vec{a} = q\vec{b}$, then $\vec{b}$ divides $\vec{a}$, written $\vec{b}|\vec{a}$. The author also defines $\vec{a}/\vec{b} = q$ if $\vec{a} = q\vec{b}$, $q \in \mathbb{Z}$ and $\vec{b} \neq \vec{0}$.

Definition 3. If $a, b \in \mathbb{Z}$, $d \in \mathbb{Z}$ is a greatest common divisor of a and b iff:

- $d|a$ and $d|b$
- if $c \in \mathbb{Z}$ such that $c|a$ and $c|b$, then $c|d$

Definition 4. If $a, b \in \mathbb{Z}$, $m \in \mathbb{Z}$ is a least common multiple of a and b iff:

- $a|m$ and $b|m$
- if $c \in \mathbb{Z}$ such that $a|c$ and $b|c$, then $m|c$

Proposition 1. $\langle \mathbb{Z}, \sqsubseteq, \sqcup \rangle$ is a pseudo-semilattice where $\sqsubseteq$ is $|$, and $\sqcup$ is a least common multiple.

Observe $7|-7$ and $-7|7$, thus $7 \sqsubseteq -7$ and $-7 \sqsubseteq 7$ but $7 \neq -7$, hence $\langle \mathbb{Z}, \sqsubseteq, \sqcup \rangle$ is not a semilattice since the ordering is not anti-symmetric.

Lemma 1. Let $a, b, c \in \mathbb{Z}$ where $a|b$ and $a|c$. Then $a|gcd(b, c)$.

Definition 5. If $\vec{a}, \vec{b} \in \mathbb{Z}^n$, $\vec{d} \in \mathbb{Z}^n$ is a greatest common divisor of $\vec{a}$ and $\vec{b}$ iff

- $\vec{d}|\vec{a}$ and $\vec{d}|\vec{b}$

- if $\vec{c} \in \mathbb{Z}^n$ such that $\vec{c}|\vec{a}$ and $\vec{c}|\vec{b}$, then $\vec{c}|\vec{d}$

For $a, b \in \mathbb{Z}$, not both 0, then $gcd(a, b)$ is written to designate the greatest common divisor of a and b which is positive. Analogously, for $\vec{a}, \vec{b} \in \mathbb{Z}^n$, not both $\vec{0}$, then $gcd(\vec{a}, \vec{b})$ designates the greatest common divisor of $\vec{a}$ and $\vec{b}$ which has a positive leading component. The convention $gcd(0, 0) = 0$ is followed and likewise $gcd(\vec{0}, \vec{0}) = \vec{0}$. Additionally, $gcd(\vec{a}, \vec{b}) = \perp$ is written if there is no common divisor of $\vec{a}$ and $\vec{b}$ (whereas 1 is always a divisor in the scalar case). Furthermore the function $lcm(a, b)$ for $a, b \in \mathbb{Z}$, not both 0, and $lcm(\vec{a}, \vec{b})$ for $\vec{a}, \vec{b} \in \mathbb{Z}^n$, not both $\vec{0}$, is analogously defined.

Lemma 2. Let $\vec{d} \in \mathbb{Z}^n$. Then $\vec{d}|\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$ iff $\vec{d}|gcd(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n)$.

Corollary 1. Let $\vec{a}, \vec{b}, \vec{c} \in \mathbb{Z}^n$. Then $gcd(\vec{a}, \vec{b}, \vec{c}) = gcd(gcd(\vec{a}, \vec{b}), \vec{c})$.

The following proposition states that if a greatest common divisor exists for $\vec{x}$ and $\vec{v}$ and likewise for $\vec{y}$ and $\vec{v}$, then a greatest common divisor exists for all three vectors. The result can be reinterpreted as a statement on the (pseudo-)join-semilattice over n-dimensional integral vectors, where the ordering is divisibility and join is gcd.

Proposition 2. Let $\vec{x}, \vec{v}, \vec{y} \in \mathbb{Z}^n$ such that $gcd(\vec{x}, \vec{v}) \neq \perp$ and $gcd(\vec{y}, \vec{v}) \neq \perp$. Then $gcd(\vec{x}, \vec{y}, \vec{v}) \neq \perp$.

2.3 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) [45] is concerned with deciding the satisfiability of formulae over constraints drawn from a background theory. Examples of such theories include difference logic [56], uninterpreted functions [12] and bit-vectors and arrays [35], amongst others. The lazy approach [57], which is the dominant approach to SMT solving, simplifies the decision problem by separating the logical structure of a formula from its interpretation in a given theory. The logical form is described by a propositional formula for which candidate truth assignments are proposed by a SAT solver. Such assignments correspond to conjunctions of atomic constraints and their negations which can then provide a bridge to

a domain-specific solver. The approach both exploits the efficiency of modern SAT solvers whilst allowing new theories to be handled by providing a domain-specific solver solely for conjunctions of theory literals, rather than arbitrary formulae, thereby reducing the conceptual complexity of the problem. The remainder of this section discusses lazy SMT solving in more detail: first, reviewing the Boolean satisfiability problem; then second, showing how it can be extended with a background theory.

2.3.1 SAT solving

The Boolean satisfiability problem (SAT) is the problem of determining whether there is an assignment of boolean values to the variables in a propositional formula under which it evaluates to true. To illustrate, consider the propositional formula:

$$f = (u \vee \neg v) \wedge (x \vee y \vee \neg z) \wedge (\neg u \vee w \vee z)$$

defined over the set of propositional variables $X = \{u, v, w, x, y, z\}$. A truth assignment is a partial mapping $\theta : X \to \{\top, \bot\}$, which is said to satisfy f if evaluating f with these assignments yields $\top$. Observe $\theta = \{v \mapsto \bot, z \mapsto \bot, w \mapsto \top\}$ is a satisfying assignment. The goal of SAT is to decide if a propositional formula f is satisfiable, though while doing so, a satisfying assignment is generated (provided one exists).

Most modern SAT solvers are based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [30] for which a recursive formulation, adapted from [63], is presented in figure 2. The first input to the algorithm is a propositional formula f in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses where a clause is a disjunction of literals and a literal is either a variable or its negation. The second input to the algorithm is a truth assignment θ. The intuition is that the call $\mathsf{DPLL}(f, \theta)$ determines the satisfiability of the formula f under the (partial) truth assignment θ. The algorithm either returns $\bot$, indicating that f is unsatisfiable under θ, or otherwise returns an extension $\theta' \supseteq \theta$ of the satisfying assignment θ. By calling $\mathsf{DPLL}(f, \emptyset)$, where $\emptyset$ denotes the empty truth assignment, the (unconditional) satisfiability of f can thus be decided.

DPLL first attempts to derive assignments that must necessarily hold for f to

```
function DPLL(f : CNF formula, θ : truth assignment)
begin
    θ1 := θ ∪ unit_propagation(f, θ)
    if (is_conflicting(f, θ1))
        return ⊥
    else if (is_satisfied(f, θ1))
        return θ1
    else
        x := choose_free_variable(f, θ1)
        θ2 := DPLL(f, θ1 ∪ {x ↦ ⊤})
        if (θ2 ≠ ⊥)
            return θ2
        else
            return DPLL(f, θ1 ∪ {x ↦ ⊥})
        end if
    end if
end
```

Figure 2: The DPLL algorithm

be satisfiable. To illustrate, consider f as above and $\theta = \{v \mapsto \top, w \mapsto \bot\}$. Since $v \mapsto \top$, the assignment $u \mapsto \top$ must then hold to ensure the clause $(u \vee \neg v)$ evaluates to $\top$. As a consequence of $u \mapsto \top$, the assignment $z \mapsto \top$ must hold for the clause $(\neg u \vee w \vee z)$ to evaluate to $\top$. By contrast, after these two assignments, the satisfiability of the clause $(x \vee y \vee \neg z)$ still depends on two unknowns x and y, hence this clause yields no further information. It follows that θ may be extended to $\theta_1 = \theta \cup \{u \mapsto \top, z \mapsto \top\}$ without affecting the satisfiability of f under θ. This process, referred to a unit propagation, is carried out in the call $\mathsf{unit\_propagation}(f, \theta)$, which thus returns θ_1. Note that unit propagation only applies when there is a single unassigned variable in a clause. (This can be detected efficiently by watched literals [55], which merely maintains two references to two unassigned variables.)

After unit propagation, DPLL tests for early termination. First, if f contains a clause for which every literal is unsatisfiable under θ_1, then f is itself unsatisfiable and $\bot$ is returned. This situation is detected by the call $\mathsf{is\_conflicting}(f, \theta_1)$. To illustrate, consider $g = (\neg x) \wedge (x \vee y) \wedge (x \vee \neg y)$ and $\theta_1 = \{x \mapsto \bot, y \mapsto \bot\}$.

Observe each literal in the second clause is unsatisfiable under θ_1, hence g is unsatisfiable. Conversely, if every clause of f contains at least one literal that is satisfied under θ_1, then f is also satisfied under θ_1 which is thus returned. This situation is detected by the call $\mathsf{is\_satisfied}(f, \theta_1)$. To illustrate, consider $h = (\neg x \vee \neg z) \wedge (x \vee y)$ and $\theta_1 = \{x \mapsto \bot, y \mapsto \top\}$. Then, the literal $\neg x$ is satisfied in $(\neg x \vee \neg z)$ and y in $(x \vee y)$, hence h is satisfiable under θ_1.

If neither of these situations applies, a currently unassigned variable is selected by the call to $\mathsf{choose\_free\_variable}(f, \theta_1)$. A recursive call is then made with the augmented assignment $\theta_1 \cup \{x \mapsto \top\}$. This either yields a satisfying assignment or else a recursive call is made under the assignment $\theta_1 \cup \{x \mapsto \bot\}$. The return value from this call determines the satisfiability of the original formula. Termination is ensured since the number of unassigned variables strictly reduces with each recursive call. To illustrate, with $\theta_1 = \{v \mapsto \top, w \mapsto \bot, u \mapsto \top, z \mapsto \top\}$ and f as above, neither of the calls $\mathsf{is\_conflicting}(f, \theta_1)$ and $\mathsf{is\_satisfied}(f, \theta_1)$ succeeds. The unassigned variable x is thus selected and the recursive call $\mathsf{DPLL}(f, \theta_1 \cup \{x \mapsto \top\})$ is made. Since $\theta_2 = \theta_1 \cup \{x \mapsto \top\}$ is a satisfying assignment for f, θ_2 is thus returned. On top of this framework, modern SAT solvers apply a range of techniques to improve performance: for instance, static and dynamic variable orderings [55], non-chronological backtracking [49] and clause-learning [48].

2.3.2 Lazy SMT solving

To illustrate the lazy approach to SMT solving in the theory of linear real arithmetic, consider the following formula:

$$\phi = (a \leq b) \wedge (a = -1 \vee a = 1) \wedge (b = 1 \vee b = 2) \wedge \neg(-1 < 2a + 2b)$$

The set of atomic constraints in ϕ is:

$$\Sigma = \{a \leq b, \quad a = -1, \quad a = 1, \quad b = 1, \quad b = 2, \quad -1 < 2a + 2b\}$$

The propositional skeleton $e(\phi)$ of ϕ is constructed first by mapping each atomic constraint in Σ to a propositional variable. Formally, $e : \Sigma \to X$ is a bijective mapping where X is a set of propositional variables. For instance, if

$X = \{u, v, w, x, y, z\}$, then e may be defined as:

$$\begin{array}{lll} e(a \leq b) = u & e(a = -1) = v & e(a = 1) = w \\ e(b = 1) = x & e(b = 2) = y & e(-1 < 2a + 2b) = z \end{array}$$

The mapping e then lifts to a formula ϕ whose atomic constraints are drawn from Σ, by replacing each atomic constraint c in ϕ with $e(c)$, while preserving the propositional structure of ϕ. For instance, applying e to ϕ results in:

$$e(\phi) = u \wedge (v \vee w) \wedge (x \vee y) \wedge \neg z$$

To decide the satisfiability of ϕ, first a satisfying assignment θ to $e(\phi)$ is sought. If no such assignment exists, then ϕ is unsatisfiable by virtue of its propositional structure. Otherwise, θ corresponds to a conjunction of theory literals, $T(\theta, e)$, which contains the literal ℓ whenever $\theta(e(\ell)) = \top$ and the literal $\neg\ell$ whenever $\theta(e(\ell)) = \bot$. In this example,

$$\theta = \{u \mapsto \top, v \mapsto \top, w \mapsto \top, x \mapsto \top, y \mapsto \top, z \mapsto \bot\}$$

is a satisfying assignment for $e(\phi)$ and $T(\theta, e)$ is defined:

$$T(\theta, e) = (a \leq b) \wedge (a = -1) \wedge (a = 1) \wedge (b = 1) \wedge (b = 2) \wedge \neg(-1 < 2a + 2b)$$

The satisfiability of $T(\theta, e)$ can be determined by a specialized solver for linear real arithmetic. The solver will return either $\top$, to indicate $T(\theta, e)$ is satisfiable, otherwise it returns a disjunctive formula t which is the negation of a so-called unsatisfiable core. An unsatisfiable core is a conjunctive subproblem that illustrates a reason for unsatisfiability. For example, the equalities $a = -1$ and $a = 1$ are mutually inconsistent, hence the conjunct $c = (a = -1) \wedge (a = 1)$ is an unsatisfiable core. Hence the solver might return $t = \neg c = \neg(a = -1) \vee \neg(a = 1)$. Note that $e(t) = (\neg v \vee \neg w)$ is a propositional clause.

If the assignment θ did not yield a solution then a new assignment is sought. However, to avoid rediscovering the same assignment, ϕ is strengthened with a new clause, a so-called blocking clause, that serves to guide the search away from

```
function DPLL_T(f : CNF formula, e : Σ → X)
begin
    θ := DPLL(f, ∅)
    if (θ = ⊥)
        return ⊥
    else
        t := deduce(T(θ, e))
        if (t = ⊤)
            return ⊤
        else
            return DPLL_T(f ∧ e(t), e)
        end if
    end if
end
```

Figure 3: The SMT framework

previously discovered solutions. Concretely, a new formula f is defined $f = e(t) \wedge \phi$, and so here the blocking clause is $e(\neg(a = -1) \vee \neg(a = 1)) = (\neg v \vee \neg w)$. Observe that the blocking clause is unsatisfiable if v and w are both true, hence the clause prevents assignments of this form being found again. The while loop is then repeated on f and any further strengthenings of f that are inferred from inconsistent assignments during the search. Eventually, either the theory solver succeeds on an assignment, hence ϕ is satisfiable, or else the formula f becomes unsatisfiable, in which case ϕ is unsatisfiable.

Figure 3 presents an algorithm, DPLL_T, that formalizes this approach, based on a recursive reformulation of Algorithm 3.3.1 from [45]. The algorithm is parameterized by a theory T and accepts two inputs. The first is a propositional formula f, initially the propositional skeleton of ϕ; and the second, a propositional encoding $e : \Sigma \to X$, as described above. The algorithm either returns $\top$, indicating satisfiability of ϕ, otherwise it returns $\bot$, indicating unsatisfiability. The procedure applies DPLL to discover satisfying assignments to f, and a theory-specific procedure deduce for detecting satisfiability of the conjunction of theory literals $T(\theta, e)$.

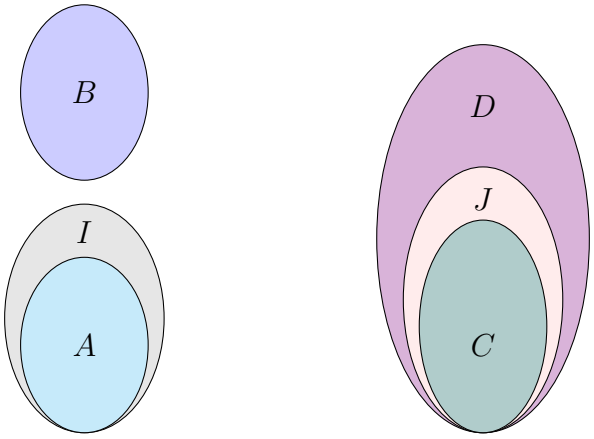


Figure 4: (a) A reverse interpolant I of A and B (b) an interpolant J of C and D

2.4 Interpolation

Given two inconsistent formulae A and B that have at least one variable in common, the problem of calculating a reverse interpolant is that of finding a formula I, that relaxes A but is still inconsistent with B, as depicted in figure 4(a). The interpolant reflects the essence of the inconsistency between A and B. The original theoretical work [29] on the existence of interpolants in the setting of first-order logic asserted that: given two formulae C and D that have at least one variable in common such that $C \models D$, then there exists an interpolant J such that $C \models J$ and $J \models D$, where the variables of J refer to variables common to both C and D. This is illustrated schematically in figure 4(b). Given A and B such that $A \wedge B \models \bot$, the existence of a reverse interpolant follows by putting $C = A$, $D = \neg B$ and $J = I$, since $C \models D$ iff $A \wedge B \models \bot$. In verification, an interpolant is usually taken to mean a reverse interpolant and this convention will be adopted henceforth.

Half a century after the Craig Interpolation Lemma [29] (which solely asserts the existence of J in the above), a multitude of techniques have been developed for algorithmically calculating interpolants. Algorithms have been devised for theories that include amongst others: uninterpreted functors [34]; arrays, sets and multisets [43]; quantifier-free Presburger arithmetic [11] and linear Diophantine equations [42]. The following sections review interpolation algorithms for propositional formulae [50, p. 105] and the theory of linear inequalities [50, p. 102]. The latter is more intuitive than the former which can only be understood with the

aid of inductive invariants and arguments (whose details are often only sketched [50]).

This introduction focuses first on interpolating propositional formulae because this algorithm is foundational: inspiring interpolation algorithms for SMT over an arbitrary theory, and even theory combinations [19]. An interpolation procedure for linear inequalities is given because it relates to themes of the thesis as well as providing an exemplar of a theory-specific algorithm.

2.4.1 Interpolating propositional formulae

Given A and B in conjunctive normal form (CNF), an interpolant can be calculated using resolution, a technique which is formulated as a system of proof rules [50]. Each proof rule takes the form $(A, B) \vdash \Theta[\phi]$, where A and B are sets of clauses, Θ is a clause and ϕ is a propositional formula (not necessarily in CNF).

Four rules are sufficient for deriving any interpolant: two rules for introducing hypotheses, and two resolution rules. These rules maintain two invariants, which are in turn defined in terms of two operators on clauses: $\Theta \setminus B$ (resp. $\Theta \downarrow B$) is a clause derived from Θ by removing (resp. keeping) those literals, p or $\neg p$, for which p occurs in B. For example, if $\Theta = (a \vee c)$ and $B = d, (a \vee b)$ then $\Theta \setminus B = c$ and conversely, $\Theta \downarrow B = a$. The two inductive invariants are as follows: i) $A \models \phi \vee (\Theta \setminus B)$ and ii) $B, \phi \models (\Theta \downarrow B)$. The force of these two invariants is that when $\Theta = \bot$, then $A \models \phi$ and $B, \phi \models \bot$, hence ϕ is an interpolant.

2.4.1.1 HYPC-A and HYPC-B

The hypothesis rules which follow distinguish between whether the hypothesis is drawn from A or B:

$$\frac{\Theta \in A}{(A, B) \vdash \Theta[\Theta \downarrow B]}\ \text{HYPC-A} \qquad \frac{\Theta \in B}{(A, B) \vdash \Theta[\top]}\ \text{HYPC-B}$$

To see that the first invariant holds for HYPC-A, observe $A \models \Theta = (\Theta \downarrow B) \vee (\Theta \setminus B) = \phi \vee (\Theta \setminus B)$, where $\phi = \Theta \downarrow B$. The second invariant holds immediately. The first and second invariants hold for HYPC-B vacuously.

2.4.1.2 RES-A

The resolution rules distinguish between whether an atomic predicate p occurs in B or not. The rule RES-A is applicable when the predicate is unique to A:

$$\frac{(A,B) \vdash (p \vee \Theta)[\phi], \quad (A,B) \vdash (\neg p \vee \Theta')[\phi'], \quad p \text{ not occurs in } B}{(A,B) \vdash (\Theta \vee \Theta')[\phi \vee \phi']} \text{ RES-A}$$

To observe the first invariant holds, suppose $A \models \phi \vee ((p \vee \Theta) \setminus B)$ and $A \models \phi' \vee ((\neg p \vee \Theta') \setminus B)$ where p does not occur in B. Then, $A \models \phi \vee p \vee (\Theta \setminus B)$ and $A \models \phi' \vee \neg p \vee (\Theta' \setminus B)$. Hence, by resolution $A \models (\phi \vee \phi') \vee ((\Theta \vee \Theta') \setminus B)$. To observe the second invariant holds, suppose $B, \phi \models (p \vee \Theta) \downarrow B = p \vee (\Theta \downarrow B)$ and $B, \phi' \models (\neg p \vee \Theta') \downarrow B = \neg p \vee (\Theta' \downarrow B)$. Hence:

$$\begin{array}{rcl} B, \phi \wedge \phi' & \models & p \vee (\Theta \downarrow B), \neg p \vee (\Theta' \downarrow B) \\ & \models & (\Theta \downarrow B) \vee (\Theta' \downarrow B) \\ & = & (\Theta \vee \Theta') \downarrow B \end{array}$$

2.4.1.3 RES-B

The following rule, RES-B, applies resolution for when RES-A is not applicable:

$$\frac{(A,B) \vdash (p \vee \Theta)[\phi], \quad (A,B) \vdash (\neg p \vee \Theta')[\phi'], \quad p \text{ occurs in } B}{(A,B) \vdash (\Theta \vee \Theta')[\phi \wedge \phi']} \text{ RES-B}$$

The conclusion may seem unexpected but observe that if $A \models \phi \vee ((p \vee \Theta) \setminus B)$ and $A \models \phi' \vee ((\neg p \vee \Theta') \setminus B)$ where p occurs in B, then $A \models \phi \vee (\Theta \setminus B)$ and $A \models \phi' \vee (\Theta' \setminus B)$. Hence:

$$\begin{array}{rcl} A & \models & \phi \wedge (\phi' \vee (\Theta' \setminus B)) \vee (\Theta \setminus B) \\ & = & (\phi \wedge \phi') \vee (\phi \wedge (\Theta' \setminus B)) \vee (\Theta \setminus B) \end{array}$$

Similarly, $A \models (\phi \wedge \phi') \vee (\phi' \wedge (\Theta \setminus B)) \vee (\Theta' \setminus B)$. Thus:

$$A \models (\phi \wedge \phi') \vee (\Theta \setminus B) \vee (\Theta' \setminus B) = (\phi \wedge \phi') \vee ((\Theta \vee \Theta') \setminus B)$$

The second invariant follows by resolution. Suppose $B, \phi \models (p \vee \Theta) \downarrow B$ and $B, \phi \models (\neg p \vee \Theta') \downarrow B$, then $B, \phi \models (\Theta \vee \Theta') \downarrow B$ by resolution since p occurs in B.

2.4.1.4 Example

To illustrate applying these rules, consider $A = \neg c, (a \vee c), \neg b$ and $B = \neg a \vee b$. To derive an interpolant for (A, B), first, two hypotheses from A are introduced:

$$\frac{}{(A,B) \vdash \neg c[\bot]}\ \text{HYPC-A} \qquad \frac{}{(A,B) \vdash (a \vee c)[a]}\ \text{HYPC-A}$$

They are combined, resolving on c:

$$\frac{(A,B) \vdash \neg c[\bot], \quad (A,B) \vdash (c \vee a)[a]}{(A,B) \vdash a[a]}\ \text{RES-A}$$

Next, the remaining hypotheses are introduced from A and B:

$$\frac{}{(A,B) \vdash \neg b[\neg b]}\ \text{HYPC-A} \qquad \frac{}{(A,B) \vdash (\neg a \vee b)[\top]}\ \text{HYPC-B}$$

Resolving on b gives:

$$\frac{(A,B) \vdash \neg b[\neg b], \quad (A,B) \vdash (b \vee \neg a)[\top]}{(A,B) \vdash \neg a[\neg b]}\ \text{RES-B}$$

Finally, a resolution on a gives:

$$\frac{(A,B) \vdash a[a], \quad (A,B) \vdash \neg a[\neg b]}{(A,B) \vdash \bot[a \wedge \neg b]}\ \text{RES-B}$$

As $\bot$ has been derived, the interpolant $a \wedge \neg b$ can be read off, completing the procedure. In general, the number of steps required to calculate such an interpolant is linear in the size of the proof tree.

2.4.2 Interpolating linear inequalities

Having shown how to derive interpolants for propositional formulae, this section shows how to derive interpolants over linear inequalities where the interpolant

takes the form $0 \leq x$ such that x is a linear expression. To derive an interpolant, proof rules are introduced to derive refutations of inconsistent formulae, along with their interpolants. Proof rules take the form $(A, B) \vdash 0 \leq x[x']$ where A and B are sets of inequalities, and x and x' are linear expressions. The intuition is that $0 \leq x$ is a linear combination of A and B, where x' represents the contribution in the linear combination from A. To derive the interpolant, two classes of rules are introduced: first, hypothesis introduction rules, and second, a combination rule. The hypothesis introduction rules that follow distinguish between whether the hypothesis is from A or B:

$$\frac{(0 \leq x) \in A}{(A, B) \vdash 0 \leq x[x]}\ \text{HYPLEQ-A} \qquad \frac{(0 \leq x) \in B}{(A, B) \vdash 0 \leq x[0]}\ \text{HYPLEQ-B}$$

The combination rule is as follows:

$$\frac{(A, B) \vdash 0 \leq x[x'], \quad (A, B) \vdash 0 \leq y[y'], \quad c_1 > 0, \quad c_2 > 0}{(A, B) \vdash 0 \leq c_1 x + c_2 y[c_1 x' + c_2 y']}\ \text{COMB}$$

To illustrate the derivation of an interpolant, consider the two sets of inequalities $A = (0 \leq 2x+y+3), (0 \leq -2x+y-1)$ and $B = (0 \leq -2y-3z-7), (0 \leq -2y+3z)$ which are mutually inconsistent. To evidence this fact, first, the HYPLEQ-A and HYPLEQ-B rules introduce two hypotheses which are the first inequalities from A and B. These are then scaled and summed using the COMB rule:

$$\dfrac{\dfrac{}{\vdash 0 \leq 2x + y + 3[2x + y + 3]}\ \text{HYPLEQ-A} \quad \dfrac{}{\vdash 0 \leq -2y - 3z - 7[0]}\ \text{HYPLEQ-B}}{\vdash 0 \leq 4x - 3z - 1[4x + 2y + 6]}\ \text{COMB}$$

Second, the HYPLEQ-A and HYPLEQ-B rules introduce two hypotheses, which are the second inequalities from A and B. These are again duly scaled and summed using the COMB rule:

$$\dfrac{\dfrac{}{\vdash 0 \leq -2x + y - 1[-2x + y - 1]}\ \text{HYPLEQ-A} \quad \dfrac{}{\vdash 0 \leq -2y + 3z[0]}\ \text{HYPLEQ-B}}{\vdash 0 \leq -4x + 3z - 2[-4x + 2y - 2]}\ \text{COMB}$$

Third, the results are summed to observe a contradiction:

$$\frac{\overline{\vdash 0 \leq 4x - 3z - 1[4x + 2y + 6]} \quad \overline{\vdash 0 \leq -4x + 3z - 2[-4x + 2y - 2]}}{\vdash 0 \leq -3[4y + 4]} \text{ COMB}$$

Observe that $I = 0 \leq 4y + 4$ is an interpolant for (A, B) since $A \models I$ (I is a linear combination of A) and $I \wedge B \models \bot$. Moreover, I is defined over y which is common to A and B. I can now be shown to be incompatible by adding linear combinations of the inequalities of B. Similar to above, first the HYPLEQ-A rule introduces the derived value from A, and the HYPLEQ-B rule introduces the first inequality from B:

$$\frac{\dfrac{\overline{\vdash 0 \leq 4y + 4[4y + 4]}\ \text{HYPLEQ-A} \quad \overline{\vdash 0 \leq -2y - 3z - 7[0]}\ \text{HYPLEQ-B}}{\vdash 0 \leq -6z - 10[4y + 4]}\ \text{COMB}}{}$$

Second, the HYPLEQ-A brings in the derived inequality from A and the HYPLEQ-B rule introduces the second inequality from B. These are then scaled and summed using the COMB rule:

$$\frac{\dfrac{\overline{\vdash 0 \leq 4y + 4[4y + 4]}\ \text{HYPLEQ-A} \quad \overline{\vdash 0 \leq -2y + 3z[0]}\ \text{HYPLEQ-B}}{\vdash 0 \leq 6z + 4[4y + 4]}\ \text{COMB}}{}$$

Third, the results are summed to observe a contradiction:

$$\frac{\overline{\vdash 0 \leq -6z - 10[4y + 4]} \quad \overline{\vdash 0 \leq 6z + 4[4y + 4]}}{\vdash 0 \leq -6[8y + 8]} \text{ COMB}$$

The interpolant $8y + 8$ that was introduced at the beginning of the section has been derived. The numbers of steps required for this derivation is linear in the size of the proof tree.

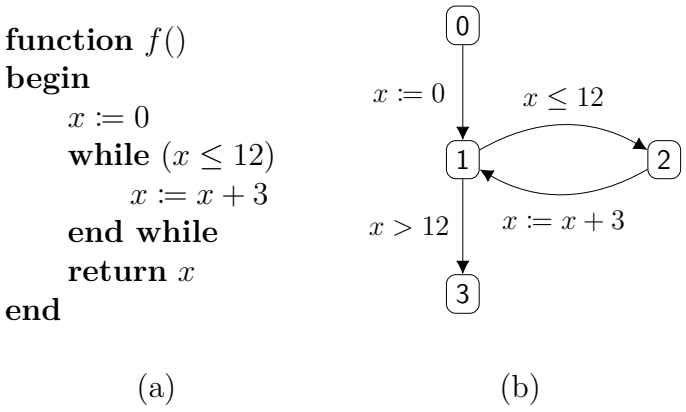


Figure 5: A toy program (a) and its flow graph (b)

2.5 Abstract Interpretation

Abstract interpretation [25] provides a rigorous basis for static program analysis. Its aim is to synthesize an abstraction of data and deploy domain operations that trace the abstraction over the paths of the program to automatically derive program invariants. By formally relating the abstract and concrete domain operations, abstract interpretation can be applied to show that an analysis is sound. Moreover, by expressing the analysis as a fixpoint of a system of equations, the outcome of an analysis can be computed iteratively. Thus abstract interpretation is not only a methodology for designing and justifying analyses: it provides a pathway for realizing them too.

The elegance of abstract interpretation is often obscured by the formality of its presentation, hence this section will motivate the essence of abstract interpretation through the toy program presented in figure 5(a). The variable x stores an (unbounded) integer and an abstract interpretation of the program aims to deduce the values x can assume at each program point. Figure 5(b) presents the control-flow graph of this program. Each node in this graph corresponds to a location in the program, either before the initial first assignment (0), before the loop guard (1), before the second assignment (2), or before the final return (3).

2.5.1 Concrete semantics

An abstract interpretation begins by modelling the concrete states of the program. First, a concrete domain $\mathcal{C}$ is introduced, which serves as a data-type to describe the concrete property of interest. In this example, the concrete domain is $\wp(\mathbb{Z})$ since, at each program point, the values assumed by x constitute a set of integers. To relate this to the execution of the program, the concrete (so-called collecting) semantics is introduced, against which the soundness of a proposed analysis can be judged. The concrete semantics defines for each program point i, the set $C_i \in \mathcal{C}$ of all concrete states obtained at that program point over all execution paths of the program. For all but the simplest of programs, the concrete semantics is incomputable, or at least prohibitively expensive to compute. For the toy program, however, it can be seen that:

$$C_0 = \mathbb{Z} \quad C_1 = \{0, 3, \ldots, 15\} \quad C_2 = \{0, 3, \ldots, 12\} \quad C_3 = \{15\}$$

where C_i is the set of values obtained by x at program point i. In particular, the value of x is undefined at program point 0 (x is assumed initialized to some unknown value), an odd number between 0 and 15 at program point 1, an odd number between 0 and 12 at program point 2 and exactly 15 at program point 3.

2.5.2 Abstract semantics

Even though the concrete semantics is in general not computable, it still may be approximated. This is achieved by introducing an abstract domain $\mathcal{A}$, whose elements are abstractions of sets of concrete program states. To illustrate this concept, the interval domain [25] will be employed. Elements of the interval domain are either intervals $[\ell, u]$ where $\ell \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ and $\ell \leq u$, or else $\bot$ indicating the empty set. Here, an interval is considered simply as a pair of bounds, which gives a compact representation of the (potentially infinite) set it represents. Note, in particular, that intervals can only represent contiguous integers, and not arbitrary sets of points, hence there is an inherent loss of information (precision) induced by this choice of domain. This is a general theme in abstract interpretation, and is the trade-off needed for tractability. A rich variety

of abstract domains have been proposed in the literature: for instance, the congruence domain [36], the polyhedral domain [27] and the octagon domain [54] and these for describing numerical properties alone. Each domain has its own unique characteristics that make it amenable for certain applications, and not for others. The challenge of abstract interpretation is in judiciously choosing, or designing, an appropriate abstraction for a given application, trading performance against precision.

The relationship between the abstract and concrete domains is made plain through a mapping $\gamma : \mathcal{A} \to \mathcal{C}$, called concretization. Intuitively, if $A \in \mathcal{A}$ is an abstract domain element, then $\gamma(A)$ is the concrete domain element it represents. For instance, in the interval domain, $\gamma([0, \infty]) = \{0, 1, 2, \ldots\}$, realizing the idea that $[0, \infty]$ describes the set of non-negative integers. Moreover, just as the concrete domain is associated with a concrete semantics, so too the abstract domain is associated with an abstract semantics. The abstract semantics specifies for each program point i, an abstract domain element $A_i \in \mathcal{A}$, which describes the set C_i of concrete states at that point. This specification is sound at program point i if $C_i \subseteq \gamma(A_i)$. For instance, for the toy program, the interval $A_1 = [0, 15]$ is a sound approximation of the set $C_1 = \{0, 3, \ldots, 15\}$ of concrete states at program point 1, since $\{0, 3, \ldots, 15\} \subseteq \gamma([0, 15]) = \{0, 1, \ldots, 15\}$. Note, however, this does not exclude the possibly of even assignments which is a consequence of abstraction.

In certain situations, a second map $\alpha : \mathcal{C} \to \mathcal{A}$ can be defined called abstraction which provides the best abstraction of C by an element of $\mathcal{A}$. This leads to the Galois connection approach to abstract interpretation [25] which provides a development methodology where γ can be synthesized from α and vice versa.

2.5.3 Fixpoint formulation

An abstract semantics for a program is typically formulated as the solution of a system of fixpoint equations which provide conditions on the sets $\gamma(A_i)$ required so that the abstract semantics be sound. For the example toy program, the following

equations could be defined:

$$\begin{array}{ll} \gamma(A_1) \supseteq \{0\} \text{ if } \gamma(A_0) \neq \emptyset & (x \coloneqq 0) \\ \gamma(A_2) \supseteq \{a \in \gamma(A_1) \mid a \leq 12\} & (x \leq 12) \\ \gamma(A_1) \supseteq \{a + 3 \mid a \in \gamma(A_2)\} & (x \coloneqq x + 3) \\ \gamma(A_3) \supseteq \{a \in \gamma(A_1) \mid a > 12\} & (x > 12) \end{array}$$

Each equation is derived from a program statement as indicated on the right. To illustrate, suppose $a \in \gamma(A_1)$. Now if A_1 is sound approximation at program point 1, then a might be a valid assignment to x at that point. Therefore, if $a \leq 12$, then the loop test ($x \leq 12$) would succeed and a would also be a valid assignment to x at program point 2, hence $a \in \gamma(A_2)$ is required for soundness. Conversely, if $a > 12$ then the loop test would fail, hence a would be a valid assignment to x at program point 3 and so $a \in \gamma(A_3)$ should hold. This justifies the second and fourth equations. Similarly, if $\gamma(A_0) \neq \emptyset$, then x might be assigned some value at program point 0, which must therefore be reachable. In this case, x would obtain the value 0 at program point 1 under the assignment $x \coloneqq 0$, hence $0 \in \gamma(A_1)$. Similarly, if $a \in \gamma(A_2)$ then a might be a valid assignment to x at program point 2, in which case $a + 3$ would be a valid assignment to x at program point 1 under the assignment $x \coloneqq x + 3$, hence $a + 3 \in \gamma(A_1)$. This justifies the first and third equations.

2.5.4 Fixpoint solution

To compute the abstract semantics, an iterative procedure is applied. Intuitively, values for each A_i are prescribed and then the semantic equations are repeatedly checked and re-evaluated. If they all hold, then a valid solution has been found which is returned. Otherwise, some A_i is updated to ensure the violated equation is satisfied, and the process is repeated. The abstract elements are initialized:

$$A_0 = [-\infty, \infty] \quad A_1 = \bot \quad A_2 = \bot \quad A_3 = \bot$$

which reflects the fact that x might take any value at program point 0, but no assignments to x have occurred at any other program point.

Now, since $\gamma(A_0) \neq \emptyset$, the first equation above must apply, hence $\gamma(A_1) \supseteq \{0\}$. But $A_1 = \bot$, hence $\gamma(A_1) = \emptyset \not\supseteq \{0\}$ and the equation is thus violated. Hence, A_1 is updated to $[0, 0]$ to ensure $\gamma(A_1) \supseteq \{0\}$, yielding:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 0] \quad A_2 = \bot \quad A_3 = \bot$$

Next, the second equation applies to show $\gamma(A_2) \supseteq \{a \in \gamma(A_1) \mid a \leq 12\} = \{0\}$. But, similar to before, $A_2 = \bot$ hence $\gamma(A_2) = \emptyset \not\supseteq \{0\}$ and the equation is violated. Hence, A_2 is updated to $[0, 0]$ to ensure $\gamma(A_2) \supseteq \{0\}$, yielding:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 0] \quad A_2 = [0, 0] \quad A_3 = \bot$$

Now, the third equation applies to show $\gamma(A_1) \supseteq \{a + 3 \mid a \in \gamma(A_2)\} = \{3\}$. But since $\gamma(A_1) = \{0\} \not\supseteq \{3\}$ the equation does not hold. In this case, it is already known that $\gamma(A_1) \supseteq \{0\}$ is necessary, hence $\gamma(A_2) \supseteq \{0, 3\}$ must hold. To ensure this, A_1 is updated to $[0, 3]$, for which $\gamma(A_1) = \{0, 1, 2, 3\} \supseteq \{0, 3\}$, hence:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 3] \quad A_2 = [0, 0] \quad A_3 = \bot$$

The most recent assignment to A_1 again violates the second equation, which is then corrected by updating A_2 to $[0, 3]$, which then leads to further relaxation of A_1. Thus a sequence of alternating updates to A_1 and A_2 repeats until finally:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 15] \quad A_2 = [0, 12] \quad A_3 = \bot$$

At this point, the second equation requires $\gamma(A_2) \supseteq \{a \in \gamma(A_1) \mid a \leq 12\} = \{0, \ldots, 12\}$ and since $\gamma(A_2) = \{0, \ldots, 12\}$, this already holds. Similarly, the third equation requires $\gamma(A_1) \supseteq \{a + 3 \mid a \in \gamma(A_2)\} = \{0, \ldots, 15\}$ and since $\gamma(A_2) = \{0, \ldots, 12\}$, this already holds as well. The first equation also applies trivially hence only the fourth equation remains to validate. In this case, it is required $\gamma(A_3) \supseteq \{a \in \gamma(A_1) \mid a > 12\} = \{13, 14, 15\}$. Since $A_3 = \bot$, this does not currently hold, so A_3 is updated to $[13, 15]$, yielding:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 15] \quad A_2 = [0, 12] \quad A_3 = [13, 15]$$

At this point, all four equations are satisfied, hence the analysis terminates. Note particularly that for each i, it holds that $C_i \subseteq \gamma(I_i)$, hence the inferred analysis is sound. However, the analysis is not fully precise. For instance, the analysis can only infer that x might take the value between 13 or 14 at program point 3. To overcome this, a more refined abstract domain must be used. For instance, a product domain construction [23] between intervals and the congruence domain [36] could be employed to reason also about the value of x modulo 3, or indeed k-intervals (to be described).

Note, the description of the analysis presented here is somewhat simplified. First, the calculation of the updates to the abstract semantics occurred by reasoning about the concretizations $\gamma(A_i)$. In practice this is not possible, hence computational procedures must be designed to compute the updates solely in terms of the abstract domain elements. This is formalized through the notion of abstract transfer functions. Second, the analysis presented here terminated rapidly. In certain situations, however, termination may not occur, or at least the number of iterations is so great that a fixpoint is impractical to compute. In this case, fixpoint acceleration techniques, for instance, widening [26] may be applied. This approach computes a sound over-approximation to the fixpoint, ensuring termination at the cost of a potential loss of precision. To an extent, this precision can be regained through the complementary technique of narrowing [26], illustrating the interplay that often arises between the fidelity of the abstraction and the fixpoint technique.

2.5.5 Widening

To illustrate a deficiency in the iterative scheme deployed for solving the fixpoint equations, consider the revision to the toy program shown in figure 6 and its fixpoint equations. Exactly as before, the system of fixpoint equations are solved iteratively to give the following intermediate assignment where A_1 is most recently assigned:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 3] \quad A_2 = [0, 0] \quad A_3 = \bot$$

The most recent assignment to A_1 again violates the second equation, which is

```
function f()
begin
    x := 0
    while (x ≤ 1024)
        x := x + 3
    end while
    return x
end
```

(a)

$$\begin{array}{ll} \gamma(A_1) \supseteq \{0\} \text{ if } \gamma(A_0) \neq \emptyset & (x := 0) \\ \gamma(A_2) \supseteq \{a \in \gamma(A_1) \mid a \leq 1024\} & (x \leq 1024) \\ \gamma(A_1) \supseteq \{a + 3 \mid a \in \gamma(A_2)\} & (x := x + 3) \\ \gamma(A_3) \supseteq \{a \in \gamma(A_1) \mid a > 1024\} & (x > 1024) \end{array}$$

(b)

Figure 6: A toy program (a) and its fixpoint equations (b)

then corrected by updating A_2 to $[0, 3]$ to give the following:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 3] \quad A_2 = [0, 3] \quad A_3 = \bot$$

Then A_1 is again violated so it is updated to give the following:

$$A_0 = [-\infty, \infty] \quad A_1 = [0, 6] \quad A_2 = [0, 3] \quad A_3 = \bot$$

This alternating sequence of updates on A_1 and A_2 will be repeated 342 times until a fixpoint is achieved. Observe that the number of iterations is sensitive to the upper bound on the guard $x \leq 1024$, which is hardly satisfactory. Worse still, if the guard was replaced by *true* then the fixpoint strategy would not even terminate. The issue of slow convergence and possible non termination motivate the introduction of widening [25].

The classical widening for intervals, which operates on two intervals to give a third, is defined thus:

$$\begin{array}{r} \bot \nabla [\ell_1, u_1] = [\ell_1, u_1] \\ [\ell_0, u_0] \nabla \bot \quad = [\ell_0, u_0] \\ [\ell_0, u_0] \nabla [\ell_1, u_1] = [\ell, u] \end{array} \quad \text{where} \quad \ell = \begin{cases} -\infty & \text{if } \ell_1 < \ell_0 \\ \ell_0 & \text{otherwise} \end{cases} \quad u = \begin{cases} +\infty & \text{if } u_1 > u_0 \\ u_0 & \text{otherwise} \end{cases}$$

The resultant interval contains both input intervals and, in general, a widening operation is monotonic in both arguments. However, the primary role of widening is to ensure rapid convergence. In the case of intervals it does so by preserving stable bounds, while erasing unstable bounds. For example, $[0,4]\nabla[0,5] = [0,\infty]$. Observe the lower bound of 0 is preserved whereas the upper bound is unstable, because $5 > 4$, hence is extended to ∞. In the example, it is sufficient for termination to widen A_1 since A_2 is defined in terms of A_1. (Alternatively A_2 could be widened since A_1 is determined by A_2.) Widening A_1 yields the following:

$$A_0 = [-\infty,\infty] \quad A_1 = [0,3]\nabla[0,6] = [0,\infty] \quad A_2 = [0,3] \quad A_3 = \bot$$

As before, the next step is to update A_2 but notice that the ∞ is tightened by the guard $a \leq 1024$.

$$A_0 = [-\infty,\infty] \quad A_1 = [0,\infty] \quad A_2 = [0,1024] \quad A_3 = [1025,\infty]$$

The focus returns to A_1 on the next iteration where widening is applied but the upper and lower bounds are both stable in the sense that $[0,\infty]$ contains $[3,1027]$, thus a fixpoint is reached where all the equations are simultaneously satisfied.

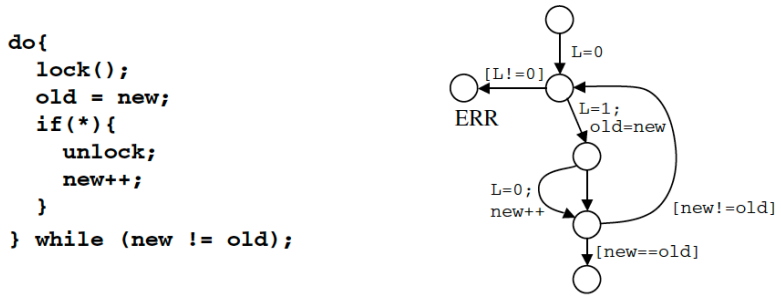
$$A_0 = [-\infty,\infty] \quad A_1 = [0,\infty]\nabla[3,1027] = [0,\infty] \quad A_2 = [0,1024] \quad A_3 = [1025,\infty]$$

In general, a widening on a poset $(P,\leq)$ is classically defined [27] as follows:

Definition 6 (widening)**.**

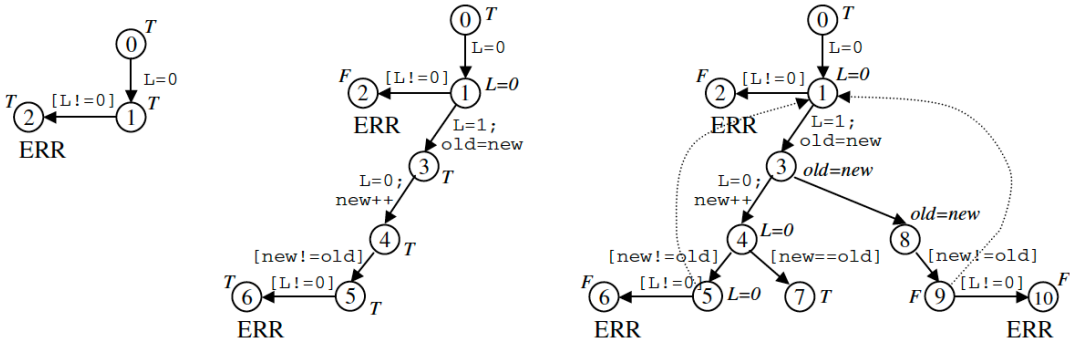
- $\forall x, y \in P,\ \ x \sqcup y \sqsubseteq x\nabla y$
- for all increasing sequences $x_0 \sqsubseteq x_1 \sqsubseteq \ldots x_n \ldots$ in P, the (widened) sequence $y_0 = x_0,\ \ y_{n+1} = y_n \nabla x_{n+1}$ is not strictly increasing (i.e. it stabilizes after a finite number of iterations)

The first requirement generalizes the monotonicity condition on intervals and the second, how termination is induced by extending unstable bounds to ∞.



(a) program fragment (b) control-flow graph

Fig. 1. A simple example program



(a) first error (b) second error (c) termination

Fig. 2. Stages of the unwinding (vertex labels in italics)

Figure 7: McMillan's demonstration of the IMPACT algorithm in [52]

2.6 The Impact algorithm

In abstract interpretation, widening drops information in order to obtain convergence, though not in a way that is informed by the verification conditions. McMillan's insight [52] was that interpolation could be used to discard superfluous information with respect to the verification conditions, which is often by itself sufficient to obtain convergence. In this section a compact overview of the IMPACT algorithm is given, reusing the example given in [52] (even reproducing Fig. 1. and Fig. 2. of [52] verbatim in our figure 7). Our narrative however seeks to tease out the intuition behind the algorithm rather than just describing it as is.

Set-up The IMPACT algorithm unwinds a control-flow graph for a program into a tree, with the objective of showing that an error state cannot be reached. To illustrate, figure 7/Fig. 1(a) shows a program whose lock and unlock functions are modelled with an integer variable L which assumes the value of 0 or 1, representing the unlocked and locked state respectively. The aim is to show that $L = 0$ whenever the function lock is reached. The set-up adopted in IMPACT is to augment the control-flow graph with transitions to the error states under consideration with the aim to show they are never enabled. The control-flow graph for the program fragment is shown in figure 7/Fig. 1(b). The example has a single error state, ERR, so a transition is inserted to the error state from immediately before the function lock is called. The transition to the error state is enabled whenever $L = 1$ and disabled otherwise, expressed as the guard $[L != 0]$ in Fig. 1(b). The edge labelled $L = 0$ denotes an assignment, hence the absence of square brackets.

High level strategy The unwinding process derives a tree, each vertex of which represents the state of the program variables at a snapshot in time during the execution of the program. The vertex is thus associated with a location: namely the location reached at that moment. The state of the program variables is represented symbolically using a formula; one formula for each vertex, which is called its label. The strategy behind IMPACT is to only represent the information pertinent to discharging the verification condition. To this end, each vertex is initially labelled with the formula *true*, whose labels are strengthened on demand, starting at the error state, propagating requirements for unreachability to earlier vertices in the unwinding graph. The labels are strengthened by conjoining them with a new requirement derived from the most recent act of interpolation. Labels thus grow stronger (but not necessarily strictly stronger) as the algorithm progresses. After the vertex labels have been strengthened along a path, then the algorithm checks for covering between the vertices of the graph. One vertex is said to cover a second vertex if the label of the second implies the label of the first and both vertices are associated with same program location. Covering is key to termination since it can prevent unwinding. In particular, unwinding can be ceased from the second vertex since the program state is subsumed by that at the first vertex. Thus if the error state is reachable from the second vertex, then it is reachable

from the first. Conversely, if the error state is not reachable from the first, it is not reachable from the second. To record this covering relationship, a covering arc is added from the second vertex to the first vertex (and the unwinding tree becomes an unwinding graph). During the unwinding process, should all unexpanded states become covered, then IMPACT terminates and the program is deemed safe.

Unwinding - Step 1 - Fig. 2(a) To illustrate how the strategy is applied in practice, we return to the example. IMPACT begins by unwinding the path that branches to the ERR immediately on entering the loop, before lock is called. This path is shown in Fig. 2(a). To show ERR is unreachable, it is required to label the error vertex *false* (since there are no states whose variable assignments satisfy *false*). A path from the root to ERR defines a sequence of formulae where the formulae are the guards and updates encountered along the path. This will be referred to henceforth as the concrete sequence since it describes the operations along the path without loss of information. Distinct from this sequence is the so-called sequence interpolant which describes states which hold along the path (with possible loss of information). The first formula in the sequence interpolant is *true* and the last is *false*, which describe the states at the root and ERR respectively. The second formula in the sequence interpolant is implied by the first formula, *true*, and the first formula from the concrete sequence. Likewise, the third formula is implied by the second formula and the second formula from the concrete sequence. This series of implied formulae is derived from a refutation of the path using a theorem prover.

Fig. 2(a) shows a path with the concrete sequence comprising of two formulae: $L = 0$ and $L \mathrel{!}= 0$. A sequence interpolant for this path takes the general form *true*, f, *false* where

$$\mathit{true} \wedge (L = 0) \models f \quad \text{and} \quad f \wedge (L \mathrel{!}= 0) \models \mathit{false}$$

In this case, it is sufficient to put $f = (L = 0)$. The vertices in the unwinding graph are updated with the strengthened formulae so that vertex 1 in the unwinding tree is strengthened from *true* to $L = 0$ and vertex 2 is strengthened from *true* to *false*. These two updates are reflected in Fig. 2(b).

Unwinding - Step 2 - Fig. 2(b) The unwinding process backtracks to explore another path. The path passes through the lock and unlock statements before again branching to ERR and is summarized in the following concrete sequence $L = 0$, $L = 1 \wedge \text{old} = \text{new}$, $L = 0 \wedge \text{new}' = \text{new}+1$, new′ != old, L != 0. Notice that primed variables are introduced to represent updated state. A sequence interpolant for this path takes the form *true*, f_1, f_2, f_3, f_4, *false* where

$$\begin{aligned} \textit{true} \wedge (L = 0) &\models f_1 \\ f_1 \wedge (L = 1 \wedge \text{old} = \text{new}) &\models f_2 \\ f_2 \wedge (L = 0 \wedge \text{new}' = \text{new} + 1) &\models f_3 \\ f_3 \wedge (\text{new}' \mathrel{!}= \text{old}) &\models f_4 \\ f_4 \wedge (L \mathrel{!}= 0) &\models \textit{false} \end{aligned}$$

Here, valid assignments are $f_1 = \textit{true}$, $f_2 = \textit{true}$, $f_3 = (L = 0)$, and $f_4 = (L = 0)$. (Note f_1 corresponds to vertex 1 in Fig. 2(b) but f_2 corresponds to vertex 3, f_3 to vertex 4 and f_4 to vertex 5). Since, after interpolation, $f_1 = f_2 = \textit{true}$ it follows there is no requirement on any variable on the path up to vertex 3 for ERR to be unreachable along this path. Because $f_1 = f_2 = \textit{true}$, the labels at vertices 1 and 3 are not strengthened. In effect, no new requirement has been added to two vertices for this path. Vertices 4 and 5 are both strengthened to $L = 0$. Now since vertex 5 and vertex 1 are both associated with the same location and the formula at vertex 5 implies the formula at vertex 1 then a covering arc is added from vertex 5 to vertex 1. Expansion of vertex 5 therefore ceases, until either such time as the label at 1 is strengthened so that vertex 1 no longer covers vertex 5, or the unwinding of the graph is complete. The strengthening of the vertices and the added covering arc are shown along the path in Fig. 2(c).

Unwinding - Step 3 - Fig. 2(c) The unwinding continues by backtracking and following the path where the while guard is not satisfied. The path ends with no subsequent states to expand. This path is shown by vertices 0, 1, 3, 4 and 7 in Fig. 2(c). It does not reach ERR so interpolation is performed, no labels are updated and the covering relations stand.

Unwinding - Step 4 - Fig. 2(c) The unwinding finishes by backtracking and following the final path to the considered where the unlock statement within the if statement is skipped and control returns to the head of the loop before branching to ERR. This path is shown in Fig. 2(c) and traverses vertices 0, 1, 3, 8, 9 and 10. Interpolation is then used to rule out the path to ERR. A sequence interpolant for this path takes the form *true*, f_1, f_2, f_3, f_4, *false* where

$$\begin{aligned} \mathit{true} \wedge (L = 0) &\models f_1 \\ f_1 \wedge (L = 1 \wedge \mathrm{old}' = \mathrm{new}) &\models f_2 \\ f_2 \wedge (\mathit{true}) &\models f_3 \\ f_3 \wedge (\mathrm{new} \mathrel{!}= \mathrm{old}') &\models f_4 \\ f_4 \wedge (L \mathrel{!}= 0) &\models \mathit{false} \end{aligned}$$

After interpolation f_4 is uniquely defined as *false* and therefore vertex 9 is thus covered by vertex 1. These updates are reflected in Fig. 2(c). As all unexpanded states are covered, the procedure terminates and the program is ruled safe with respect to the error condition.

It is interesting that the labels of one path can retain information that is superfluous on another path. For instance, the labels in the path shown in Fig. 2(b) retain information on the variable L but drop the relationship between new and old. In contrast, in Fig. 2(c), the relationship between new and old was maintained in the labels of vertices 3 and 8 whereas L was irrelevant. The labels are strong enough to rule the error state unreachable, but no more. It is also intriguing to observe that for a given program location, which itself defines a set of vertices in the final unwinding graph, then the disjunction of all the labels at those vertices provides an invariant for that program location.

Chapter 3

From Intervals to Marches

3.1 Introduction

The IMPACT algorithm [52] focuses on infinite-state model checking and therefore one would expect it to be a semi-algorithm, but, as will be seen later in the following chapter, its termination behaviour can be improved by judicious handling of conjunctive formulae. As the unwinding graph is explored, the formulae at vertices are strengthened by adding conjuncts. These conjuncts arise because of loops in the automaton: different iterations of the loop producing different constants in the conjuncts. These constants define n-dimensional sequences of points where constant increments give a stride between consecutive points, which is an n-dimensional vector. A symbolic summary of these points can be used to derive a summary of a conjunctive formula.

Strided intervals have been proposed for summarizing repeated sequences of offsets that occur when accessing memory in a loop in low-level code [3]. A k-bit strided interval, denoted $s[\ell, u]$ where s is the stride and $[\ell, u]$ is the interval, is interpreted with the aid of a concretization map γ as a set of integers:

$$\gamma(s[\ell, u]) = \{i \in [-2^{k-1}, 2^{k-1} - 1] \mid \ell \leq i \leq u, i \equiv \ell \,(\mathsf{mod}\ s)\}$$

Thus the set $\{3, 5, 9\}$ can be represented by $2[3, 9]$ since $\{3, 5, 9\} \subseteq \{3, 5, 7, 9\} = \gamma(2[3, 9])$, though it also can be represented by $2[3, 10]$. Strided intervals describe

sets of scalars by augmenting an all-encompassing range with a divisibility property that is expressed using modulo. Observe that the domain can be reconstructed using a multiplicative factor which is applied to all elements of an interval; for instance, $\{3, 5, 9\} \subseteq \{3, 5, 7, 9\} = \gamma(3 + 2[0, 3])$ for some appropriate γ (see section 3.5). This multiplicative approach generalizes to factors which are themselves n-dimensional vectors. This provides a pathway for generalizing strided intervals to higher-dimensional sets of points. This pathway also yields a way of lifting simpler abstractions to a numeric domain that itself provides the basis for abstracting conjuncts of formulae which differ in their constants. This chapter describes this development and provides a complete specification for an implementation, augmented with accompanying proofs.

The development of the March domain is structured into three (largely orthogonal) steps. First, the k-interval abstract domain is specified (in section 3.3) which represents sets of integers where each integer differs from its predecessor by a uniform step. These sets are represented by combining a classical interval abstraction (reviewed in section 3.2) with an integer multiplier that scales all the elements of the interval. A k-interval abstraction is therefore a one-dimensional abstraction – it represents sets of one-dimensional points. Second, these one-dimensional abstractions, while useful in their own right, are generalized to ranges that represent sets of n-dimensional vectors. Range abstractions (defined in section 3.4) are formed by combining k-intervals with an n-dimensional vector multiplier where the vector has integer coefficients. These abstractions are interpreted by scaling the vector by the elements of the k-interval. As a result, the n-dimensional points represented by a range abstraction lie on a line that passes through the origin. Third, to relax this geometric constraint, a constant vector offset is added to a range to give the March^n abstract domain (specified in section 3.5). In summary, an element of the March^n domain gives a symbolic representation of a sequence of n-dimensional points that sit on a line which does not necessarily pass through the origin.

Motivation This work arose from an attempt to directly program a higher-dimensional counterpart to strided intervals when it became evident that formalization was required. (Interestingly, the literature work behind this formalization exposed errors in the domain operations for intervals [58], which is surprising given the interval domain is one of the oldest and arguably most thoroughly studied abstract domain [37].) The March^n domain was formulated to faithfully model the vectors of integer values which occur in sequences of formulae which arise when IMPACT is applied to loops. A sequence of formulae can then be replaced with a single quantified formula by first extracting vectors of integer values from the coefficients of the variables in the formulae: one vector for each formula. Then, second, the k-interval component of the march abstraction is used to prescribe the integer values that a quantified variable can assume. The position (offset) vector component of the March^n domain then specifies how a value assigned to the quantified variable is then transformed into a vector of integer values. These vectors are exactly those which arise as coefficients in the original sequence of formulae.

The March^n domain is carefully formulated to facilitate this transformation: this is the primary motivation behind the domain. The domain is formulated so that the sequence of formulae can be translated into a single, clear and compact quantified formula which can be directly presented to an SMT solver. (Fuller details of how a sequence of formulae can be collapsed into a single quantified formula are given in Chapter 5; see also the discussion which follows example 40). The secondary motivation for the introducing the new constituent domains (kInt and Range^n domains) was to allow the March^n domain to be built from a set of simpler abstractions that themselves had been formulated with a suite of proven propositions and lemmas. The net result is that the March^n domain strictly generalizes strided intervals as well as avoiding its representation problems. The extent to which the March^n domain finds application outside symbolic model checking remains open to investigation but strided intervals have found application in binary analysis [3]. Therefore it would be interesting to investigate how additional expressiveness of the March^n domain helps in this setting. More generally still, it is always surprising how new applications arise for general purpose domains even decades after their conception.

3.2 The interval abstract domain: Int

Intervals are sets of contiguous integers, where the minimum and maximum values, if they exist, are called the lower and upper bound respectively. Bounded, ordered sets of integers are naturally abstracted by the least interval that contains their lower and upper bounds; this abstraction dropping information on any integer strictly in between the bounds. Sets without an upper bound can be represented by an interval with a symbolic upper bound of $+\infty$ and, likewise, $-\infty$ for sets without a lower bound. The interval abstract domain is such a natural abstraction [26] that it continues to find new applications despite being over 40 years old [37]; in this case in deriving compact representations of conjunctive formulae. The abstract domain Int is classically formalized as follows:

Definition 7. $\mathsf{Int} = \{\bot\} \cup \{[\ell, u] \mid \ell \leq u, \ell \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}\}$

The Int domain includes $\bot$ to represent the empty set $\emptyset$ which is a degenerate interval since its integers are vacuously contiguous. Conversely, the domain element $[-\infty, \infty]$ represents $\mathbb{Z}$. For uniformity of presentation, square brackets are used for both bounded and unbounded intervals. The interval domain constitutes a lattice as is formally asserted in the following proposition:

Proposition 3. $\langle \mathsf{Int}, \sqsubseteq, \sqcup, \sqcap \rangle$ is a lattice where:

- $\bot \sqsubseteq I$ for all $I \in \mathsf{Int}$ and $[\ell_1, u_1] \sqsubseteq [\ell_2, u_2]$ iff $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$.
- $[\ell_1, u_1] \sqcup [\ell_2, u_2] = [min(\ell_1, \ell_2), max(u_1, u_2)]$
- $[\ell_1, u_1] \sqcap [\ell_2, u_2] = \begin{cases} \bot & \text{if } max(\ell_1, \ell_2) > min(u_1, u_2) \\ [max(\ell_1, \ell_2), min(u_1, u_2)] & \text{otherwise} \end{cases}$

and $I \sqcup \bot = \bot \sqcup I = I$ and $I \sqcap \bot = \bot \sqcap I = \bot$.

The intuition behind join $\sqcup$ is that it finds the least interval that contains the two given intervals. Dually, meet $\sqcap$ finds the greatest interval that is enclosed by both of the given intervals. Note the side condition $max(\ell_1, \ell_2) > min(u_1, u_2)$ in the definition of meet is sometimes wrongly stated, for example in a tutorial [58], which hints at the subtlety of computing domain operations.

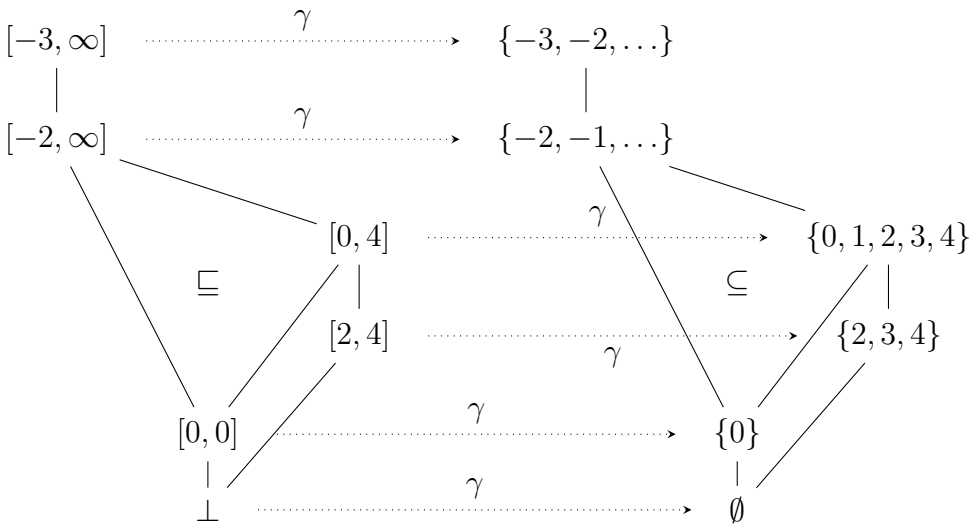


Figure 8: The correspondence between $\sqsubseteq$ and $\subseteq$ through the prism of γ.

The notation $[\ell, u]$ is merely a finite symbolic representation of an interval. To highlight the distinction between the representation and what is actually represented, a concretization map γ is introduced:

Lemma 3. Let $I_1, I_2 \in \mathsf{Int}$. Then $I_1 \sqsubseteq I_2$ iff $\gamma(I_1) \subseteq \gamma(I_2)$ where $\gamma([\ell, u]) = \{x \in \mathbb{Z} \mid \ell \leq x \leq u\}$ and $\gamma(\bot) = \emptyset$.

The lemma asserts there is a one-to-one correspondence between the $\sqsubseteq$ ordering operation on the elements of Int, which constitutes an abstract domain, and the subset ordering $\subseteq$ on their concretizations which fall within the underlying concrete domain. The mirroring between the orderings in the abstract and concrete settings validates the formulation of the ordering operation on Int.

Example 1. Figure 8 illustrates how the $\subseteq$ ordering preserves the $\sqsubseteq$ ordering after concretization has been applied for some specific intervals; namely $I_0 = \bot$, $I_1 = [0, 0]$, $I_2 = [2, 4]$, $I_3 = [0, 4]$, $I_4 = [-2, \infty]$ and $I_5 = [-3, \infty]$. Observe $I_i \sqsubseteq I_j$ iff $\gamma(I_i) \subseteq \gamma(I_j)$.

To enable Int to be extended to richer domains that are constructed by translating and scaling intervals, scalar addition and multiplication operations are introduced for intervals:

Definition 8 (Scalar addition and multiplication)**.** Let $k \in \mathbb{Z}$. Then

$$k + [\ell, u] = [\ell + k, u + k] \qquad k \cdot [\ell, u] = \begin{cases} [k\ell, ku] & \text{if } k \geq 0 \\ [ku, k\ell] & \text{otherwise} \end{cases}$$

and $k + \bot = \bot = k \cdot \bot$. Note if $k = 0$, then $0 \cdot [\ell, u] = [0, 0] = 0 \cdot [u, \ell]$.

To provide further scaffolding for the abstract domains that follow, some results pertaining to scalar addition and multiplication are now given. The first result shows that scalar multiplication is monotonic in the sense that multiplying two intervals by an integer $a \in \mathbb{Z}$ preserves the relative ordering of two intervals. The example which follows illustrates that scalar multiplication is monotonic even when the scalar is negative.

Lemma 4. Let $a \in \mathbb{Z}$ and $I_1, I_2 \in \mathsf{Int}$ where $I_1 \sqsubseteq I_2$. Then $a \cdot I_1 \sqsubseteq a \cdot I_2$

Example 2. Let $I_1 = [0, 4]$ and $I_2 = [-1, 5]$. Observe $I_1, I_2 \in \mathsf{Int}$ and $I_1 \sqsubseteq I_2$. Then $-5 \cdot I_1 = -5 \cdot [0, 4] = [-20, 0] \sqsubseteq [-25, 5] = -5 \cdot [-1, 5] = -5 \cdot I_2$.

The second result asserts that scalar multiplication is associative. The independence of the ordering of the application of the integer scalars then follows by the commutativity of integer multiplication. The corollary which follows provides two useful instances of the lemma.

Lemma 5. Let $a, b \in \mathbb{Z}$ and $I \in \mathsf{Int}$. Then $a \cdot (b \cdot I) = (ab) \cdot I = b \cdot (a \cdot I)$

Example 3. Let $a = 2, b = -3$ and $I = [1, 3]$.

- Then $a \cdot (b \cdot I) = 2 \cdot (-3 \cdot [1, 3]) = 2 \cdot [-9, -3] = [-18, -6]$
- Then $(ab) \cdot I = (2 \times -3) \cdot [1, 3] = -6 \cdot [1, 3] = [-18, -6]$
- Then $b \cdot (a \cdot I) = -3 \cdot (2 \cdot [1, 3]) = -3 \cdot [2, 6] = [-18, -6]$

Corollary 2. Let $a \in \mathbb{Z}$ and $I \in \mathsf{Int}$.

- Then $a \cdot I = -a \cdot (-1 \cdot I)$
- Then $I = -1 \cdot (-1 \cdot I)$

The final lemma of this trilogy shows a distributivity property of scalar multiplication over the join of two intervals. The accompanying example illustrates that distributivity holds even when the join of two intervals introduces elements contained in neither interval. The corollary again provides two useful instances of the lemma. The second case of the corollary hints at a form of distributivity result that appears in, and is key to, calculating the join in the domains which follow.

Lemma 6. Let $a \in \mathbb{Z}$ and $I_1, I_2 \in \mathsf{Int}$. Then $a \cdot (I_1 \sqcup I_2) = a \cdot I_1 \sqcup a \cdot I_2$

Example 4. Let $a = -3$, $I_1 = [-4, -2]$ and $I_2 = [1, 3]$. Observe $I_1, I_2 \in \mathsf{Int}$. Then $a \cdot (I_1 \sqcup I_2) = -3 \cdot ([-4, -2] \sqcup [1, 3]) = -3 \cdot [-4, 3] = [-9, 12] = [6, 12] \sqcup [-9, -3] = -3 \cdot [-4, -2] \sqcup -3 \cdot [1, 3] = a \cdot I_1 \sqcup a \cdot I_2$.

Corollary 3. Let $a, b \in \mathbb{Z}$ and $I_1, I_2 \in \mathsf{Int}$.

- Then $-1 \cdot (a \cdot I_1 \sqcup b \cdot I_2) = a \cdot (-1 \cdot I_1) \sqcup b \cdot (-1 \cdot I_2)$
- Then $c \cdot ((a/c) \cdot I_1 \sqcup (b/c) \cdot I_2) = a \cdot I_1 \sqcup b \cdot I_2$ where $c \in \mathbb{Z}$ and $c \neq 0$.

3.3 The k-interval abstract domain: kInt

Recall that the Int domain represents sets of contiguous integers. Observe that the image of the integers of an interval under scalar multiplication is non-contiguous if the scalar multiplier is not a unit, that is ± 1 in $\mathbb{Z}$. A natural generalization of intervals is therefore to pair this domain with an integer scalar multiplier, as is formalized below:

Definition 9. The k-interval domain is defined:

$$\begin{aligned} \mathsf{kInt} = \quad & \{\bot, \top\} \cup \\ & \{kI \mid k \in \mathbb{Z}, I \in \mathsf{Int}, I = [1, 1]\} \cup \\ & \{kI \mid k \in \mathbb{Z}, I \in \mathsf{Int}, k > 0, I = [\ell, u], \ell < u\} \end{aligned}$$

Note that the k-interval domain has a particular syntactic form for distinguishing k-intervals that describe a single point. Also observe that the ordering relation on k-intervals which follows relies as much on the divisibility of the scalar multipliers as on the ordering of the scaled intervals:

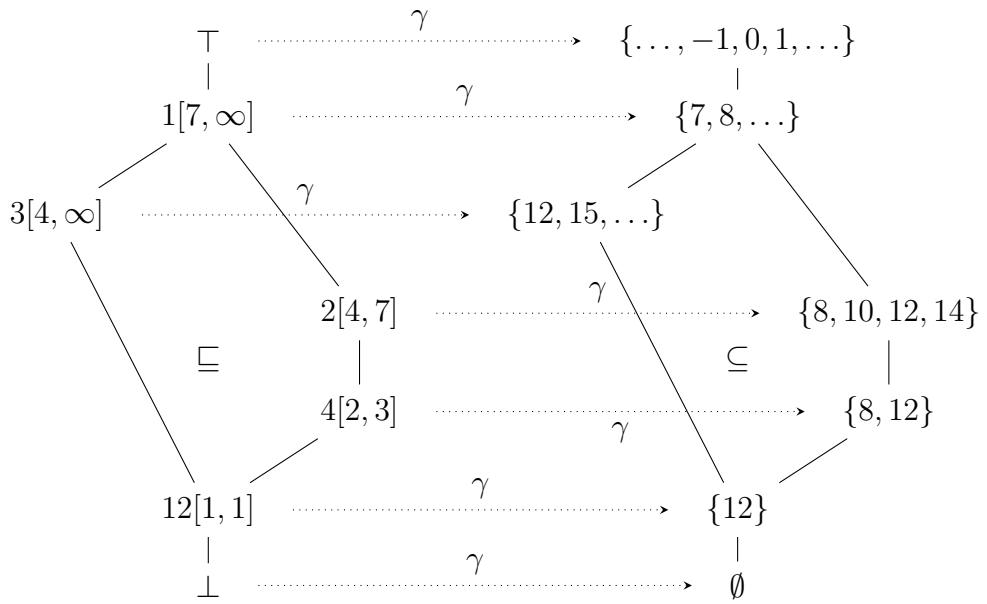


Figure 9: The correspondence between $\sqsubseteq$ and $\subseteq$ through the prism of γ.

Definition 10. The binary relation $\sqsubseteq$ on kInt is the least relation such that:

- $\bot \sqsubseteq kI \sqsubseteq \top$ for all $kI \in \mathsf{kInt}$
- $k_1 I_1 \sqsubseteq k_2 I_2$ if $k_2 | k_1$ and $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$

This ordering is validated by the following lemma which asserts that the ordering $\sqsubseteq$ mirrors the subset ordering $\subseteq$ on the image sets of the k-intervals under γ:

Lemma 7. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$. Then $k_1 I_1 \sqsubseteq k_2 I_2$ iff $\gamma(k_1 I_1) \subseteq \gamma(k_2 I_2)$ where $\gamma(kI) = \{kx \mid x \in \gamma(I)\}$, $\gamma(\bot) = \emptyset$ and $\gamma(\top) = \mathbb{Z}$.

The representation of k-intervals is actually canonical, that is, $\gamma(k_1 I_1) = \gamma(k_2 I_2)$ iff $k_1 I_1 = k_2 I_2$, where $k_1 I_1 = k_2 I_2$ iff $k_1 = k_2$ and $I_1 = I_2$. To see this, suppose $\gamma(k_1 I_1) = \gamma(k_2 I_2)$, then by lemma 7, $k_1 I_1 \equiv k_2 I_2$. Thus either $k_1 I_1 = k_2 I_2 = \bot$, or $k_1 I_1 = k_2 I_2 = \top$, or $k_2 | k_1$ and $k_1 | k_2$ and $k_1 \cdot I_1 = k_2 \cdot I_2$. Suppose $k_1, k_2 \leq 0$ or $k_1, k_2 > 0$. Then $k_1 = k_2$. So $I_1 = I_2$. Now suppose $k_1 \leq 0$ and $k_2 > 0$. Since $k_1 \leq 0$, then, by definition 9, $I_1 = [1, 1]$. Also since $k_1 | k_2$ and $k_2 | k_1$ (and $k_2 > 0$), then $k_1 = -k_2 \neq 0$. By the first result of corollary 2, $[1, 1] = I_1 = -1 \cdot I_1 = I_2$. This requires $I_2 = [-1, -1]$, but since $k_2 I_2 \in \mathsf{kInt}$, then

this cannot hold. Thus there are no cases where $k_1 \leq 0$ and $k_2 > 0$ such that $k_1 I_1 \equiv k_2 I_2$. A similar argument holds for $k_1 > 0$ and $k_2 \leq 0$. The rationale for adopting a canonical representation is that it simplifies join in that it removes the need to operate at the level of equivalence classes.

Example 5. Without the syntactic restrictions on the kInt domain observe that zero can be represented an infinite number of ways: namely, $0[1,1]$, $0[2,2]$, $0[3,3]$, ..., $0[1,2]$, $0[1,3]$, $0[1,4]$, ..., whereas an non-zero integer, for example, -2, can be represented four ways: $-2[1,1]$, $-1[2,2]$, $1[-2,-2]$ and $2[-1,-1]$ whereas a non-degenerate k-interval, such as that which represents the points 0 and 4, has only two: $-4[-1,0]$ and $4[0,1]$.

The following proposition shows how the relative ordering on two k-intervals is preserved by multiplication and division of their scalars:

Proposition 4. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$.

- If $d \in \mathbb{Z}$, then $(dk_1) \cdot I_1 \sqsubseteq (dk_2) \cdot I_2$
- If $d \in \mathbb{Z} \setminus \{0\}$, $d|k_1$ and $d|k_2$, then $(k_1/d) \cdot I_1 \sqsubseteq (k_2/d) \cdot I_2$

Some technical lemmas follow which relate a k-interval to an interval derived by scalar multiplication. The final lemmas characterize special properties of k-intervals formed from intervals that describe single points.

Lemma 8. Let $kI \in \mathsf{kInt} \setminus \{\bot, \top\}$. Then $\gamma(kI) \subseteq \gamma(k \cdot I)$

Corollary 4. Let $kI \in \mathsf{kInt} \setminus \{\bot, \top\}$. Then $kI \sqsubseteq 1(k \cdot I)$

Lemma 9. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$.

- If $k_1 I_1 \sqsubseteq k_2 I_2$ where $I_2 = [1,1]$, then $k_1 I_1 = \bot$, or $k_1 = k_2$ and $I_1 = [1,1]$.
- If $|\gamma(kI)| = 1$, then $I = [1,1]$.

3.3.1 Join

Following the usual development, $\sqcup$ is defined thus:

Proposition 5. $\langle \mathsf{klnt}, \sqsubseteq, \sqcup \rangle$ is a join-semilattice where:

$$k_1 I_1 \sqcup k_2 I_2 = \begin{cases} k_1 I_1 & \text{if } k_1 = k_2 \text{ and } I_1 = I_2 = [1,1] \\ k\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) & \text{otherwise} \end{cases}$$

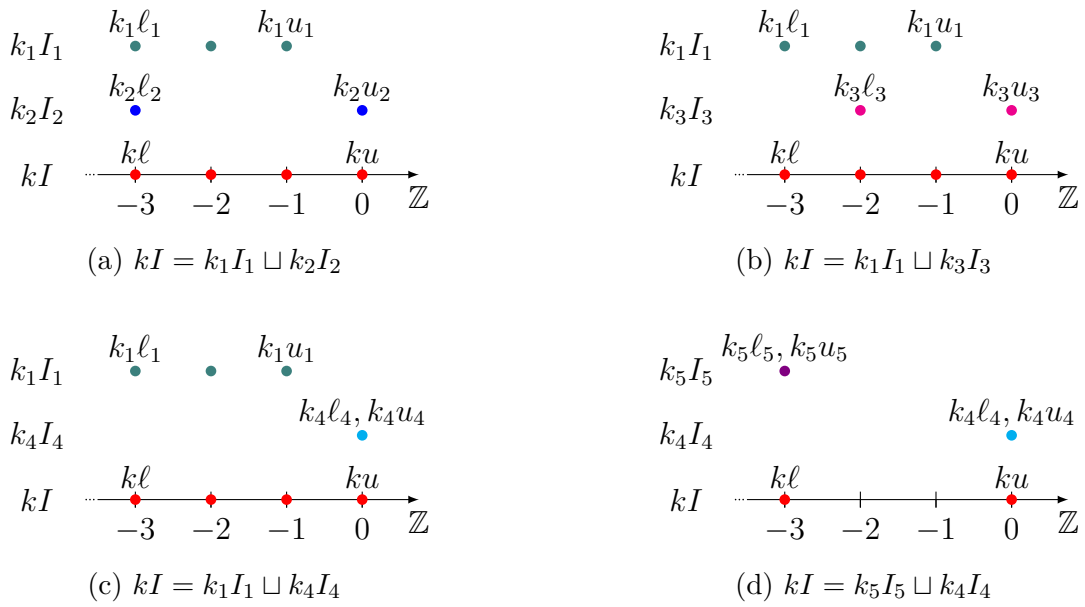
where $k = gcd(k_1, k_2)$, $kI \sqcup \bot = \bot \sqcup kI = kI$ and $kI \sqcup \top = \top \sqcup kI = \top$ for all $kI \in \mathsf{klnt}$.

Observe if $k = 0$, then $k_1 = k_2 = 0$. But if $k_1 = k_2 = 0$, then $I_1 = I_2 = [1,1]$ which is handled by the first case. Therefore in the second case $k > 0$ and thus division by k is safe. The second case also illustrates a general idiom for constructing $\sqcup$ in which a common multiplier k is factored out and then applied to the result of applying $\sqcup$ of a less expressive domain; in this case, intervals. Note that klnt cannot be immediately extended to a lattice by defining $k_1 I_1 \sqcap k_2 I_2 = \sqcup\{kI \in \mathsf{klnt} \mid kI \sqsubseteq k_1 I_1, kI \sqsubseteq k_2 I_2\}$ since $\sqcup$ is only defined on pairs (and therefore finite combinations) of k-intervals.

The following examples, though illustrative in themselves, are used when calculating the least upper bound for ranges. These are then, in turn, used for calculating the least upper bound for marches.

Example 6. Let $k_1 I_1 = 1[-3,-1]$ and $k_2 I_2 = 3[-1,0]$. The two multipliers k_1 and k_2 are weakened to a common $k = gcd(k_1, k_2)$. Here $k = gcd(1,3) = 1$. Then the intervals I_1 and I_2 are scaled to compensate for the smaller multiplier k. Since $k_2 = 3$, the interval $I_2 = [-1,0]$ is scaled to give $3/1 \cdot [-1,0] = [-3,0]$. Since $k_1 = k$, I_1 is not scaled. Hence $k_1 I_1 \sqcup k_2 I_2 = 1\big(1/1 \cdot [-3,-1] \sqcup 3/1 \cdot [-1,0]\big) = 1([-3,-1] \sqcup [-3,0]) = 1[-3,0]$. Figure 10(a) illustrates this join where the concretizations of $k_1 I_1$ and $k_2 I_2$ are shown in green and blue respectively; the concretization of kI is given in red. Note each point in the concretization of kI corresponds to a point in the concretization of $k_1 I_1$ or $k_2 I_2$. Hence no extraneous points are introduced.

Example 7. Let $k_1 I_1 = 1[-3,-1]$ and $k_3 I_3 = 2[-1,0]$ so that the first points of the concretizations, as well as their last, are out of alignment. Since $gcd(1,2) = 1$, then



(a) $kI = k_1I_1 \sqcup k_2I_2$ (b) $kI = k_1I_1 \sqcup k_3I_3$

(c) $kI = k_1I_1 \sqcup k_4I_4$ (d) $kI = k_5I_5 \sqcup k_4I_4$

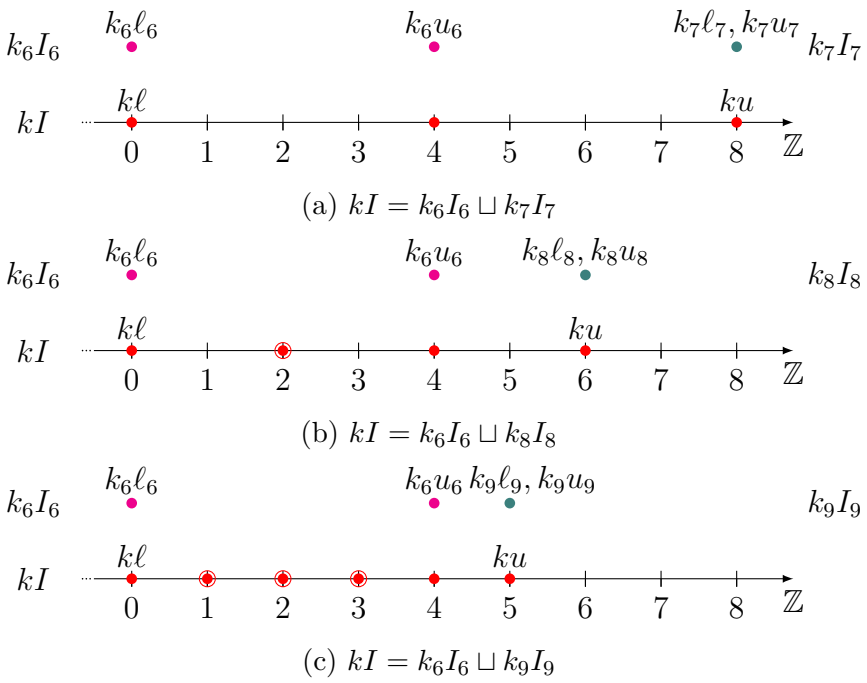
Figure 10: Subfigures showing examples 6 to 9

$kI = k_1I_1 \sqcup k_3I_3 = 1\big(1/1 \cdot [-3,-1] \sqcup 2/1 \cdot [-1,0]\big) = 1([-3,-1] \sqcup [-2,0]) = 1[-3,0]$. Figure 10(b) illustrates that, again, no extraneous points are introduced.

Example 8. Let $k_1I_1 = 1[-3,-1]$ and $k_4I_4 = 0[1,1]$ such that k_4I_4 is degenerate in that it describes a single point. See figure 10(c). Since $gcd(1,0) = 1$, then $k_1I_1 \sqcup k_4I_4 = 1\big(1/1 \cdot [-3,-1] \sqcup 0/1 \cdot [1,1]\big) = 1([-3,-1] \sqcup [0,0]) = 1[-3,0]$. This illustrates that, in general, the magnitude of k can exceed the magnitude of k_4, and likewise k_1.

Example 9. Let $k_5I_5 = -3[1,1]$ and $k_4I_4 = 0[1,1]$ so that both k-intervals are degenerate. See figure 10(d). Since $gcd(-3,0) = 3$, then $k_5I_5 \sqcup k_4I_4 = 3\big(-3/3 \cdot [1,1] \sqcup 0/3 \cdot [1,1]\big) = 3([-1,-1] \sqcup [0,0]) = 3[-1,0]$. More generally, whenever there are two degenerate k-intervals, no information is lost through join.

Example 10. To illustrate where extraneous points are introduced, consider the series of examples illustrated in figure 11. First, let $k_6I_6 = 4[0,1]$ and $k_7I_7 = 8[1,1]$. Since $gcd(4,8) = 4$, then $k_6I_6 \sqcup k_7I_7 = 4[0,1] \sqcup 8[1,1] = 4\big(4/4 \cdot [0,1] \sqcup 8/4 \cdot [1,1]\big) = 4([0,1] \sqcup [2,2]) = 4[0,2]$. Note in this case the point 8 can be reached by extending the interval of k_6I_6 from $[0,1]$ to $[0,2]$, thus no information is lost. However,



(a) $kI = k_6I_6 \sqcup k_7I_7$

(b) $kI = k_6I_6 \sqcup k_8I_8$

(c) $kI = k_6I_6 \sqcup k_9I_9$

Figure 11: Subfigures showing example 10

if the point had been 12, an extra point would have been be introduced at 8. Second, let $k_8I_8 = 6[1,1]$. Since $gcd(4,6) = 2$, then $k_6I_6 \sqcup k_8I_8 = 4[0,1] \sqcup 6[1,1] = 2\big(4/2 \cdot [0,1] \sqcup 6/2 \cdot [1,1]\big) = 2([0,2] \sqcup [3,3]) = 2[0,3]$. Since the common multiplier k is strictly less than both k_1 and k_2, an extra point at 2 is introduced (indicated by a ring). Third, let $k_9I_9 = 5[1,1]$. Since $gcd(4,5) = 1$, then $k_6I_6 \sqcup k_9I_9 = 4[0,1] \sqcup 5[1,1] = 1\big(4/1 \cdot [0,1] \sqcup 5/1 \cdot [1,1]\big) = 1([0,4] \sqcup [5,5]) = 1[0,5]$. The common multiplier of $k = 1$ triggers loss of precision.

A version of scalar multiplication is next introduced for kInt:

Definition 11. Scalar multiplication $\otimes : \mathbb{Z} \times \mathsf{kInt} \to \mathsf{kInt}$ is defined by:

$$\lambda \otimes kI = \begin{cases} (\lambda k)I & \text{if } I = [1,1] \text{ or } \lambda > 0 \\ 0[1,1] & \text{else if } \lambda = 0 \\ (-\lambda k)(-1 \cdot I) & \text{otherwise} \end{cases}$$

where $\lambda \otimes \bot = \bot$ and $\lambda \otimes \top = \top$.

Observe in the first case that scalar multiplication merely scales the multiplier associated with the k-interval. The second and third cases require additional manipulation to ensure that the result of scaling is a valid k-interval.

Unlike the version of scalar multiplication for Int, the number of elements represented by the k-interval before and after scaling always remains constant (except when the multiple is zero). Non-unit scalar multiplication relaxes an interval, but when applied to a k-interval, results in an element which is incomparable to the original, as the following example shows:

Example 11. Let $I \in \mathsf{Int}$ where $I = [-1, 1]$. Then $2 \cdot I = 2 \cdot [-1, 1] = [-2, 2]$. Note $[-1, 1] \sqsubseteq [-2, 2]$. Now $2 \otimes 1I = 2 \otimes 1[-1, 1] = 2[-1, 1]$. But $\gamma(2[-1, 1]) = \{-2, 0, 2\}$ and $\gamma(1[-1, 1]) = \{-1, 0, 1\}$, hence by lemma 7, $2[-1, 1]$ and $1[-1, 1]$ are incomparable.

The following lemma shows that scalar multiplication commutes with concretization, hence scalar multiplication for k-intervals results in no loss of information, as its accompanying example demonstrates:

Lemma 10. Let $\lambda \in \mathbb{Z}$, $kI \in \mathsf{kInt} \setminus \{\top\}$. Then $\gamma(\lambda \otimes kI) = \{\lambda x \mid x \in \gamma(kI)\}$.

Example 12. Consider $\lambda = -3$ and $4[-2, 1] \in \mathsf{kInt}$. Then $\gamma(\lambda \otimes 4[-2, 1]) = \gamma((-(-3) \times 4)(-1 \cdot [-2, 1])) = \gamma(12[-1, 2]) = \{-12, 0, 12, 24\} = \{-3x \mid x \in \{-8, -4, 0, 4\}\} = \{\lambda x \mid x \in \gamma(4[-2, 1])\}$

The next lemma serves as an aid to streamline manipulations in proofs: it avoids performing a case analysis on the sign of the multiplier. The accompanying lemma also abstracts over the three cases of $\otimes$ and asserts that the ordering relation for scaled k-intervals reflects that used to order unscaled k-intervals.

Lemma 11. Let $kI \in \mathsf{kInt} \setminus \{\bot, \top\}$. Then $kI = k \otimes 1I$.

Lemma 12. Let $\lambda, \mu \in \mathbb{Z}$ and $k_1I_1, k_2I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$. Then $\lambda \otimes k_1I_1 \sqsubseteq \mu \otimes k_2I_2$ iff $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$.

The following result establishes for a given k-interval, the scalar multiplier k, the interval I and its offset λ can all be negated without effect, provided they are negated simultaneously.

Lemma 13. Let $k(\lambda + I) \in \mathsf{kInt}$ where $\lambda \in \mathbb{Z}$ and $I \neq [1, 1]$. Then $k(\lambda + I) \equiv -k(-\lambda + (-1 \cdot I))$.

All the results which follow are categorized into one of three types according to whether the result is broadly a statement relating to monotonicity, associativity or distributivity.

3.3.2 Monotonicity

The following two corollaries state monotonicity properties. The first relates scalar multiplication on intervals, $\cdot$, to that on k-intervals, $\otimes$, in the special case where the k-multiplier is one. The second shows that $\otimes$ is monotonic, even when the multiplier is strictly negative.

Corollary 5. Let $\lambda \in \mathbb{Z}$ and $1I_1, 1I_2 \in \mathsf{kInt}$. Then $\lambda \otimes 1I_1 \sqsubseteq 1I_2$ iff $\lambda \cdot I_1 \sqsubseteq 1 \cdot I_2$.

Corollary 6. Let $k_1I_1 \sqsubseteq k_2I_2$ where $k_1I_1, k_2I_2 \in \mathsf{kInt}$.

- Then $\lambda \otimes k_1I_1 \sqsubseteq \lambda \otimes k_2I_2$ where $\lambda \in \mathbb{Z}$.
- Then $1 \otimes k_1I_1 \sqsubseteq 1 \otimes k_2I_2$.
- Then $-1 \otimes k_1I_1 \sqsubseteq -1 \otimes k_2I_2$.

The following proposition shows that scaling two k-intervals, related by $\sqsubseteq$, by a positive integer, either by integer multiplication or division, preserves the syntactic structure of the k-interval: there is no need to deploy the $\otimes$ operator to adjust both the scalar and the interval. The companion corollary relaxes the restriction that the scalar d is strictly positive, albeit at the cost of using $\otimes$.

Proposition 6. Let $k_1I_1 \sqsubseteq k_2I_2$ where $k_1I_1, k_2I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$. Furthermore, let $d \in \mathbb{Z}$ such that $d > 0$.

- Then $(dk_1)I_1 \sqsubseteq (dk_2)I_2$
- If $d|k_1$ and $d|k_2$, then $(k_1/d)I_1 \sqsubseteq (k_2/d)I_2$

Corollary 7. Let $k_1I_1 \sqsubseteq k_2I_2$ where $k_1I_1, k_2I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$.

- Let $d \in \mathbb{Z}$. Then $(dk_1) \otimes 1I_1 \sqsubseteq (dk_2) \otimes 1I_2$.
- Let $d \in \mathbb{Z} \setminus \{0\}$ such that $d|k_1$ and $d|k_2$. Then $(k_1/d) \otimes 1I_1 \sqsubseteq (k_2/d) \otimes 1I_2$.

3.3.3 Associativity

Both the following results are statements on associativity. The first, which shows that integer multiplication commutes with $\otimes$, mirrors the analogous lemma 5 for intervals. Note how the second equality follows from the commutativity of integer multiplication. The second introduces a cancellation property which follows from the associativity of integer multiplication and scalar multiplication of vectors.

Lemma 14. Let $\lambda, \mu \in \mathbb{Z}$ and $kI \in \mathsf{kInt}$. Then $\lambda \otimes (\mu \otimes kI) = (\lambda\mu) \otimes kI = \mu \otimes (\lambda \otimes kI)$.

Corollary 8. Let $\vec{x}/\vec{y} \in \mathbb{Z}$, $\vec{w}/\vec{x} \in \mathbb{Z}$ and $kI \in \mathsf{kInt}$. Then $\vec{x}/\vec{y} \otimes (\vec{w}/\vec{x} \otimes kI) = \vec{w}/\vec{y} \otimes kI$ where $\vec{w}/\vec{y} \in \mathbb{Z}$.

3.3.4 Distributivity

The following lemma of shows a distributivity property of scalar multiplication over the join of two k-intervals. Again, the corollary which follows generalizes this result so that the scalar multiplier to the k-intervals is not the same on both sides.

Lemma 15. Let $\lambda \in \mathbb{Z}$, $k_1I_1, k_2I_2 \in \mathsf{kInt}$. Then $\lambda \otimes (k_1I_1 \sqcup k_2I_2) = \lambda \otimes k_1I_1 \sqcup \lambda \otimes k_2I_2$.

Example 13. To illustrate that the lemma holds even when λ is strictly negative, consider $\lambda = -2$, $k_1I_1 = 5[1,2]$ and $k_2I_2 = 0[1,1]$. First, observe $k_1I_1 \sqcup k_2I_2 = 5[1,2] \sqcup 0[1,1] = 5\big(5/5 \cdot [1,2] \sqcup 0/5 \cdot [1,1]\big) = 5\big(1 \cdot [1,2] \sqcup 0 \cdot [1,1]\big) = 5\big([1,2] \sqcup [0,0]\big) = 5[0,2]$. Then $\lambda \otimes (k_1I_1 \sqcup k_2I_2) = -2 \otimes (5[1,2] \sqcup 0[1,1]) = -2 \otimes 5[0,2] = 10[-2,0] = 10[-2,-1] \sqcup 0[1,1] = -2 \otimes 5[1,2] \sqcup -2 \otimes 0[1,1] = \lambda \otimes k_1I_1 \sqcup \lambda \otimes k_2I_2$.

Corollary 9. Let $\vec{x}/\vec{y}$, $\vec{v}/\vec{x}$ and $\vec{w}/\vec{x} \in \mathbb{Z}$ and $k_1I_1, k_2I_2 \in \mathsf{kInt}$. Then it holds $\vec{x}/\vec{y} \otimes \big(\vec{v}/\vec{x} \otimes k_1I_1 \sqcup \vec{w}/\vec{x} \otimes k_2I_2\big) = \vec{v}/\vec{y} \otimes k_1I_1 \sqcup \vec{w}/\vec{y} \otimes k_2I_2$ where $\vec{v}/\vec{y}, \vec{w}/\vec{y} \in \mathbb{Z}$.

Example 14. To show how distributivity is applicable even when the scalar multipliers on the k-intervals differ, consider $\vec{v} = (16,8)$, $\vec{w} = (12,6)$, $\vec{x} = (4,2)$, $\vec{y} = (2,1)$. Observe $\vec{v}/\vec{x} = 4$ and $\vec{w}/\vec{x} = 3$ hence the scalar multipliers on k_1I_1 and k_2I_2 differ. For concreteness, let $k_1I_1 = -3[1,1]$ and $k_2I_2 = 10[3,4]$. Then $\vec{x}/\vec{y} = 2$, $\vec{v}/\vec{y} = 8$ and $\vec{w}/\vec{y} = 6$. Then $\vec{x}/\vec{y} \otimes \big(\vec{v}/\vec{x} \otimes k_1I_1 \sqcup \vec{w}/\vec{x} \otimes k_2I_2\big) = 2 \otimes (4 \otimes -3[1,1] \sqcup 3 \otimes 10[3,4]) = 2 \otimes (-12[1,1] \sqcup 30[3,4]) = 2 \otimes 6[-2,20] = 12[-2,20] = -24[1,1] \sqcup 60[3,4] = 8 \otimes -3[1,1] \sqcup 6 \otimes 10[3,4] = \vec{v}/\vec{y} \otimes k_1I_1 \sqcup \vec{w}/\vec{y} \otimes k_2I_2$.

3.4 The range abstract domain: Range^n

Up until now, the abstract domains presented can be used to represent sets generated by arithmetic sequences. Sets generated by sequences with a common difference of 0 or 1 can be represented by intervals. More generally, sets generated by arithmetic sequences where the difference is an arbitrary integer k and whose initial term is a multiple of k require k-intervals. Regardless, the sequences enumerate integers on the one-dimensional number line. To lift this development to sequences of vectors, a k-interval can be augmented with a n-dimensional vector which enables the representation of sets of points in n-dimensional space:

Definition 12. Given $n \in \mathbb{N}$, the Range^n domain is defined:

$$\mathsf{Range}^n = \{\bot, \top\} \cup (\mathsf{kInt} \setminus \{\bot, \top\}) \times (\mathbb{Z}^n \setminus \{\vec{0}\}))$$

Observe Range^n is parametric on the dimension n. Observe also that the zero vector is excluded from the set of permissible vector multipliers. As a consequence, the only way to represent the singleton set that contains the origin is to have a scalar multiplier on the k of 0. Notice too that $\top$ and $\bot$ are only permitted outside the context of a pair. This simplifies the structure of the domain without limiting its expressiveness.

For insight into the roles of the different components of a range, consider a range $kI\vec{v}$ and suppose that $\vec{v}$ is irreducible in the sense that ± 1 are the only common divisors of the elements of $\vec{v}$. The two components of $kI\vec{v}$ contribute to three different aspects of the abstract element: the cardinality of the interval I corresponds to the cardinality of the set of points $kI\vec{v}$ represents; k scales the elements of I; and $\vec{v}$ determines the direction of the line on which the points are found. In the more general case that $\vec{v}$ is reducible, it is the combination of k and the common factor of the components of $\vec{v}$ that provides the scaling. The intuition behind the range domain, put simply, is that it permits the number line to be reorientated to any desired direction.

Definition 13. The binary relation $\sqsubseteq$ on Range^n is the least relation such that:

- $\bot \sqsubseteq r \sqsubseteq \top$ for all $r \in \mathsf{Range}^n$

- $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$ if $I_1 = I_2 = [1,1]$ and $k_1\vec{x} = k_2\vec{y}$
- $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$ if $I_2 \neq [1,1]$, $k_2\vec{y} | k_1\vec{x}$ and $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$

Observe in the second case, since $I_2 \neq [1,1]$, then $k_2\vec{y} \neq \vec{0}$, so division by $k_2\vec{y}$ is defined. In particular, if $k_1\vec{x} = \vec{0}$, the ordering relation reduces to $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 = 0[1,1] \sqsubseteq 1I_2$. Note the structural resemblance to corollary 7. In the setting of ranges, the ordering relation requires the divisibility of integral vectors, rather than integers. Like as for k-intervals, the ordering is validated by the following lemma which asserts that the ordering $\sqsubseteq$ mirrors the subset ordering $\subseteq$ on the image sets of the ranges under γ:

Lemma 16. Let $r_1, r_2 \in \mathsf{Range}^n$. Then $r_1 \sqsubseteq r_2$ iff $\gamma(r_1) \subseteq \gamma(r_2)$ where $\gamma(kI\vec{v}) = \{x\vec{v} \mid x \in \gamma(kI)\}$, $\gamma(\bot) = \emptyset$ and $\gamma(\top) = \mathbb{Z}^n$.

To amplify the discussion given above, note the cardinality property holds by virtue of the structure of $kI\vec{v}$. First observe since $\vec{v} \neq \vec{0}$, then $|\gamma(kI\vec{v})| = |\gamma(kI)|$. Second, note that if $k = 0$, then $kI = 0[1,1]$, hence $|\gamma(I)| = 1 = |\gamma(kI\vec{v})|$. But if $k \neq 0$, then $|\gamma(kI)| = |\gamma(I)|$, hence $|\gamma(kI\vec{v})| = |\gamma(I)|$.

Example 15. To illustrate the cardinality property described above, consider $kI\vec{v} \in \mathsf{Range}^n$ where $kI = 2[1,3]$, $n = 2$ and $\vec{v} = (1,5)$. Observe $\vec{v}$ is irreducible since if $\vec{v} = \lambda\vec{w}$ where $\vec{w} \in \mathbb{Z}^n$ then $\lambda = \pm 1$. Then $|\gamma(I)| = |\{1,2,3\}| = 3$. Observe $|\gamma(kI\vec{v})| = |\{2\vec{v}, 4\vec{v}, 6\vec{v}\}| = 3$.

Lemma 17. Let $kI\vec{v} \in \mathsf{Range}^n$ where $|\gamma(kI\vec{v})| = 1$. Then $I = [1,1]$.

Since the range domain is a product of a k-interval and an n-dimensional vector, a close correspondence between the ordering of range elements and their respective k-interval elements follows:

Lemma 18. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$ and $\vec{x} \in \mathbb{Z}^n \setminus \{\vec{0}\}$. Then $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{x}$ iff $k_1 I_1 \sqsubseteq k_2 I_2$.

The following lemma provides meaning to a range element whose interval is translated by an integer offset. It states that the points represented by offsetting the interval in a range can also be found by offsetting the range directly, albeit with a vector offset. The accompanying corollary lifts this concept to quotiented vectors.

Lemma 19. Let $k(\lambda+I)\vec{v} \in \mathsf{Range}^n$ where $\lambda \in \mathbb{Z}$. Then $\gamma(k(\lambda+I)\vec{v}) = \{\lambda k\vec{v}+\vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$.

Corollary 10. Let $k(\vec{t}/(k\vec{v})+I)\vec{v} \in \mathsf{Range}^n$ where $\vec{t} \in \mathbb{Z}^n$, $k\vec{v}|\vec{t}$ and $k\vec{v} \neq \vec{0}$. Then $\gamma(k(\vec{t}/(k\vec{v})+I)\vec{v}) = \{\vec{t}+\vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$.

Example 16. To illustrate the lemma, suppose $k(\lambda+I)\vec{v} \in \mathsf{Range}^n$ where $k=2$, $\lambda=5$, $I=[1,2]$ and $\vec{v} \in \mathbb{Z}^n \setminus \{\vec{0}\}$. Observe $k(\lambda+I)\vec{v} = 2(5+[1,2])\vec{v} = 2[6,7]\vec{v} \in \mathsf{Range}^n$. Then $\gamma(k(\lambda+I)\vec{v}) = \gamma(2[6,7]\vec{v}) = \{12\vec{v}, 14\vec{v}\} = \{10\vec{v}+2\vec{v}, 10\vec{v}+4\vec{v}\} = \{10\vec{v}+\vec{x} \mid \vec{x} \in \gamma(2[1,2]\vec{v})\} = \{\lambda k\vec{v}+\vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$.

Unlike k-intervals, the syntactic structure of ranges is free enough to permit different range elements $r_1, r_2, \ldots, r_n$ to be considered equivalent, that is $r_1 \equiv r_2 \equiv \ldots \equiv r_n$, where the equivalence relation is defined $k_1I_1\vec{x} \equiv k_2I_2\vec{y}$ iff $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{y}$ and $k_2I_2\vec{y} \sqsubseteq k_1I_1\vec{x}$. As discussed earlier, a range with an irreducible vector demonstrates most clearly the three different aspects of a range. However, it is clearly possible to either move a divisor of k into the vector (provided $k \neq 0$) or to move common divisors of the elements of the vector back into k. Since concretization of a range $kI\vec{v}$ produces vectors that are the product of the elements of I, k and $\vec{v}$, then it is clear the resultant vectors will be equal, and thus the ranges equivalent, regardless of how the divisors are distributed. For example, for range elements with a reducible vector $\vec{v}$, the vector can equally be expressed as $(\vec{v}/\vec{z})\vec{z}$ where $\vec{v}/\vec{z} \neq \pm 1$. As shown earlier in lemma 11, a k-value can be brought out to premultiply the k-interval $1I$. The common factor $\vec{v}/\vec{z}$ can then be brought out from the vector and combined with the k to give the equivalent range $((k\vec{v})/\vec{z} \otimes 1I)\vec{z}$, as asserted in the following lemma:

Lemma 20. Let $kI\vec{v} \in \mathsf{Range}^n$ and $\vec{z} \in \mathbb{Z}^n$ where $\vec{z}|k\vec{v}$ and $\vec{z} \neq \vec{0}$. Then $kI\vec{v} \equiv ((k\vec{v})/\vec{z} \otimes 1I)\vec{z}$.

Example 17. To illustrate the lemma, consider $kI\vec{v} \in \mathsf{Range}^n$ where $kI = 2[1,2]$ and $\vec{v} = (-6,3)$, so $k\vec{v} = (-12,6)$. Also suppose $\vec{z} = (4,-2)$. Observe $\vec{z}|k\vec{v}$ since

$-3\vec{z} = k\vec{v}$, but note $\vec{z} \nmid \vec{v}$. Then

$$\begin{aligned}
\gamma(kI\vec{v}) &= \gamma(2[1,2](-6,3)) \\
&= \{(-12,6),(-24,12)\} \\
&= \gamma(3[-2,-1](4,-2)) \\
&= \gamma((-3 \otimes 1[1,2])(4,-2)) \\
&= \gamma(((-12,6)/(4,-2) \otimes 1[1,2])(4,-2)) \\
&= \gamma(((k\vec{v})/\vec{z} \otimes 1I)\vec{z})
\end{aligned}$$

Thus $kI\vec{v} \equiv ((k\vec{v})/\vec{z} \otimes 1I)\vec{z}$, which concurs with the lemma.

The following corollary details convenient results that move factors between k and the vector:

Corollary 11.

- Let $kI(\lambda\vec{v}) \in \mathsf{Range}^n$. Then $kI(\lambda\vec{v}) \equiv (\lambda \otimes kI)\vec{v}$.
- Let $kI\vec{v} \in \mathsf{Range}^n$ where $k \neq 0$. Then $kI\vec{v} \equiv 1I(k\vec{v})$.

As a general strategy in domain construction, an abstract domain D that is a pre-order can always be lifted to a domain which is a partially ordered set by using equivalence classes to gain anti-symmetry. To explain the construction, suppose D is equipped with a concretization map $\gamma : D \to \wp(C)$ where C is a concrete domain and $\langle D, \sqsubseteq \rangle$ is a pre-ordered set. An equivalence relation $d_1 \equiv d_2$ iff $\gamma(d_1) = \gamma(d_2)$ can be used to quotient D as follows $D/\equiv = \{[d]_\equiv \mid d \in D\}$. Then the quotient domain $D/\equiv$ is partially ordered by $[d_1]_\equiv \sqsubseteq [d_2]_\equiv$ iff $d_1 \sqsubseteq d_2$. Observe $\sqsubseteq$ is anti-symmetric on $D/\equiv$ by construction. The problem with this strategy is that it introduces extra layers of complexity, namely, equivalence classes and quotient domains. Therefore we seek to induce a canonical representation whenever this is straightforward which is the case for k-intervals. This is less so for ranges and therefore instead we choose to work with a more general form of join, the rationale always being to choose the simplest mathematical formulation for the purposes at hand.

A similar argument holds for $k_1 > 0$ and $k_2 \leq 0$. The rationale for adopting a canonical representation is that it simplifies join in that it removes the need to operate at the level of equivalence classes.

3.4.1 Interpolation results

Given two ranges r_0 and r_2, an interpolation result in this setting characterizes a range r_1 that falls between the two ranges in the sense that $r_0 \sqsubseteq r_1 \sqsubseteq r_2$. Such an interpolant can be derived by scalar multiplication on intervals:

Lemma 21. Let $kI(\lambda\vec{v}) \in \mathsf{Range}^n$ where $\lambda \in \mathbb{Z}$. Then $kI(\lambda\vec{v}) \sqsubseteq k(\lambda \cdot I)\vec{v} \sqsubseteq 1(k\lambda \cdot I)\vec{v}$

An alternative interpolation result is $kI(\lambda\vec{v}) \sqsubseteq 1(k \cdot I)(\lambda\vec{v}) \sqsubseteq 1(k\lambda \cdot I)\vec{v}$, where the interval is first weakened by scalar multiplication by k, and then by λ, rather than the other way around. Although this is interesting in itself, it is not needed in the overall development of the domain hence it is not formally proven. The following corollary makes two convenient simplifications: the first relates to the left $\sqsubseteq$ for $k = 1$ and the second the right $\sqsubseteq$ for $\lambda = 1$.

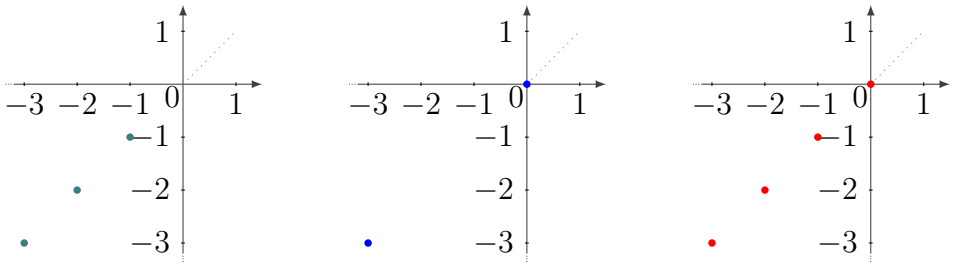
Corollary 12. Let $kI(\lambda\vec{v}) \in \mathsf{Range}^n$ where $\lambda \in \mathbb{Z}$. Then

- $1I(\lambda\vec{v}) \sqsubseteq 1(\lambda \cdot I)\vec{v}$
- $kI\vec{v} \sqsubseteq 1(k \cdot I)\vec{v}$

The following result states that given an interpolation property on divisibility, namely $k_2\vec{y}|\vec{z}|k_1\vec{x}$, an interpolation result for ranges follows:

Lemma 22. Let $k_1I_1\vec{x}$, $k_2I_2\vec{y} \in \mathsf{Range}^n$ and $\vec{z} \in \mathbb{Z}^n$ where $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{y}$, $I_1 \neq [1, 1]$, $k_2\vec{y}|\vec{z}$ and $\vec{z}|k_1\vec{x}$. Then $k_1I_1\vec{x} \sqsubseteq 1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z} \sqsubseteq k_2I_2\vec{y}$.

Observe that the $\sqsubseteq$ ordering is merely a quasi-ordering, rather than a partial ordering. To see this, consider $k_1I_1\vec{x} = 2[0, 1](3, 4)$ and $k_2I_2\vec{y} = 2[-1, 0](-3, -4)$. Observe $k_1I_1\vec{x} \neq k_2I_2\vec{y}$ but $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{y}$ and $k_2I_2\vec{y} \sqsubseteq k_1I_1\vec{x}$. This precludes the existence of a join, since the result of a join is unique. Pseudo-join relaxes the notion of join to return a range arbitrarily chosen from several candidates:



(a) r_1 (b) r_2 (c) $r_1 \sqcup r_2$

Figure 12: Graphs showing r_1, r_2 and their join $r_1 \sqcup r_2$ from example 18. Observe the result is analogous to $k_1 I_1 \sqcup k_2 I_2$ from example 6 but with the number line orientated along the line represented by the vector (1,1).

3.4.2 Pseudo-join

Proposition 7. $\langle \mathsf{Range}^n, \sqsubseteq, \sqcup \rangle$ is a pseudo-join-semilattice where:

$$k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y} = \begin{cases} k_1 I_1 \vec{x} & \text{if } I_1 = I_2 = [1,1] \text{ and } k_1\vec{x} = k_2\vec{y} \\ \big((k_1\vec{x})/\vec{z} \otimes 1 I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1 I_2\big)\vec{z} & \text{else if } \vec{z} = gcd(k_1\vec{x}, k_2\vec{y}) \neq \bot \\ \top & \text{otherwise} \end{cases}$$

and $r \sqcup \bot = \bot \sqcup r = r$ and $r \sqcup \top = \top \sqcup r = \top$ for all $r \in \mathsf{Range}^n$.

Observe if $\vec{z} = \vec{0}$, then $k_1\vec{x} = k_2\vec{y} = \vec{0}$. But since $\vec{x}, \vec{y} \in \mathbb{Z}^n \setminus \{\vec{0}\}$, then $k_1 = k_2 = 0$ and thus $I_1 = I_2 = [1,1]$. This is handled by the first case. Therefore in the second case, $\vec{z} \neq \vec{0}$ and thus division by $\vec{z}$ is defined.

The following examples serve to illustrate how the join of k-intervals can be lifted to joins for ranges. Each of the following four examples mirrors the four examples numbered 6 to 9.

Example 18. Consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_2 = 1[-1,0](3,3)$. Since $gcd(1(1,1), 1(3,3)) = (1,1)$ and, using the result from example 6 in the final

step, then:

$$\begin{aligned} r_1 \sqcup r_2 &= 1[-3,-1](1,1) \sqcup 1[-1,0](3,3) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (3,3)/(1,1) \otimes 1[-1,0]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 3 \otimes 1[-1,0]\big)(1,1) \\ &= (1[-3,-1] \sqcup 3[-1,0])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

The two k-intervals are scaled up so that they share a common vector (1,1) and then are joined. The scaling of the k-multiples reflects the factor by which the original vectors (1,1) and (3,3) are scaled down, with the $\otimes$ operator ensuring the resultant k-intervals are correctly defined. The result is shown in figure 12.

Example 19. Now consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_3 = 1[-1,0](2,2)$. Since $gcd(1(1,1), 1(2,2)) = (1,1)$ and, using the result from example 7 in the final step, then:

$$\begin{aligned} r_1 \sqcup r_3 &= 1[-3,-1](1,1) \sqcup 1[-1,0](2,2) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (2,2)/(1,1) \otimes 1[-1,0]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 2 \otimes 1[-1,0]\big)(1,1) \\ &= (1[-3,-1] \sqcup 2[-1,0])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

Again the result is the same as that shown in figure 12 and the result is analogous to example 7 with the number line orientated along $(1,1)$.

Example 20. To continue the development, consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_4 = 0[1,1](1,1)$. Since $gcd(1(1,1), 0(1,1)) = (1,1)$ and, using the result from

example 8 in the final step, then:

$$\begin{aligned} r_1 \sqcup r_4 &= 1[-3,-1](1,1) \sqcup 0[1,1](1,1) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (0,0)/(1,1) \otimes 1[1,1]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 0 \otimes 1[1,1]\big)(1,1) \\ &= (1[-3,-1] \sqcup 0[1,1])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

Example 21. Now consider the ranges $r_5 = -3[1,1](1,1)$ and $r_4 = 0[1,1](1,1)$. Since $gcd(-3(1,1), 0(1,1)) = (3,3)$, then:

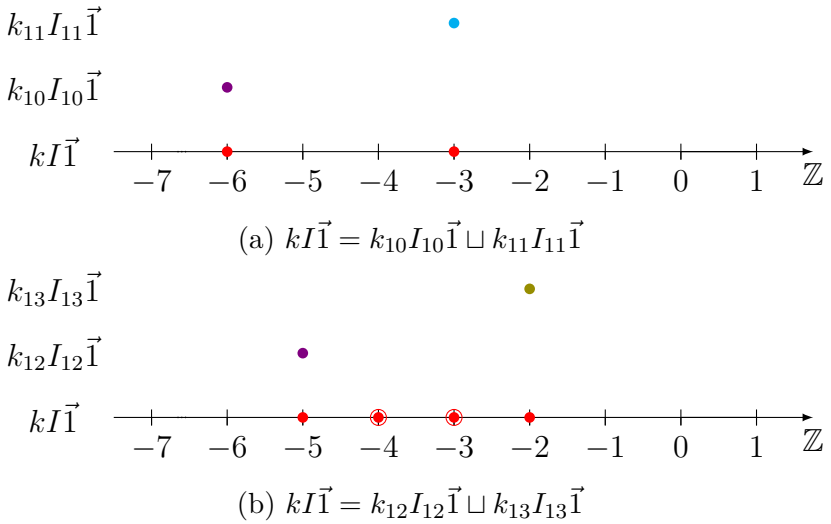
$$\begin{aligned} r_5 \sqcup r_4 &= -3[1,1](1,1) \sqcup 0[1,1](1,1) \\ &= \big((-3,-3)/(3,3) \otimes 1[1,1] \sqcup (0,0)/(3,3) \otimes 1[1,1]\big)(3,3) \\ &= \big(-1 \otimes 1[1,1] \sqcup 0 \otimes 1[1,1]\big)(3,3) \\ &= (-1[1,1] \sqcup 0[1,1])(3,3) \\ &= 1[-1,0](3,3) \end{aligned}$$

Example 22. Finally consider the ranges $r_6 = 1[1,3](0,1)$ and $r_7 = 1[0,1](0,2)$. Since $gcd(1(0,1), 1(0,2)) = (0,1)$, then:

$$\begin{aligned} r_6 \sqcup r_7 &= 1[1,3](0,1) \sqcup 1[0,1](0,2) \\ &= \big((0,1)/(0,1) \otimes 1[1,3] \sqcup (0,2)/(0,1) \otimes 1[0,1]\big)(0,1) \\ &= \big(1 \otimes 1[1,3] \sqcup 2 \otimes 1[0,1]\big)(0,1) \\ &= (1[1,3] \sqcup 2[0,1])(0,1) \\ &= 1[0,3](0,1) \end{aligned}$$

3.5 The march abstract domain: March^n

To motivate marches, observe that a weakness of both the k-interval domain and the range domain (its analogue in n-dimensions) is that the abstractions are anchored to the origin.



(a) $kI\vec{1} = k_{10}I_{10}\vec{1} \sqcup k_{11}I_{11}\vec{1}$

(b) $kI\vec{1} = k_{12}I_{12}\vec{1} \sqcup k_{13}I_{13}\vec{1}$

Figure 13: Subfigures showing example 23.

Example 23. Figure 13 uses the ranges $k_{10}I_{10}\vec{1}$, $k_{11}I_{11}\vec{1}$, $k_{12}I_{12}\vec{1}$ and $k_{13}I_{13}\vec{1}$ to illustrate this. The ranges in part (a), namely $k_{10}I_{10}\vec{1}$ and $k_{11}I_{11}\vec{1}$, represent two points, one at -6 and the other at -3. The two ranges in part (b), $k_{12}I_{12}\vec{1}$ and $k_{13}I_{13}\vec{1}$, likewise represent two points, -5 and -2, equally separated by a difference of 3. Observe that the points in part (b) are the merely those of part (a) translated by one unit. However, the joins differ dramatically. The join of $k_{10}I_{10}\vec{1}$ and $k_{11}I_{11}\vec{1}$ introduces no new points, whereas the join of $k_{11}I_{11}\vec{1}$ and $k_{13}I_{13}\vec{1}$ results in a range with double the number of points. This is because 3 divides both -3 and -6 but 1 is the largest divisor of -5 and -2. Put another way, by adding multiples of 3 to both -6 and -3 it is possible to reach the origin. Likewise, it is possible to reach the origin by adding multiples of 1 to both -5 and -2. It is this requirement to anchor a range at the origin that induces a loss of information in part (b).

To overcome this limitation, the march domain is introduced:

Definition 14. Given $n \in \mathbb{N}$, the March^n domain is defined:

$$\mathsf{March}^n = \{\bot, \top\} \cup (\mathbb{Z}^n \times (\mathsf{Range}^n \setminus \{\top, \bot\}))$$

If $m = (s, r) \in \mathsf{March}^n$, we write $m = s + r$.

The march domain combines a position (offset) vector with a range. While a

range is anchored to the origin, a march merely requires the distance between any two represented points to be a multiple of a given vector.

Example 24. Continuing with example 23, observe how the points -5 and -2 can be represented by $\vec{-8}+3[1,2]\vec{1}$. The role of the $\vec{-8}$ is to define a translation of the points that gives a new set of points which are representable as a range whilst also relaxing the anchoring requirement.

The first step in formalizing the march domain is introducing its ordering relation:

Definition 15. The binary relation $\sqsubseteq$ on March^n is the least relation such that:

- $\bot \sqsubseteq m \sqsubseteq \top$ for all $m \in \mathsf{March}^n$
- $\vec{s_1} + k_1 I_1 \vec{x} \sqsubseteq \vec{s_2} + k_2 I_2 \vec{y}$ if $I_1 = I_2 = [1,1]$ and $\vec{s_1} + k_1\vec{x} = \vec{s_2} + k_2\vec{y}$
- $\vec{s_1} + k_1 I_1 \vec{x} \sqsubseteq \vec{s_2} + k_2 I_2 \vec{y}$ if $I_1 = [1,1]$, $I_2 \neq [1,1]$, $k_2\vec{y} | (\vec{s_2} - (\vec{s_1} + k_1\vec{x}))$ and $[0,0] \sqsubseteq (\vec{s_2} - (\vec{s_1} + k_1\vec{x}))/(k_2\vec{y}) + I_2$
- $\vec{s_1} + k_1 I_1 \vec{x} \sqsubseteq \vec{s_2} + k_2 I_2 \vec{y}$ if $I_1 \neq [1,1]$, $I_2 \neq [1,1]$, $k_2\vec{y} | (\vec{s_2} - \vec{s_1})$ and $k_1 I_1 \vec{x} \sqsubseteq k_2((\vec{s_2} - \vec{s_1})/(k_2\vec{y}) + I_2)\vec{y}$

For a degenerate march $\vec{s_1} + k_1 I_1 \vec{x}$, that is where $I_1 = [1,1]$, the point represented by the march is simply the sum of the offset vector, $\vec{s_1}$, and the vector generated by the range, $k_1\vec{x}$. Thus for two degenerate marches, if $\vec{s_1} + k_1\vec{x} = \vec{s_2} + k_2\vec{y}$, then each precedes the other. For the third and fourth cases above, for $\vec{s_1} + k_1 I_1 \vec{x} \sqsubseteq \vec{s_2} + k_2 I_2 \vec{y}$ to hold, it is necessary for the points of $\vec{s_1} + k_1 I_1 \vec{x}$ to be located on the same line as those of $\vec{s_2} + k_2 I_2 \vec{y}$. Specifically for the third case, where $\vec{s_1} + k_1 I_1 \vec{x}$ is degenerate, the translation $-(\vec{s_1} + k_1\vec{x})$ on the march $\vec{s_2} + k_2 I_2 \vec{y}$ reduces the ordering relation to solely checking whether the translated march contains the origin. (Other translations could be used in the ordering. In fact, if $y \in \mathbb{Z}$, then $-(\vec{s_1} + k_1\vec{x} + yk_2\vec{y})$ will suffice, but changing the test to $[-y,-y] \sqsubseteq (\vec{s_2} - (\vec{s_1} + k_1\vec{x} + yk_2\vec{y}))/(k_2\vec{y}) + I_2$.) For the fourth case where m_1 represents two or more points, the direction is important. After the translation has been made, the problem is reduced to a range ordering decision. Like as for k-intervals and ranges, the ordering is again validated by the following lemma which

asserts that the ordering $\sqsubseteq$ mirrors the subset ordering $\subseteq$ on the image sets of the ranges under γ. The accompanying corollary asserts that applying the same offset to the position vectors of two marches will not compromise their ordering.

Lemma 23. Let $m_1, m_2 \in \mathsf{March}^n$. Then $m_1 \sqsubseteq m_2$ iff $\gamma(m_1) \subseteq \gamma(m_2)$ where $\gamma(\vec{s} + kI\vec{v}) = \{\vec{s} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$, $\gamma(\bot) = \emptyset$ and $\gamma(\top) = \mathbb{Z}^n$.

Corollary 13. Let $m_1, m_2 \in \mathsf{March}^n$ where $m_1 = \vec{s}_1 + k_1 I_1 \vec{x}$ and $m_2 = \vec{s}_2 + k_2 I_2 \vec{y}$. Then $\vec{s}_1 + k_1 I_1 \vec{x} \sqsubseteq \vec{s}_2 + k_2 I_2 \vec{y}$ iff $(\vec{s}_1 + \vec{s}) + k_1 I_1 \vec{x} \sqsubseteq (\vec{s}_2 + \vec{s}) + k_2 I_2 \vec{y}$

The following lemma states that if a march has a single point in its concretization, then the interval must take a unique form:

Lemma 24. Let $\vec{s} + kI\vec{v} \in \mathsf{March}^n$. Then $|\gamma(\vec{s} + kI\vec{v})| = 1$ implies $I = [1, 1]$ and $|\gamma(kI\vec{v})| = 1$.

Similar to ranges, the march domain has enough degrees of freedom for two marches m_1, m_2, to represent the same points, while being syntactically different. Two (possibly syntactically different) marches are considered equivalent, denoted $m_1 \equiv m_2$, if $m_1 \sqsubseteq m_2$ and $m_2 \sqsubseteq m_1$. The following lemma demonstrates an immediate application of the concept. It states that it is possible to offset the position vector of a march by a multiple of $k\vec{v}$ as long as the same multiple (but opposite sign) is added to its interval. The equal but opposite translations cancel each other. The ability to change the position vector of marches in this way (henceforth called rebasing) proves useful in the following lemmas and corresponding proofs. The accompanying corollary lists a number of permutations of the same concept.

Lemma 25. Let $kI\vec{v} \in \mathsf{Range}^n$ where $I \neq [1, 1]$, $\lambda \in \mathbb{Z}$ and $\vec{s} \in \mathbb{Z}^n$. Then $\vec{s} + kI\vec{v} \equiv (\vec{s} - \lambda k\vec{v}) + k(\lambda + I)\vec{v}$.

Corollary 14.

Let $kI\vec{v} \in \mathsf{Range}^n$ where $I \neq [1, 1]$ and let $\vec{u}, \vec{s}, \vec{t} \in \mathbb{Z}^n$ where $k\vec{v} | \vec{u}$.

1. Then $\vec{s} + kI\vec{v} \equiv (\vec{s} - \vec{u}) + k(\vec{u}/(k\vec{v}) + I)\vec{v}$
2. Let $\vec{u} = k\vec{v}$. Then $\vec{s} + kI\vec{v} \equiv (\vec{s} - k\vec{v}) + k(1 + I)\vec{v}$
3. Let $\vec{u} = -k\vec{v}$. Then $\vec{s} + kI\vec{v} \equiv (\vec{s} + k\vec{v}) + k(-1 + I)\vec{v}$

4. Let $\vec{u} = \vec{s} - \vec{t}$. Then $\vec{s} + kI\vec{v} \equiv \vec{t} + k((\vec{s} - \vec{t})/(k\vec{v}) + I)\vec{v}$

Observe the following special cases also hold:

1. Let $\vec{s} = \vec{0}$. Then $\vec{0} + kI\vec{v} \equiv -\vec{u} + k(\vec{u}/(k\vec{v}) + I)\vec{v}$.
   - Moreover, let $k = 1$. Then $\vec{v}|\vec{u}$ and $\vec{0} + 1I\vec{v} \equiv -\vec{u} + 1(\vec{u}/\vec{v} + I)\vec{v}$
2. Let $\vec{s} = \vec{0}$ and $\vec{u} = k\vec{v}$. Then $\vec{0} + kI\vec{v} \equiv -k\vec{v} + k(1 + I)\vec{v}$
   - Moreover, let $k = 1$. Then $\vec{u} = \vec{v}$ and $\vec{0} + 1I\vec{v} \equiv -\vec{v} + 1(1 + I)\vec{v}$
3. Let $\vec{s} = \vec{0}$ and $\vec{u} = -k\vec{v}$. Then $\vec{0} + kI\vec{v} \equiv k\vec{v} + k(-1 + I)\vec{v}$
   - Moreover, let $k = 1$. Then $\vec{u} = -\vec{v}$ and $\vec{0} + 1I\vec{v} \equiv \vec{v} + 1(-1 + I)\vec{v}$

The relative ordering of marches formed by adding a position vector to ranges follows the ordering of the ranges. Conversely, the ordering of ranges follows the ordering of marches, formed from those ranges, that additionally share a common position vector.

Lemma 26. Let $\vec{s_1} + k_1 I_1 \vec{x} \in \mathsf{March}^n$ and $k_1 I_1 \vec{x},\ k_2 I_2 \vec{y} \in \mathsf{kInt} \setminus \{\bot, \top\}$. Then $\vec{s_1} + k_1 I_1 \vec{x} \sqsubseteq \vec{s_1} + k_2 I_2 \vec{y}$ iff $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$.

In the concretization of marches, the points described by the range are all added to the position vector. Degenerate marches, which represent a single point, can alternatively be expressed using the vacuous range $0[1, 1]\vec{1}$, henceforth called the zero range. The following lemma substantiates this:

Lemma 27. Let $m = \vec{s} + kI\vec{v} \in \mathsf{March}^n$ where $I = [1, 1]$. Then $m \equiv (\vec{s} + k\vec{v}) + 0[1, 1]\vec{1}$

The following definition lifts a range to a march (and acts as a convenient shorthand in the proofs):

Definition 16. The dimension increase operator $(.)^{+1} : \mathsf{Range}^n \to \mathsf{March}^n$ is defined thus: $kI\vec{v}^{+1} = \vec{0} + kI\vec{v}$.

It can be observed immediately that $\gamma(kI\vec{v}) = \gamma(kI\vec{v}^{+1})$.

While it is always possible to lift a (non-$\top$) range into a march by introducing the zero vector, the converse is not always possible. Every degenerate march can be represented by a degenerate range, albeit possibly in a different syntactic form if the represented point lies on the origin. However, only marches whose points lie on a line that passes through the origin can be replicated by a range. The act of converting a march to a range is formalized thus:

Definition 17. The dimension reduction operator $(.)^{-1} : \mathsf{March}^n \to \mathsf{Range}^n$ is defined thus:

$$(\vec{s} + kI\vec{v})^{-1} = \begin{cases} 0[1,1]\vec{1} & \text{if } I = [1,1] \text{ and } \vec{s} + k\vec{v} = \vec{0} \\ 1[1,1](\vec{s} + k\vec{v}) & \text{else if } I = [1,1] \text{ and } \vec{s} + k\vec{v} \neq \vec{0} \\ 1(\vec{s}/\vec{z} + (k\vec{v})/\vec{z} \cdot I)\vec{z} & \text{else if } \vec{z} = gcd(\vec{s}, k\vec{v}) \neq \perp \\ \top & \text{otherwise} \end{cases}$$

whereas $\top^{-1} = \top$ and $\perp^{-1} = \perp$.

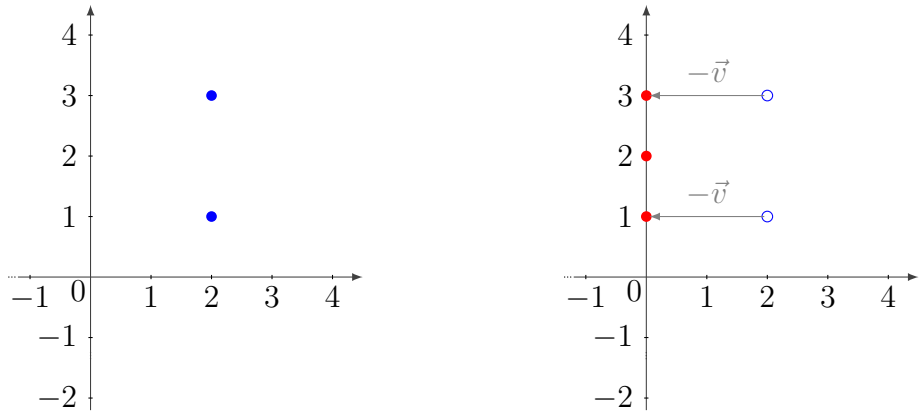
In the third case of the above definition, the existence of a common divisor $\vec{z}$ of the position vector $\vec{s}$ of the march and its $k\vec{v}$ ensures the points described by the march lie on a line that passes through the origin. However, since ranges have fewer degrees of freedom than marches, some loss of information may occur in conversion, as indicated by the presence of scalar multiplication.

The following lemma asserts two properties: the first is that conversion yields a range that encloses the points of the march and the second is that conversion gives a least range satisfying the first property. The properties are formalized below:

Lemma 28. Let $\vec{s} + kI\vec{v} \in \mathsf{Range}^n$. Then:

- $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma((\vec{s} + kI\vec{v})^{-1})$.
- If $r \in \mathsf{Range}^n$ and $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(r)$, then $\gamma((\vec{s} + kI\vec{v})^{-1}) \subseteq \gamma(r)$.

To provide the machinery to formulate join for marches, it is necessary to translate the points of a march and then relax them to a range. To do this, the march translation operator offsets the points by $-\vec{u}$ so that they lie on a line with direction $k\vec{v}$ and then applies dimension reduction:



(a) $m_1 = (2,3) + 1[0,1](0,-2)$ (b) $m_1 \ominus \vec{v}$ where $\vec{v} = (2,0)$

Figure 14: Before (a) and after (b) translation of m_1 by $\vec{v} = (2,0)$

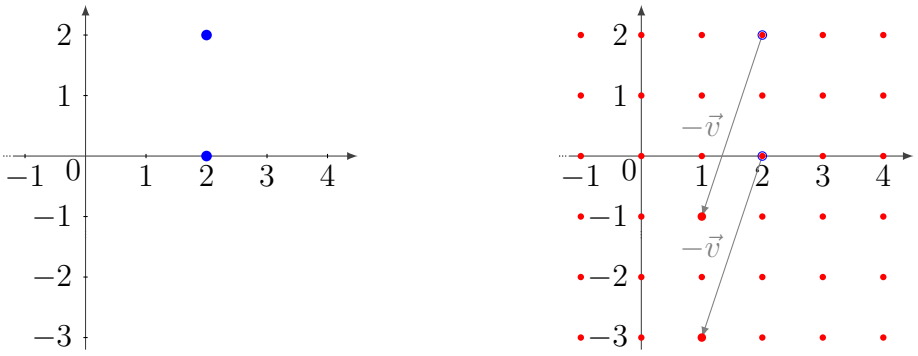
Definition 18. The march translation operator $\ominus : \mathsf{March}^n \times \mathbb{Z}^n \to \mathsf{Range}^n$ is defined thus: $(\vec{s} + kI\vec{v}) \ominus \vec{u} = ((\vec{s} - \vec{u}) + kI\vec{v})^{-1}$ whereas $\bot \ominus \vec{u} = \bot$ and $\top \ominus \vec{u} = \top$.

Example 25. To illustrate march translation, let $m_1 = (2,3) + 1[0,1](0,-2)$ and $\vec{v} = (2,0)$. Observe $gcd((0,3),(0,-2)) = (0,1)$. Then:

$$\begin{aligned} m_1 \ominus \vec{v} &= (((2,3) - (2,0)) + 1[0,1](0,-2))^{-1} \\ &= ((0,3) + 1[0,1](0,-2))^{-1} \\ &= 1((0,3)/(0,1) + (0,-2)/(0,1) \cdot [0,1])(0,1) \\ &= 1(3 + (-2) \cdot [0,1])(0,1) \\ &= 1(3 + [-2,0])(0,1) \\ &= 1[1,3](0,1) \end{aligned}$$

Figure 14 illustrates this march translation. Observe $\gamma(m_1 \ominus \vec{v}) = \gamma(1[1,3](0,1)) = \{(0,1),(0,2),(0,3)\}$. Hence although m_1 describes two points, its translation, $m_1 \ominus \vec{v}$, describes three, demonstrating that translation can induce a loss of information.

Example 26. As a further demonstration of loss of information, consider $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} = (2,-2) + 2[1,2](0,1)$ and suppose $\vec{v} = (1,3)$, as shown in figure 15.



(a) $m_2 = (2,-2) + 2[1,2](0,1)$ (b) $m_2 \ominus \vec{v}$ where $\vec{v} = (1,3)$

Figure 15: Before (a) and after (b) translation of m_2 by $\vec{v} = (1,3)$

Observe $I_2 \neq [1,1]$ and $gcd(\vec{s}_2 - \vec{v}, k_2\vec{y}) = gcd((1,-5),(0,2)) = \perp$, hence case 4 of definition 17 applies. Thus:

$$\begin{aligned} m_2 \ominus \vec{v} &= (((2,-2) - (1,3)) + 2[1,2](0,1))^{-1} \\ &= ((1,-5) + 2[1,2](0,1))^{-1} \\ &= \top \end{aligned}$$

Observe the translated points of m_2 do not lie on a line through the origin, so the only range that can enclose them is $\top$.

Example 27. To illustrate that information is not necessarily lost, consider $m_3 = \vec{s}_3 + k_3 I_3 \vec{m} = (3,-2) + 1[0,2](-1,2)$ and $\vec{v} = (2,0)$. Observe $gcd((1,-2),(-1,2)) = (1,-2)$. Then:

$$\begin{aligned} m_3 \ominus \vec{v} &= (((3,-2) - (2,0)) + 1[0,2](-1,2))^{-1} \\ &= ((1,-2) + 1[0,2](-1,2))^{-1} \\ &= 1((1,-2)/(1,-2) + (-1,2)/(1,-2) \cdot [0,2])(1,-2) \\ &= 1(1 + (-1) \cdot [0,2])(1,-2) \\ &= 1(1 + [-2,0])(1,-2) \\ &= 1[-1,1](1,-2) \end{aligned}$$

Note $\gamma(m_3 \ominus \vec{v}) = \gamma(1[-1,1](1,-2)) = \{(-1,2),(0,0),(1,-2)\}$. Since $k_3\vec{m}$ is a multiple of $\vec{s}_3 - \vec{v}$, dimension reduction incurs no loss of information. The resulting translation is shown in figure 16.

Furthermore, since any single point can be represented by a range, degenerate marches can be translated by any $\vec{v}$ without loss of information: either the first or the second case of definition 17 will apply.

Example 28. Consider the degenerate march $m_4 = \vec{s}_4 + k_4 I_4 \vec{n} = (2,0) + 0[1,1]\vec{1}$ and suppose $\vec{v} = (2,0)$. Since $I_4 = [1,1]$ and $(\vec{s}_4 - \vec{v}) + k_4\vec{v}_4 = \vec{0}$, the first case of definition 17 applies:

$$\begin{aligned} m_4 \ominus \vec{v} &= (((2,0) - (2,0)) + 0[1,1]\vec{1})^{-1} \\ &= ((0,0) + 0[1,1]\vec{1})^{-1} \\ &= 0[1,1]\vec{1} \end{aligned}$$

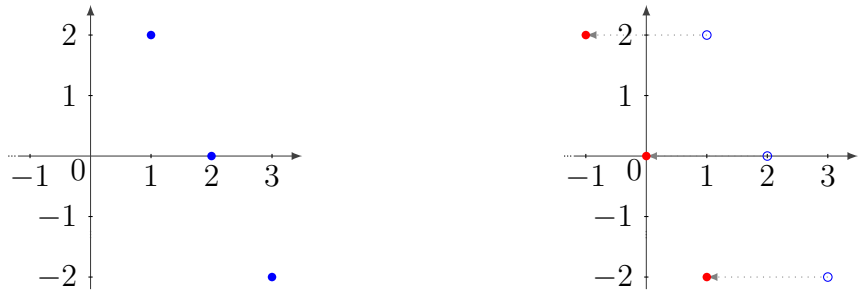
The initial and translated march both represent a single point.

The preceding examples can be summarized by the following proposition:

Proposition 8. Let $m = \vec{s} + kI\vec{v} \in \mathsf{March}^n$ and $\vec{u} \in \mathbb{Z}^n$. Then $m \sqsubseteq \vec{u} + (m \ominus \vec{u})$. Moreover, if $I = [1,1]$ or $k\vec{v} | (\vec{s} - \vec{u})$, then $m \equiv \vec{u} + (m \ominus \vec{u})$.

Examples 25 and 26 correspond to the *then* assertion of proposition 8; examples 27 and 28 correspond to the *moreover* assertion.

Example 29. As a final example of march translation, which provides scaffolding for the development that follows, consider $m_5 = \vec{s}_5 + k_5 I_5 \vec{p} = (-1,-2) + 1[0,2](2,2)$ and $m_6 = \vec{s}_6 + k_6 I_6 \vec{q} = (4,3) + 1[0,1](-4,-4)$. Now suppose $\vec{v} = \vec{s}_6 + k_6\vec{q} = (4,3) + 1(-4,-4) = (0,-1)$. Observe $gcd((-1,-1),(2,2)) = (1,1)$ and



(a) $m_3 = (3,-2) + 1[0,2](-1,2)$ (b) $m_3 \ominus \vec{v}$ where $\vec{v} = (2,0)$

Figure 16: Before (a) and after (b) translation of m_3 by $\vec{v} = (2,0)$

$gcd((4,4),(-4,-4)) = (4,4)$. Then:

$$\begin{aligned} m_5 \ominus \vec{v} &= (((-1,-2) - (0,-1)) + 1[0,2](2,2))^{-1} \\ &= ((-1,-1) + 1[0,2](2,2))^{-1} \\ &= 1((-1,-1)/(1,1) + (2,2)/(1,1) \cdot [0,2])(1,1) \\ &= 1(-1 + 2 \cdot [0,2])(1,1) \\ &= 1(-1 + [0,4])(1,1) \\ &= 1[-1,3](1,1) \end{aligned}$$

and

$$\begin{aligned} m_6 \ominus \vec{v} &= (((4,3) - (0,-1)) + 1[0,1](-4,-4))^{-1} \\ &= ((4,4) + 1[0,1](-4,-4))^{-1} \\ &= 1((4,4)/(4,4) + (-4,-4)/(4,4) \cdot [0,1])(4,4) \\ &= 1(1 + (-1) \cdot [0,1])(4,4) \\ &= 1(1 + [-1,0])(4,4) \\ &= 1[0,1](4,4) \end{aligned}$$

Similar to before, the $\sqsubseteq$ ordering on marches is a quasi-ordering, rather than a partial ordering. To see this, consider $m_1 = (3,0) + 1[1,1](0,3)$ and $m_2 = (0,3) + 1[1,1](3,0)$. Observe $m_1 \neq m_2$ but $m_1 \sqsubseteq m_2$ and $m_2 \sqsubseteq m_1$. Again, this

precludes the existence of a join. The pseudo-join is formulated in terms of the march translation operator $\ominus$:

3.5.1 Pseudo-join

Proposition 9. $\langle \mathsf{March}^n, \sqsubseteq, \sqcup \rangle$ is a pseudo-join-semilattice where:

$$m_1 \sqcup m_2 = \begin{cases} \vec{u} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) & \text{if } I_1 = [1,1] \\ \vec{v} + (m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v}) & \text{otherwise} \end{cases}$$

such that:

- $m_1 = \vec{s}_1 + k_1 I_1 \vec{x} \in \mathsf{March}^n$ and $\vec{u} = \vec{s}_1 + k_1 \vec{x}$
- $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} \in \mathsf{March}^n$ and $\vec{v} = \vec{s}_2 + k_2 \vec{y}$

and $m \sqcup \top = \top \sqcup m = \top$ for all $m \in \mathsf{March}^n$.

The rationale behind march translation is to reduce the join of two marches to the join on two ranges. If $I_1 = [1,1]$, both marches are translated by $\vec{u} = \vec{s}_1 + k_1 \vec{x}$, so that the translation of m_1, $m_1 \ominus \vec{u}$, lies on the origin. The join $m_1 \ominus \vec{u}$ and $m_2 \ominus \vec{u}$ then reduces to adding a zero to the interval of the second range, $m_2 \ominus \vec{u}$, if not already present. The case for $I_2 = [1,1]$ is symmetric. If neither $I_1 = [1,1]$ nor $I_2 = [1,1]$, then there is no advantage conferred by translating by either $\vec{u}$ or $\vec{v}$, so $\vec{v}$ is chosen merely to avoid introducing another case into proposition 9. Finally, a march is recovered from the join of the two ranges by another translation, albeit in the opposite direction.

Example 30. To illustrate join of marches consider $m_1 = \vec{s}_1 + k_1 I_1 \vec{x} = (2,3) + 1[0,1](0,-2)$ and $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} = (2,-2) + 2[1,2](0,1)$ as per examples 25 and 26, whose points are shown in black and blue respectively in the top left graph of figure 17. Since $I_1 \neq [1,1]$ then $\vec{v} = \vec{s}_2 + k_2 \vec{y} = (2,-2) + 2(0,1) = (2,0)$. Observe $m_1 \ominus \vec{v} = 1[1,3](0,1)$ from example 25 and, using similar working, $m_2 \ominus \vec{v} = ((0,-2) + 2[1,2](0,1))^{-1} = 1[0,1](0,2)$. Thus, using the range join working of example 22, $m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v} = r_6 \sqcup r_7 = 1[0,3](0,1)$. Hence $m_1 \sqcup m_2 = \vec{v} + (m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v}) = (2,0) + 1[0,3](0,1)$. The join is shown in red in figure 17.

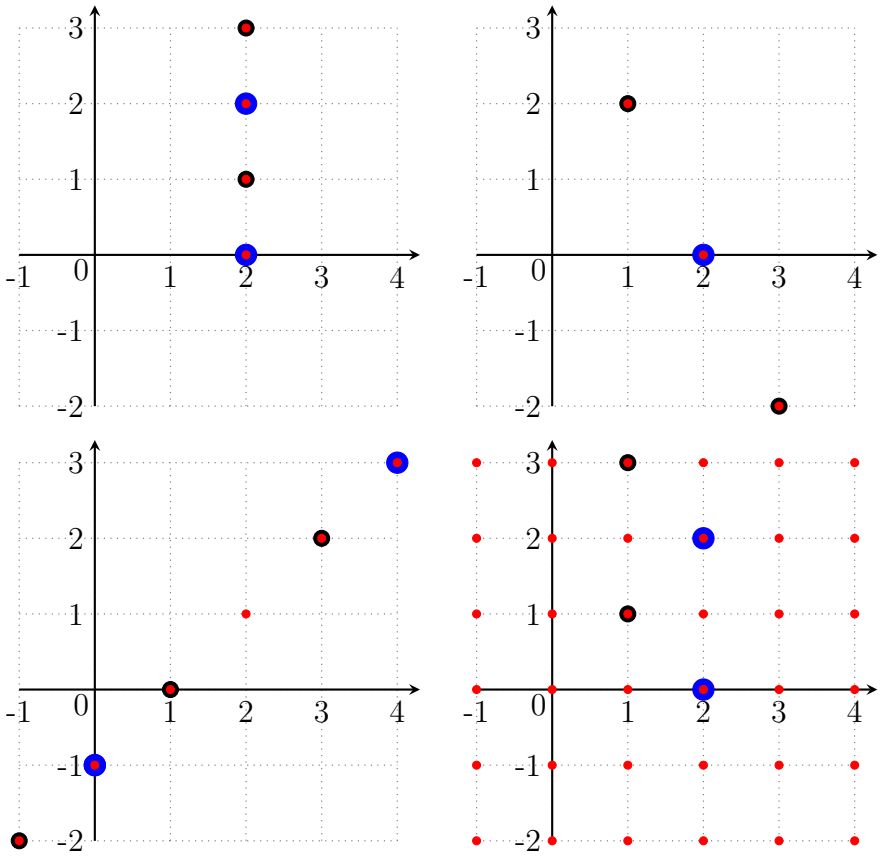
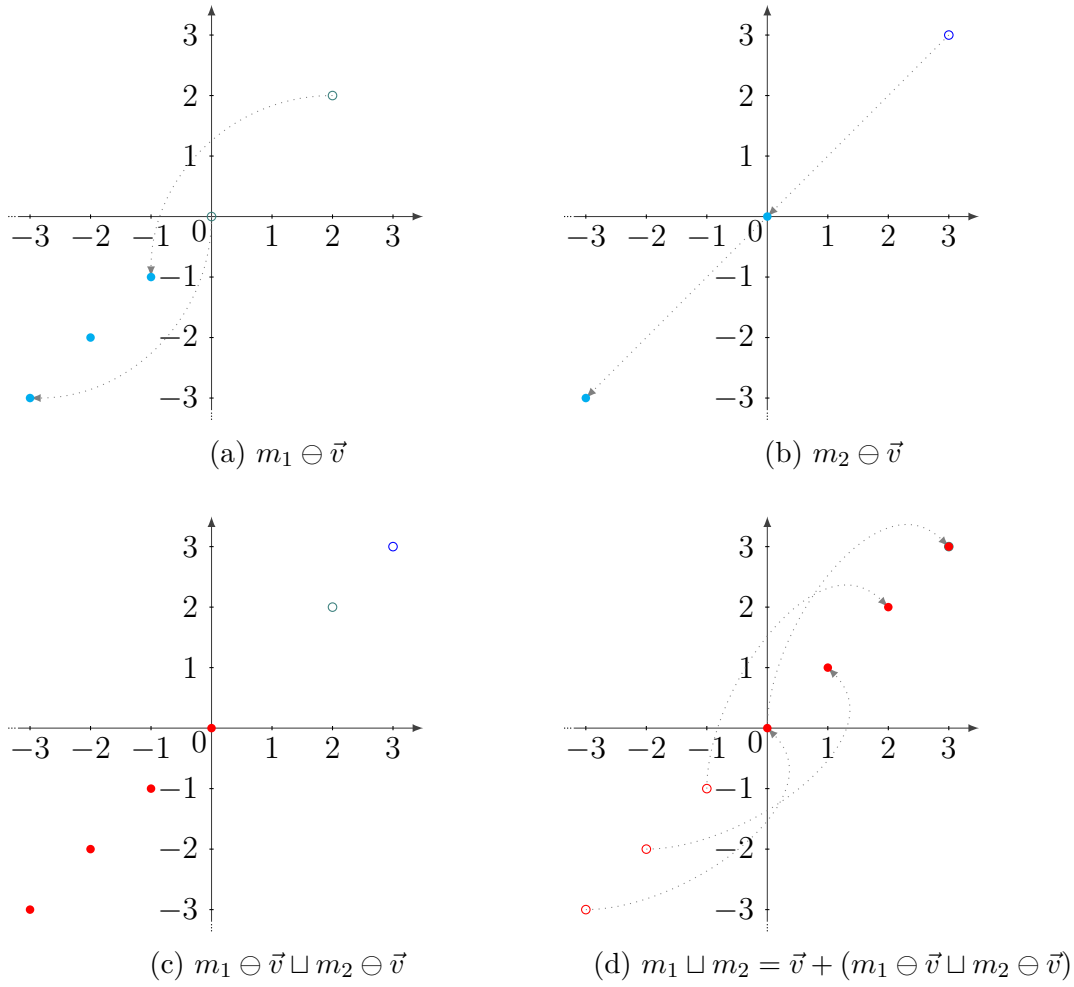


Figure 17: Four marches - top left: $m_1 = (2,3) + 1[0,1](0,-2)$ and (in blue) $m_2 = (2,-2) + 2[1,2](0,1)$, top right: $m_3 = (3,-2) + 1[0,2](-1,2)$ and (in blue) $m_4 = (2,0) + 0[1,1]\vec{1}$, bottom left: $m_5 = (-1,-2) + 1[0,2](2,2)$ and (in blue) $m_6 = (4,3) + 1[0,1](-4,-4)$, bottom right: (in blue) $m_2 = (2,-2) + 2[1,2](0,1)$ and $m_7 = (1,1) + 2[0,1](0,1)$

Example 31. Consider m_3 and m_4 as per examples 27 and 28, whose points are shown in black and blue respectively in the top right graph of figure 17. Then $\vec{v} = \vec{s}_4 + k_4\vec{n} = (2,0) + 0(1,1) = (2,0)$. As shown in the aforementioned examples, $m_3 \ominus \vec{v} = 1[-1,1](1,-2)$ and $m_4 \ominus \vec{v} = 0[1,1]\vec{1}$. But $\gamma(m_4 \ominus \vec{v}) = \gamma(0[1,1]\vec{1}) = \{(0,0)\} \subseteq \{(-1,2),(0,0),(1,-2)\} = \gamma(1[-1,1](1,-2)) = \gamma(m_3 \ominus \vec{v})$. Thus $m_3 \ominus \vec{v} \sqcup m_4 \ominus \vec{v} = m_3 \ominus \vec{v} = 1[-1,1](1,-2)$. Hence $m_3 \sqcup m_4 = \vec{v} + (m_3 \ominus \vec{v} \sqcup m_4 \ominus \vec{v}) = (2,0) + 1[-1,1](1,-2)$, as illustrated by the red dots.

Example 32. Consider $m_5 = (-1,-2)+1[0,2](2,2)$ and $m_6 = (4,3)+1[0,1](-4,-4)$



(a) $m_1 \ominus \vec{v}$ (b) $m_2 \ominus \vec{v}$

(c) $m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v}$ (d) $m_1 \sqcup m_2 = \vec{v} + (m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v})$

Figure 18: Graph of example 25. Note the resultant range has more points than m_1.

from example 29, whose points are shown in black and blue respectively in the lower left graph of figure 17. Then, as shown in example 29, $m_5 \ominus \vec{v} = 1[-1, 3](1, 1)$ and $m_6 \ominus \vec{v} = 1[0, 1](4, 4)$ where $\vec{v} = (0, -1)$. Using a similar method to that shown in, say, example 22, it follows $m_5 \ominus \vec{v} \sqcup m_6 \ominus \vec{v} = 1[-1, 4](1, 1)$. Hence $m_5 \sqcup m_6 = \vec{v} + (m_5 \ominus \vec{v} \sqcup m_6 \ominus \vec{v}) = (0, -1) + 1[-1, 4](1, 1)$.

Example 33. Consider $m_2 = (2, -2) + 2[1, 2](0, 1)$ and $m_7 = (1, 1) + 2[0, 1](0, 1)$. Then $\vec{v} = \vec{s_7} + k_7\vec{r} = (1, 1) + 2(0, 1) = (1, 3)$. First, $m_2 \ominus \vec{v} = \top$, as derived in example 26. Analogous to example 29, it follows $m_7 \ominus \vec{v} = 1[-1, 0](0, 2)$. Thus

$m_2 \ominus \vec{v} \sqcup m_7 \ominus \vec{v} = \top \sqcup 1[-1, 0](0, 2) = \top$. Hence $m_2 \sqcup m_7 = \vec{v} + (m_2 \ominus \vec{v} \sqcup m_7 \ominus \vec{v}) = \vec{v} + \top = \top$. This is shown in the lower right graph of figure 17.

The following corollary, which is essentially a specialization of proposition 9 for the commonly occurring case for where the marches m_1 and m_2 represent points $\vec{s_1}$ and $\vec{s_2}$ respectively, shows how pseudo-join can be computed naturally from the difference $\vec{s_2} - \vec{s_1}$.

Corollary 15. Let $m_1, m_2 \in \mathsf{March}^n$ where $m_1 = \vec{s_1} + 0[1, 1]\vec{1}$ and $m_2 = \vec{s_2} + 0[1, 1]\vec{1}$ where $\vec{s_1} \neq \vec{s_2}$. Then $m_1 \sqcup m_2 = \vec{s_1} + 1[0, 1](\vec{s_2} - \vec{s_1})$.

3.6 Expressiveness of March^n and its limitations

As described, the March^n domain is comprised of elements which ultimately draw from three subdomains: the interval domain, the k-interval domain and the range domain. This section summarizes the expressiveness of the domain, and its subdomains, and notes their limitations.

An interval $[\ell, u]$ represents the integers falling between ℓ and u. The domain Int does not have a distinguished $\top$ element since Int contains $[-\infty, +\infty]$. Through concretization, the interval $[-\infty, +\infty]$ represents all integers, $\mathbb{Z}$, and the element $\bot$ represents no integers. Therefore the interval domain can represent sets of contiguous integers (including single integers), or all integers, or none. An interval is a one-dimensional abstraction and therefore cannot describe areas or volumes.

An element $k[\ell, u]$ of the k-interval domain represents a subset of integers ranging from $k\ell$ to ku, separated by a common difference k. In addition, the distinguished element $\top$ of the k-interval domain represents all integers, $\mathbb{Z}$, and the element $\bot$ represents the empty set of integers. Therefore, an element of the k-interval domain can therefore represent sets of integers separated by a difference k, or all integers, or none. The integers represented by a k-interval, unlike that of an interval, are therefore not necessarily contiguous. An element of the k-interval domain is also a one dimensional abstraction (so no areas or volumes can

be represented) but it provides an extra degree of expressiveness over intervals in that it provides the ability to express integers that are regularly spaced.

An element $kI\vec{v}$ of the range domain represents a set of points in n-dimensional space, formed by the product of the integers of kI and the vector $\vec{v}$. The points lie on a line (not a ray) which is described by the direction of the vector $\vec{v}$ and whose spacing is determined by the common difference k of the k-interval. Unlike the interval and k-interval domain, the element $\top$ of the range domain represents all points in n-dimensional space: it is higher-dimensional rather than one dimensional. The $\top$ element arises when two ranges, $k_1 I_1 \vec{x}$ and $k_2 I_2 \vec{y}$, are joined where there is no greatest common divisor of the vectors, $k_1\vec{x}$ and $k_2\vec{y}$. For one to exist, the vectors must be collinear and the magnitudes of the vectors must share a common factor. Similar to the previous domains, the element $\bot$ represents the empty set. Therefore, an element of the ranges domain can therefore represent a set of points which lie on a line in n-dimensional space, all integers in that space, or none. It is a linear abstraction with three degrees of freedom.

An element $\vec{s} + kI\vec{v}$ of the march domain represents a set of points in n-dimensional space formed by adding $\vec{v}$ to the points of $kI\vec{v}$. The element $\top$ can be formed through the join of two march abstractions m_1 and m_2. There are three places in which $\top$ could occur during the join: first, through the march translation of m_1; second, through the march translation of m_2, or, third, through the join of the ranges resulting from the two translations above. The requirement for the join of the marches not to go to $\top$ is that the points of both marches lie on a common (infinite) line. Unlike ranges, it is not necessary for this line to pass through the origin as the march domain has an extra degree of freedom provided by the presence of the vector $\vec{v}$. Similar to $\top$ in the range domain, $\top$ in the march domain represents all integers in n-dimensional space and $\bot$ represents the empty set. It is a linear abstraction with four degrees of freedom.

To summarize, an element of the interval domain and k-interval domain are one-dimensional abstractions where elements represent subsets of bounded integers with unit and non-unit spacing respectively. The k-interval domain represents

points on a line which passes through the origin and elements of the march domain represent points on a line anywhere in n-dimensional space (but that pass through points that are defined by n-dimensional vectors). The domains therefore represent linear abstractions increasing incrementally from one degree of freedom for the interval domain to four degrees of freedom for the march domain. The march domain, being the ultimate domain introduced here, reflects a periodic progression (of a certain fixed size) of values in n dimensions. These could represent progressions of nested loop counter variables, whose offsets and increments do not necessarily coincide with each other.

3.7 Realization of the abstract domains in code

The domains described above were implemented in ImpactExplorer using Scala. Code snippets are given below for all the main operations of the domains. Importantly, the code directly reflects the mathematical formulations of the domains. All the earlier examples were implemented and checked by running them through ImpactExplorer. These are presented separately in Appendix C.

3.7.1 Abstract Domain

Because the top level abstract domain March^n, is itself formulated from subdomains which are themselves abstract domains, it is useful to specify the behaviour of a generic abstract domain, namely those operations that an abstract domain must implement. The `AbstractDomain[A]` trait is therefore introduced which is parameterized by name `A`. The trait stipulates that an abstract domain is required to implement $\sqsubseteq$, $\sqcup$ and $\sqcap$ where the method names preserve these mathematical symbols through the use of unicode. A fourth method, the negation of $\sqsubseteq$, is defined as a convenient shorthand. A set of method aliases are also included for ease of use when using the operations at a console.

```
package library.maths

trait AbstractDomain[A] { self: A ⇒

  def ⊑(that: A): Boolean
```

```
  def ⋢(that: A): Boolean = !((this ⊑ that) || (this == that))
  def ⊔(that: A): A
  def ⊓(that: A): A

  def sqsse: A ⇒ Boolean = ⊑
  def join: A ⇒ A = ⊔
  def meet: A ⇒ A = ⊓
}
```

The Scala encoding of the abstract domains will now be presented in the same order in which the domains were introduced earlier in the chapter.

3.7.2 The interval abstract domain: Int

First, the Int domain was implemented as follows:

```
sealed trait IntDomain extends AbstractDomain[IntDomain] {

  def ⊑(that: IntDomain): Boolean = IntDomain.⊑(this, that)
  def ⊔(that: IntDomain): IntDomain = IntDomain.⊔(this, that)
  def ⊓(that: IntDomain): IntDomain = IntDomain.⊓(this, that)

}
```

The definitions of three of the methods above ($\sqsubseteq$, $\sqcup$ and $\sqcap$) are delegated to methods defined in the companion object `IntDomain` (where static methods are located). Each of the four methods and some Int-specific methods are now described in turn:

The $\sqsubseteq$ operator is defined thus, where the cases are considered in order:

```
def ⊑ (i1: IntDomain, i2: IntDomain): Boolean =
  (i1, i2) match {
    case (⊥, _)                                                  ⇒ true
    case (Interval(_, _), ⊥)                                     ⇒ false
    case (Interval(opt_l1, opt_u1), Interval(opt_l2, opt_u2))    ⇒ opt_l2 <= opt_l1 &&
                                                                   opt_u1 <= opt_u2
  }
```

Note the final boolean expression relies on a custom implementation of the `<=` operator acting over an arbitrary `optInt: Option[Int]`, as shown below:

```
def <= (optInt2: Option[Int]): Boolean =
  (optInt, optInt2) match {
    case (None, _)          ⇒ true
```

```
    case (_, None)           ⇒ true
    case (Some(x), Some(y))  ⇒ x <= y
  }
```

As will be discussed further in relation to the ⊔ operator, `None` represents $-\infty$ when used as the first value in a pair of arguments and ∞ when used as the second.

The join operator ⊔ is defined thus:

```
def ⊔ (i1: IntDomain, i2: IntDomain): IntDomain =
  (i1, i2) match {
    case (Interval(opt_l1, opt_u1), Interval(opt_l2, opt_u2)) ⇒
      // None (-infty) works naturally here
      val min_l1l2: Option[Int] = Set(opt_l1, opt_l2).min
      // Here we have to take a different approach
      val max_u1u2: Option[Int] =
        if (opt_u1.isEmpty || opt_u2.isEmpty) None
        else Some(Set(opt_u1.get, opt_u2.get).max)
      Interval(min_l1l2, max_u1u2)
    case (⊥, i2)                                          ⇒ i2
    case (i1, ⊥)                                          ⇒ i1
  }
```

The code above is close to the specification for join in the Int domain. However the calculation of the maximum of u_1 and u_2 required special handling. To explain why, it is necessary to explain the structure of an interval `Interval(opt_l1, opt_u1)`. An `Interval(opt_l1, opt_u1)` takes two `Option[Int]` values as arguments. This allows `Some(i)` to represent a specific integer `i`, whereas the absence of a value, `None`, represents either $-\infty$ when provided as the first argument (`opt_l1`) or ∞ when provided as the second argument (`opt_u1`). In Scala, `None < Some(i)` for all integers `i`, so `Set(opt_l1, opt_l2).min` returns the correct `Option` value for the minimum of the lower bounds of the two intervals. However, the maximum of `opt_u1` or `opt_u2` should be `None`, representing ∞, if either value is `None`, otherwise it should be the higher of the two integer values.

The meet operator ⊓ is defined thus:

```
def ⊓ (i1: IntDomain, i2: IntDomain): IntDomain =
  (i1, i2) match {
    case (Interval(opt_l1, opt_u1), Interval(opt_l2, opt_u2)) ⇒
      val max_l1l2: Option[Int] =
```

```
        if(opt_l1.isEmpty || opt_l2.isEmpty) None
        else Some(Set(opt_l1.get, opt_l2.get).max)
      val min_u1u2: Option[Int] = Set(opt_u1, opt_u2).min
      if (max_l1l2 > min_u1u2) ⊥
      else Interval(max_l1l2, min_u1u2)
    case (⊥, _)                                    ⇒ ⊥
    case (_, ⊥)                                    ⇒ ⊥
}
```

The $\sqcap$ operator is defined analogously to the $\sqcup$ operator. The Int domain is the only domain which implements $\sqcap$. The other domains realize $\sqcap$ with `???`, which simply throws a `NotImplementedError` whenever it is called. Since `???` has result type `Nothing`, a subtype of every other type, it satisfies the requirement of the `AbstractDomain[A]` trait. Although it is hard to imagine an abstract domain not equipped with $\sqcap$, it turns out that in the use-case of quantifier introduction, our particular concern, $\sqcap$ was not actually required, hence this partial implementation.

For completeness, the abstract domain operators for k-intervals, ranges and marches are presented in Appendix B in a similar fashion.

3.8 Concluding discussion

Classical intervals and the three steps of domain development presented here represent four different properties of sets of n-dimensional points. First, with the existing classical interval abstract domain, the length or size of an interval manifests itself in the number of represented points. It is the cardinality of the set generated through concretization. Second, the k-value in a k-interval provides a scaling factor. It is always possible to generate the same points from a k-interval using an interval instead (as has been seen, this can be done using the *dot* operator). However, for non-degenerate intervals where k does not equal one, extraneous points will be introduced. Third, the vector in a range abstraction lifts the k-interval abstraction to n dimensions. With the range abstractions in their simplest form, the vector will be irreducible, that is, it will not be divisible by any integer factor other than ± 1. Similar to how any integer can be considered a product of primes, any set of integer points lying on a line that passes through the origin can be constructed from a unique combination of an interval, k-value and irreducible vector – that is, an element of the Range^n domain. Fourth, the vector offset provided

by the March^n domain provides a way of producing sets of points lying on lines which do not pass through the origin. However, unlike the Range^n domain, many elements from the March^n domain are considered equivalent: that is, they represent the same set of points. It is possible to remove this generality by rebasing the range of a march to have a zero lower bound and changing the march vector offset accordingly so that any set of points would have a unique representation under all four domains. However, this remains a theoretical exercise.

The n-dimensional positions of the points are determined by scaled position vectors. Therefore it can be seen in the definitions of various operators that the k-scaling factor from k-intervals and the vectors from the range abstractions are often multiplied together and can be considered unified. But this raises a question: once the range domain is defined with an n-dimensional vector introduced, could the k-value be moved into the range vector so that the k-value part of a k-interval becomes redundant? After all, it would therefore be possible to represent the same sets of points using a combination of an interval and a scaled vector.

Chapter 4

Widening Impact

4.1 Introduction

McMillan's insight [52] was that interpolation offers a vehicle for reducing (ideally minimizing) how much information needs to be tracked for discharging a verification condition. Interpolation realizes the so-called lazy abstraction [38] approach to model checking, where different parts of the computation are analyzed to different degrees, sufficient to verify the property in hand. McMillan's abstract model is an unwinding graph, which is defined in terms of an unwinding tree. An unwinding tree represents the different branches through a program by merging common prefixes of the branches. An unwinding graph is an unwinding tree, augmented with a binary covering relation which stipulates back-arcs between vertices of the graph. The back-arcs mark where the formula at one vertex subsumes the formula at another (where both vertices relate to the same location). The computational value of a back-arc is that it indicates where a subtree does not warrant exploration since it is a variant of another subtree. All these tactics come together in the IMPACT algorithm [52].

The IMPACT algorithm is classically presented in terms of a recursive procedure DFS that walks the unwinding graph and is itself defined in terms of three auxiliary procedures: CLOSE, REFINE and EXPAND. The subtlety of the algorithm is that although COVER always introduces new back-arcs, REFINE can remove back-arcs from the covering relation. (Actually, the interaction is more complex than this

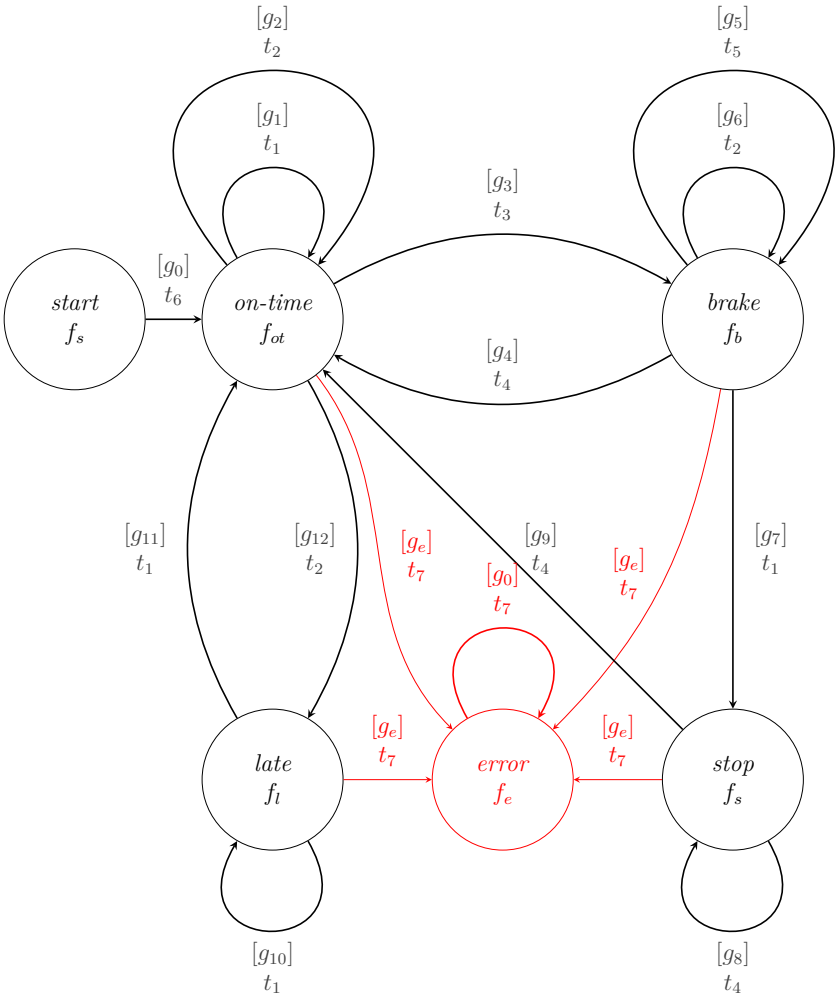


Figure 19: Automaton under analysis

since Cover can also remove back-arcs so as to ensure a covered vertex does not cover others.) The covering relation, therefore, does not grow monotonically and since covering is used to induce termination, the termination behaviour of a run of the algorithm is non-obvious. In fact, as a vertex is covered, and then uncovered, it is not even clear that progress is being made.

In this chapter, Impact is presented from a new perspective by studying the interaction between Cover and Refine, the procedures which check for covering and apply refinement (interpolation). The key concepts of Impact are presented

on demand through the development of a case study. The case study also illustrates how the problematic interaction between covering and refinement can be suppressed by the introduction of quantified formulae, specifically steps 16 to 27. The example gives a series of refinement and back propagation steps which add conjuncts to the formulae that label the nodes of the unwinding graph which prevent covering. Thus the unwinding graph grows without bound and IMPACT fails to terminate. After illustrating this problem, the development changes tack to show how it can be remedied by generalizing the labels to induce covering. Steps 24 to 27 show how quantification can be introduced to compactly represent conjoined formulae which are parameterized by a variable which ranges over an interval. Relaxing one end of the interval strengthens the quantified formula so that it represents all the conjoined formulae and others. The force of this is that strengthening can then add no conjunct to the quantified formula, permitting covering to be observed. Although the case study consists of 39 steps, the presentation is actually a rationalized reconstruction that was distilled from many exploratory (longer) runs.

Visualization of the unwinding graph is key to understanding the operational behaviour of IMPACT. This chapter presents a series of unwinding graphs in a format that is optimized for the printed page. However Appendix D complements this presentation with the same series of graphs where the vertices are fully annotated with their identifier, location and formula since this is better suited to viewing with a preview where the reader can zoom in and out.

4.2 Case study of a train controller

The case study focuses on a control system for a train [4] that is required to stay on time. If the train falls behind schedule, or speeds ahead, the control system takes corrective action by accelerating or decelerating the train accordingly. If it is particularly early, the controller will stop the train for some time before proceeding. The system receives two inputs: the time, given by a wall-clock, and the number of beacons encountered by the train on the line. To be on time, the train must pass a beacon every second. In the formalization of the problem, the variable b represents the number of encountered beacons, s is the wall-clock time in seconds, and d is an

Guard	Definition	Update	Definition
g_0	$\top$	t_1	$b' = b + 1 \wedge d' = d \wedge s' = s$
g_1	$b < s + 9$	t_2	$b' = b \wedge d' = d \wedge s' = s + 1$
g_2	$b > s - 9$	t_3	$b' = b + 1 \wedge d' = 0 \wedge s' = s$
g_3	$b = s + 9$	t_4	$b' = b \wedge d' = 0 \wedge s' = s + 1$
g_4	$b = s + 1$	t_5	$b' = b + 1 \wedge d' = d + 1 \wedge s' = s$
g_5	$d < 9$	t_6	$b' = 0 \wedge d' = 0 \wedge s' = 0$
g_6	$b > s + 1$	t_7	$b' = b \wedge d' = d \wedge s' = s$
g_7	$d = 9$		
g_8	$b > s + 1$		
g_9	$b = s + 1$		
g_{10}	$b < s - 1$		
g_{11}	$b = s - 1$		
g_{12}	$b = s - 9$		
g_e	$(b - s < -20) \vee$ $(b - s > 20)$		

Figure 20: Guard definitions and update definitions for the automaton

auxiliary counter used during braking. The train can be in four states: *on-time*, *brake*, *late* and *stop*. For the train to be on time, $s - 9 < b < s + 9$. If $b = s - 9$, the train is late. It accelerates until $b = s - 1$ where the train is back on time. If $b = s + 9$, the train is early and has to brake and then stop. When $b = s + 1$, the train is again on time. Although not desirable, it is acceptable for the train to be late, but not exceedingly late, or even exceedingly early. This operational requirement is expressed as $|b - s| \leq 20$ or, equivalently, $-20 \leq b - s \leq 20$.

Figure 19 shows the control system rendered as an automaton. Note the automaton has two extra states: *start* and *error*. The *start* state marks the entry point to the automaton in which the variables b, d and s are all unconstrained. The single transition out of *start* into *on-time* consists of a guard g_0 and an update t_6, both given in figure 20. The guard g_0 is vacuous, meaning the transition from *start* to *on-time* is always made, irrespective of the values of the variables. Following [52], guards are delineated with square brackets. The update t_6 sets b, d and s each to zero. The *error* state is reachable from all states except *start*, providing the guard g_e is satisfied, where g_e is the negation of the operational

$$
\begin{array}{lcl}
f_0 & = & \top \\
f_1 & = & r_0 \\
f_2 & = & \bot \\
f_3 & = & r_0 \wedge (d < 9 \rightarrow r_1) \\
f_4 & = & r_0 \wedge (d < 9 \rightarrow r_1) \wedge (d < 8 \rightarrow r_2) \\
f_5 & = & r_0 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 9 \rightarrow r_3)) \\
f_6 & = & r_0 \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 8 \rightarrow r_4)))) \\
f_7 & = & r_0 \wedge (d = 9 \rightarrow r_1) \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 8 \rightarrow r_4)))) \\
f_8 & = & r_0 \wedge (\forall i.i \geq 0 \rightarrow ((9 < d + i) \vee r_3)) \\
f_9 & = & r_0 \wedge (d = 9 \rightarrow r_1) \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 0 \rightarrow ((8 < d + i) \vee r_4))))
\end{array}
$$

where

$$
\begin{array}{lcl}
r_0 & = & -20 \leq b - s \leq 20 \\
r_1 & = & -21 \leq b - s \leq 19 \\
r_2 & = & -22 \leq b - s \leq 18 \\
r_3 & = & -21 \leq b + i - s \leq 19 \\
r_4 & = & -22 \leq b + i - s \leq 18
\end{array}
$$

Figure 21: Formulae f_i as used in the unwinding graphs

requirement. The corresponding transition t_7 is the identity function. The *error* state, and transitions leading into it, are coloured red to differentiate them from the body of the automaton which realizes the controller itself.

4.3 Impact and widening, in 39 steps

This section presents a run-through of how the unwinding graph develops for the case study. Each step, or steps, is presented with the description of the relevant operation of the IMPACT algorithm, complete with an explanation of how the vertices of the unwinding graph are updated. Rather than giving a purely mathematical formulation of IMPACT, intuition is provided on the operations and a commentary on how they interact. However, for clarity, key concepts are, nevertheless, still formulated as mathematical definitions.

Figure 22: Step 0: initialization (see also figure 68)

Figure 23: Step 1: Expansion of 0 (see also figure 69)

4.3.1 Step 0: initialization

The analysis begins at the nominated starting state of the automaton, *start*. The corresponding and initial unwinding vertex in the unwinding graph is labelled 0 where the unique integer identifier assigned is henceforth called the vertex label. (This vertex corresponds to the root labelled ϵ in the classical formulation [52].) By default, a leaf vertex in the unwinding graph is coloured green. The resultant unwinding graph, which contains a single vertex, is shown in figure 22. The unwinding graphs given in the body of the chapter serve to animate the development of the graphs and detail the covering relation. Each vertex is associated with a location and a formula. For vertex 0, for example, the location is *start* and the formula is f_0, as given in figure 21. For the digital archive of the thesis, where the reader can zoom in and out, Appendix D gives all the graphs with the vertices annotated with their identifier, location and formula. (Later in the development, further formulae are introduced, namely f_1, f_2, f_3, f_4, f_5, f_6, f_7 and f_8, which appear first in the unwinding graphs at steps 2, 2, 18, 22, 23, 24, 28 and 30 respectively.) Formulae along branches are strengthened when the error state is reached. As this has not yet occurred, $f_0 = \top$. In fact, since this formula characterizes the state of the entry location, any formula other than $\top$ would indicate that it is possible to reach an error location. The program is deemed safe if IMPACT terminates and the formula for the entry state remains $\top$.

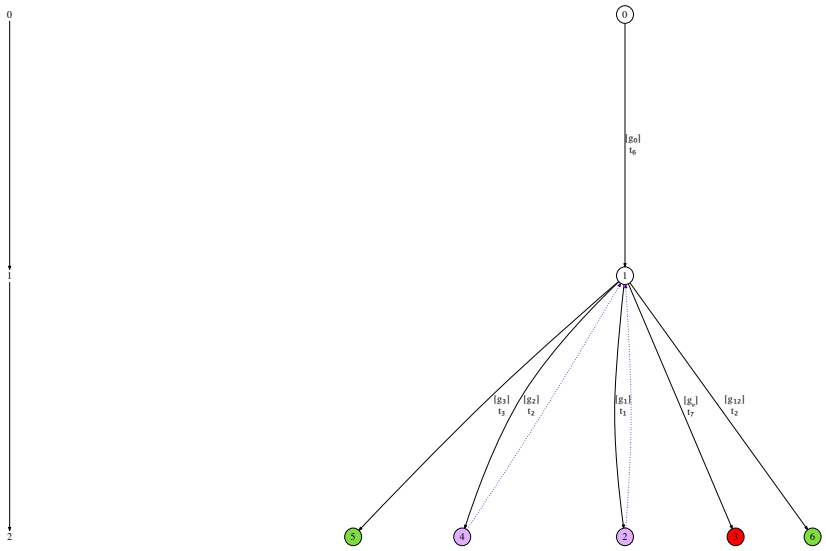


Figure 24: Step 2: Expansion of 1 (see also figure 70)

4.3.2 Step 1: expand 0

Unwinding vertex 0 is expanded. Vertex 0 corresponds to automaton location *start* and the only location reachable from *start* in a single step is *on-time*. Thus the expansion of 0 produces a single vertex, 1, and just one new edge (0, 1). This gives figure 23. The edge is created to simulate the transition from *start* to *on-time* in the automaton. The transition is guarded by the formula g_0 (see figure 20), which is always satisfied. The effect of the transition is described by the formula t_6 which specifies the variables b, d and s are set to 0. The formula at vertex 1 is assigned to $\top$, as is any newly created vertex. Again, this is because the error state has yet to be encountered along the branch.

4.3.3 Step 2: expand 1

After expanding vertex 1, which is automaton location *on-time*, five automaton locations are reachable including the error location. This results in the unwinding graph given in figure 24. Two locations reachable from *on-time* are *on-time* (because of the presence of two self-loops). These two child vertices of *on-time*, vertex 2 and vertex 4, are coloured purple to indicate they are currently covered, in this case by their parent. The error location is reachable from any automaton location;

the automaton transitions to the error state if the variables have an assignment that is considered to be erroneous.

4.3.4 Step 3: refine 3

Expansion of vertex 1 has given rise to an error location at vertex 3, to which the operation refine is applied, as shown in figure 25. Refine strengthens the formulae down a branch of the unwinding graph that leads to the error location to demonstrate its unreachability. If it cannot do so, then the error path is realizable and the automaton is deemed unsafe. IMPACT then ends its analysis.

step 3.1: unfolding the error path To illustrate the refine operation, consider now the error path $\pi = (0, g_0 \wedge t_6, 1), (1, g_e \wedge t_7, 3)$ which leads from 1 to 3. More generally, an error path is a sequence of actions where each action is a triple (ℓ, T, m) where ℓ and m are the entry and exit locations of the action, and T is a transition formula, itself comprised of a conjunction of a guard and an update formula. Next the error path π is unfolded; the unfolding denoted $\mathcal{U}(\pi)$. Unfolding is defined in terms of variable priming: the variables in each transition formula T are primed i times, denoted $T^{<i>}$, where i represents the position of its action in the sequence π. To illustrate:

$$
\begin{aligned}
\mathcal{U}(\pi) &= (g_0 \wedge t_6)^{<0>}, (g_e \wedge t_7)^{<1>} \\
&= (\top \wedge b' = 0 \wedge s' = 0 \wedge d' = 0)^{<0>}, \\
&\qquad\qquad (b - s < -20 \vee b - s > 20 \wedge b' = b \wedge d' = d \wedge s' = s)^{<1>} \\
&= (b' = 0 \wedge s' = 0 \wedge d' = 0), \\
&\qquad\qquad (b' - s' < -20 \vee b' - s' > 20 \wedge b'' = b' \wedge d'' = d' \wedge s'' = s')
\end{aligned}
$$

step 3.2: calculating a sequence interpolant A sequence interpolant for $\mathcal{U}(\pi)$ is then calculated. Interpolation requires three conditions:

Definition 19 (Sequence interpolant)**.** The sequence of formulae $\hat{A}_0, \ldots, \hat{A}_n$ is a sequence interpolant for $\mathcal{U}(\pi) = A_1, \ldots, A_n$ if:

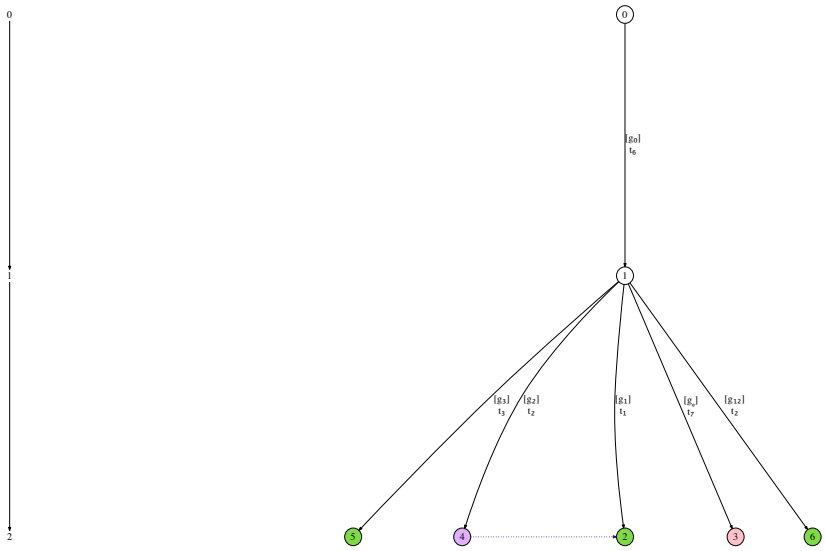


Figure 25: Step 3: Refinement of 3 (see also figure 71)

- $\hat{A}_0 = \top$ and $\hat{A}_n = \bot$
- for all $1 \leq i \leq n$, $\hat{A}_{i-1} \wedge A_i \models \hat{A}_i$
- for all $1 \leq i < n$, $\hat{A}_i \in \mathcal{L}(A_1, \ldots, A_i) \cap \mathcal{L}(A_{i+1}, \ldots, A_n)$

that is, $\hat{A}_i$ can only be defined in the variables common to both the prefix $A_1, \ldots, A_i$ and the suffix $A_{i+1}, \ldots, A_n$.

To illustrate, let $\mathcal{U}(\pi) = A_1, A_2$, where:

$$\begin{aligned} A_1 &= b' = 0 \wedge s' = 0 \wedge d' = 0 \\ A_2 &= (b' - s' < -20 \vee b' - s' > 20) \wedge b'' = b' \wedge d'' = d' \wedge s'' = s' \end{aligned}$$

To see

$$\hat{A}_0 = \top, \quad \hat{A}_1 = -20 \leq b' - s' \leq 20, \quad \hat{A}_2 = \bot$$

is a sequence interpolant, first observe:

$$\hat{A}_0 \wedge A_1 = A_1 \models b' = 0 \wedge s' = 0 \models -20 \leq b' - s' \leq 20 = \hat{A}_1$$

Observe $\hat{A}_1$ is the weakest formula that when combined with A_1 entails $\hat{A}_2$. This is no coincidence because the sequence interpolant is calculated by inferring weakest

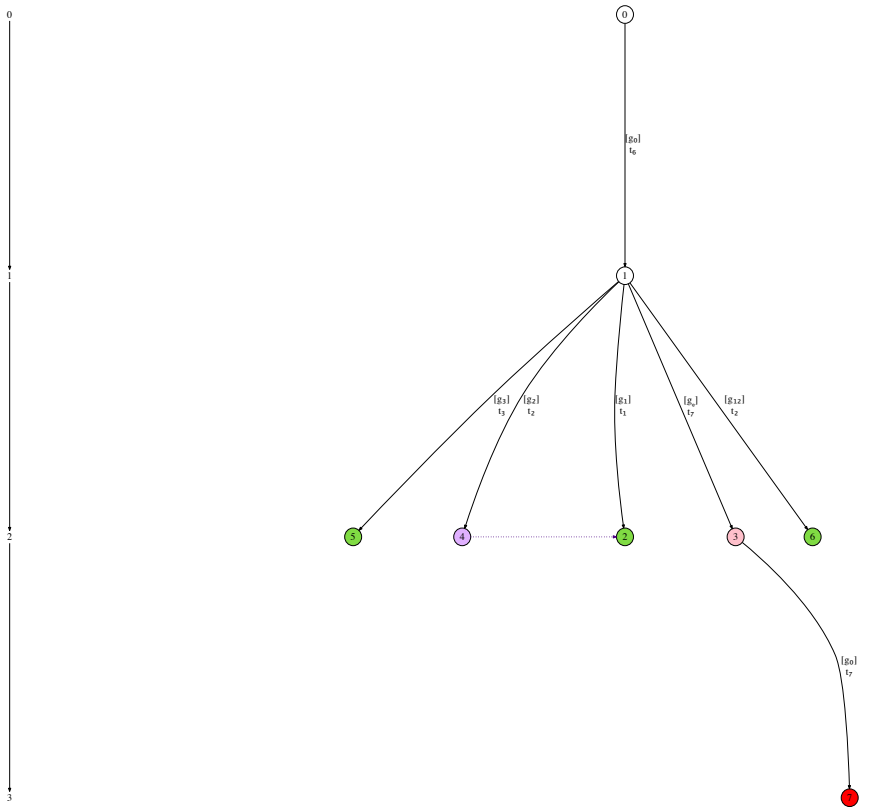


Figure 26: Step 4: Expansion of 3 (see also figure 72)

preconditions, although other strategies are applicable.

Next, observe $\hat{A}_1 \wedge b' - s' < -20 \models \bot$ and likewise $\hat{A}_1 \wedge b' - s' > 20 \models \bot$. Therefore:

$$\hat{A}_1 \wedge A_2 \models \hat{A}_1 \wedge (b' - s' < -20 \vee b' - s' > 20) \models \bot = \hat{A}_2$$

Since an interpolant exists, the error path is not realizable and analysis proceeds by updating the unwinding graph using the sequence interpolant.

step 3.3: updating the unwinding graph The unwinding graph is updated using formulae obtained by unpriming the sequence interpolant. Given a formula F, the unpriming operation, denoted $F^{<-i>}$ removes i levels of priming from each variable of F.

Consider again $\pi = (0, g_0 \wedge t_6, 1), (1, g_e \wedge t_7, 3)$ and observe that 0, 1 and 3

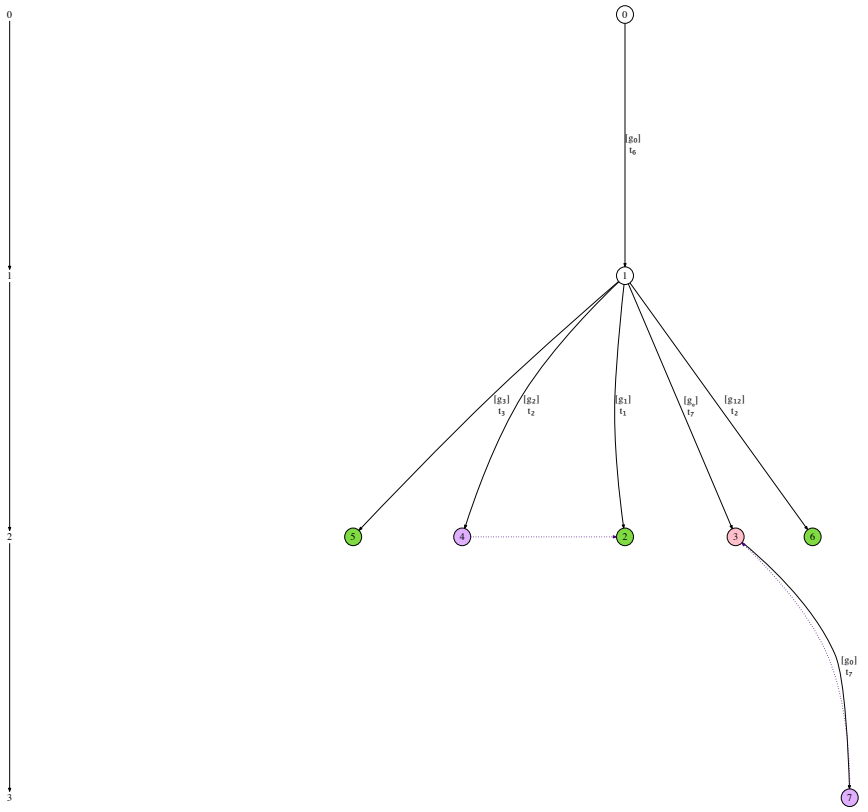


Figure 27: Step 5: Refinement of 7 (see also figure 73)

correspond to vertices in the unwinding graph. These vertices have formulae f_0, f_0 and f_0 respectively where $f_0 = \top$, and the formulae are assigned the names v_0, v_1 and v_2 to indicate their position down a branch. Hence, $v_0 = v_1 = v_2 = f_0$. These formulae are strengthened during refinement. If $v_i \not\models \hat{A}_i^{<-i>}$, then the formula v_i is updated to be $v_i \wedge \hat{A}_i^{<-i>}$. This strengthens v_i to incorporate the requirement of the interpolant $\hat{A}_i^{<-i>}$. At position 0, $v_0 = f_0 = \top \models \top = \hat{A}_0^{<-0>} = \hat{A}_0$. However at position 1, $v_1 = f_0 = \top \not\models -20 \leq b - s \leq 20 = \hat{A}_1^{<-1>}$. Thus v_1 is strengthened to $f_0 \wedge \hat{A}_1^{<-1>} = \hat{A}_1^{<-1>} = -20 \leq b - s \leq 20$. This new formula in the unwinding graph is f_1, since f_0 is already defined. At position 2, $v_2 = f_0 \not\models \bot = \hat{A}_1^{<-2>}$, so v_2 is updated to $f_0 \wedge \bot = \bot$. Since this too is new, it is given the label f_2. The net effect is to replace the formulae v_0, v_1, v_2 down a branch in the unwinding graph with f_0, f_1, f_2 (where the correspondence in the indices is entirely coincidental).

step 3.4: removal of covering arcs (back-arcs) Strengthening a branch can interfere with covering arcs, which are auxiliary data structures used to infer that the analysis has finished. Specifically, a covering arc is used to elide expansion of a vertex by indicating the presence of another vertex, corresponding to the same automaton location, whose formula is more general. Only the latter vertex requires analysis since its state subsumes the former. The formal concept of covering encapsulates these requirements:

Definition 20. A vertex w covers vertex v iff vertices v and w correspond to the same automaton location and the formula of vertex v entails that of vertex w.

Another perspective on covering is that the exploration of a covered vertex would not ultimately expose any behaviour beyond that found by exploring the covering vertex. Specifically, if the error location cannot be reached from the covering vertex, then it cannot be reached from the covered vertex. Conversely, ruling out an error from a covered vertex does not avoid an analysis of the covering vertex.

Before refinement, f_0 held at vertices 1, 2 and 4, all of which corresponded to the location *on-time*. Covering arcs (2, 1) and (4, 1) can therefore be inserted, though when and where an arc is introduced is left open in [52]. During refinement, once strengthening is complete, covering arcs that ended in strengthened vertices are removed, since the covering relation may no longer hold. Since v_1 (vertex 1) was strengthened, both covering arcs (2, 1) and (4, 1) are dropped.

4.3.5 Step 4: expand 3 and Step 5: refine 7

The algorithm presented in [52] terminates only when all leaves are covered. Since error vertex 3 is itself a leaf, it is therefore necessary for it to be covered. To do so, another vertex corresponding to the error location, having $\bot$ as its formula, must exist in the unwinding graph. However, since no such vertex exists, one would proceed by expanding 3, so that 3 is no longer a leaf, circumventing the need for covering. Expansion results in figure 26. Vertex 7, the newly added error vertex, is generated by transitioning from the automaton error location, back to itself, through a self-loop. The pronounced arc leading to vertex 7 is a by-product of the way the graphs are displayed. Graphs are rendered in such a way that

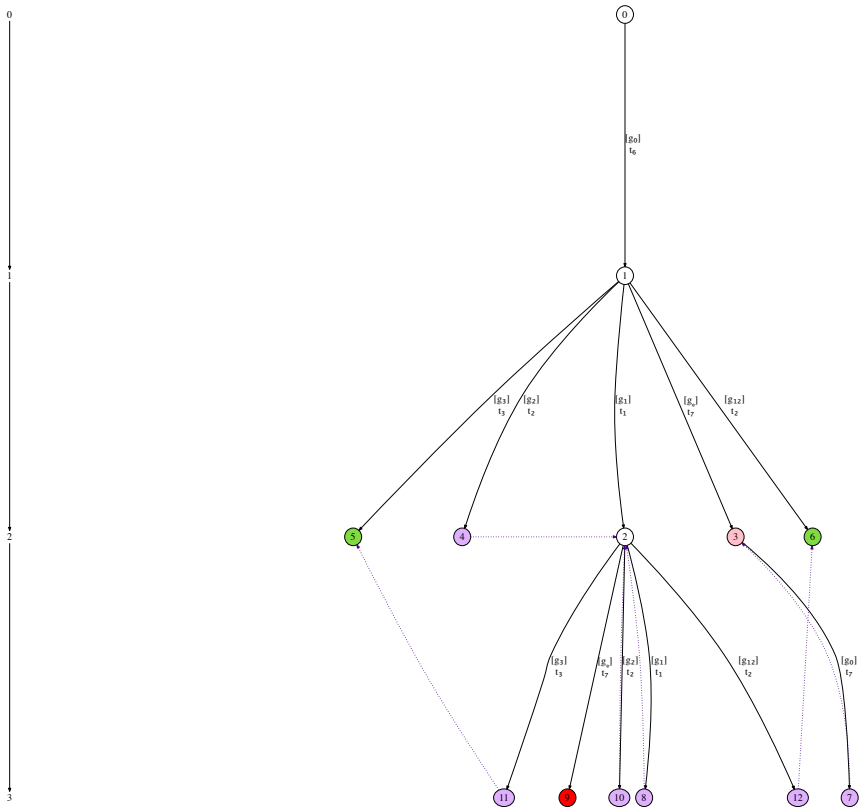


Figure 28: Step 6: Expansion of 2 (see also figure 74)

the relative positions of vertices are stable across different unwinding graphs that arise at different steps in the development. This makes it easier to observe how the unwinding graphs change from one step to the next.

Refinement gives figure 27. However 7 is itself a leaf so has the same requirement for covering. However, unlike previously, a vertex now exists which could cover it, namely vertex 3. Vertex 7 is then refined with the aim of strengthening its formula to $\bot$, necessary for it to be covered by 3. This commentary amplifies the oblique explanation [52, section 2] "we will assume that every location has at least one outgoing action. This can be made true without affecting program safety by adding self-loops."

To refine 7, first its error path $\pi = (0, g_0 \wedge t_6, 1), (1, g_e \wedge t_7, 3), (3, g_0 \wedge t_7, 7)$ is

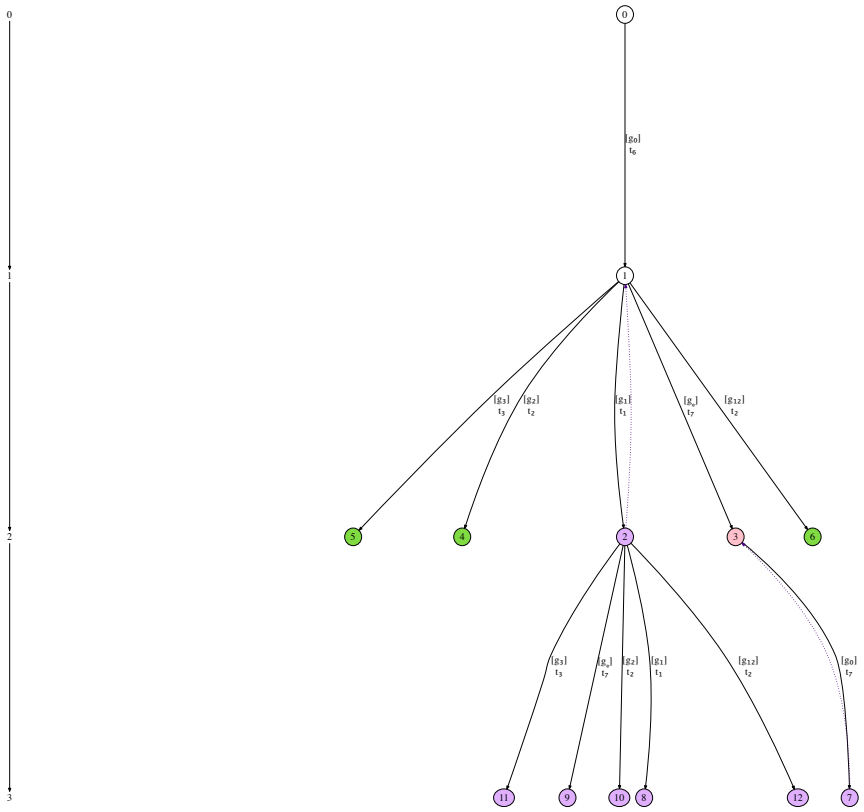


Figure 29: Step 7: Refinement of 9 (see also figure 75)

unfolded to:

$$
\begin{aligned}
\mathcal{U}(\pi) &= (g_0 \wedge t_6)^{<0>}, (g_e \wedge t_7)^{<1>}, (g_0 \wedge t_7)^{<2>} \\
&= (\top \wedge b' = 0 \wedge s' = 0 \wedge d' = 0)^{<0>}, \\
&\qquad (b - s < -20 \vee b - s > 20 \wedge b' = b \wedge d' = d \wedge s' = s)^{<1>}, \\
&\qquad (\top \wedge b' = b \wedge d' = d \wedge s' = s)^{<2>} \\
&= (b' = 0 \wedge s' = 0 \wedge d' = 0), \\
&\qquad (b' - s' < -20 \vee b' - s' > 20 \wedge b'' = b' \wedge d'' = d' \wedge s'' = s'), \\
&\qquad (\top \wedge b''' = b'' \wedge d''' = d'' \wedge s''' = s'')
\end{aligned}
$$

where 0 corresponds to automaton location *start*, 1 to *on-time* and 3 and 7 to *error*. Next, from the unfolded error path, a sequence interpolant $\hat{A}_0, \hat{A}_1, \hat{A}_2, \hat{A}_3$ is calculated. This yields $\hat{A}_0 = \top$, $\hat{A}_1 = -20 \leq b' - s' \leq 20$ and $\hat{A}_2 = \hat{A}_3 = \bot$.

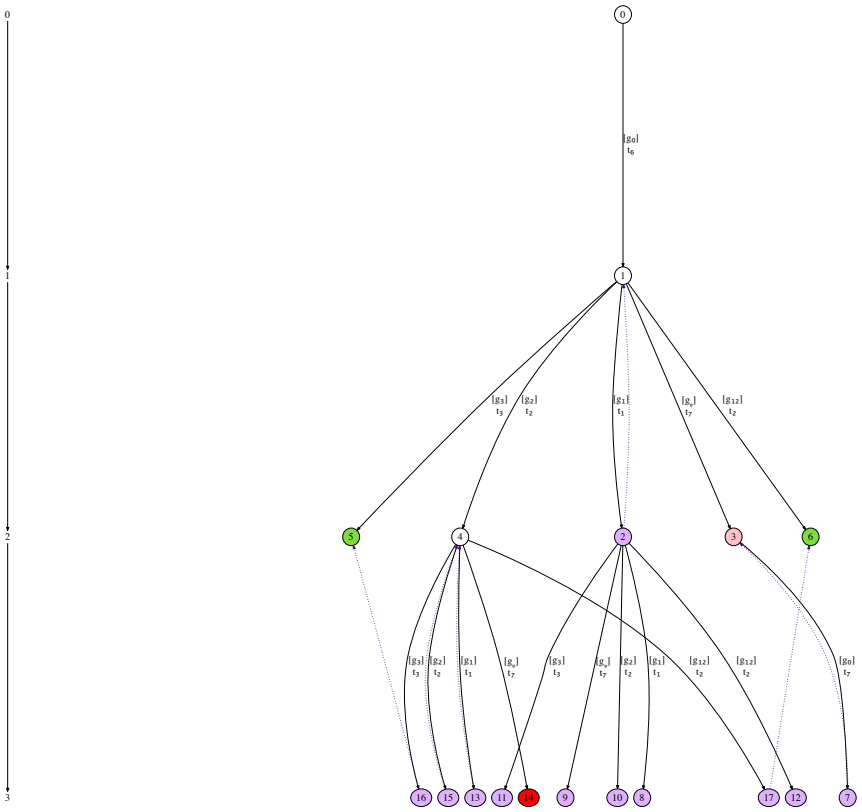


Figure 30: Step 8: Expansion of 4 (see also figure 76)

Similar to before, the formula at the vertex at position i in the error path is named v_i. Thus $v_0 = f_0$, $v_1 = f_1$, $v_2 = f_2$ and $v_3 = f_0$. Now since $\hat{A}_0 = \top \models f_0 = v_0$, and $\hat{A}_1^{<-1>} = -20 \leq b - s \leq 20 \models f_1 = v_1$, and, furthermore, $\hat{A}_2^{<-2>} = \bot \models f_2 = v_2$, no strengthening of v_0, v_1 or v_2 is required. However $\hat{A}_3^{<-3>} = \bot \not\models \top = f_0 = v_3$, so v_3 is strengthened to $v_3 \wedge \bot = \bot = f_2$. Since f_2 holds at vertex 3, a covering arc (7, 3) is added, so that 7 is covered, as desired. (A technical point is that whenever a covering arc (v, w) is added, then any covering arc that ends in v or a descendant of v must be removed. However, since there are no covering arcs below 7, this follows vacuously.)

4.3.6 Step 6: expand 2 and Step 7: refine 9

Now that error location 3 has been refined, and its branch closed off by the covering of its child vertex, the remaining leaves in the graph need to be considered. Vertex 2

is arbitrarily chosen for expansion and its resultant child error location, vertex 9, is refined. Expansion and refinement gives figures 28 and 29 respectively. The error path $\pi = (0, g_0 \wedge t_6, 1), (1, g_1 \wedge t_1, 2), (2, g_e \wedge t_7, 9)$ is unfolded to:

$$\begin{aligned}\mathcal{U}(\pi) &= (g_0 \wedge t_6)^{<0>}, (g_1 \wedge t_1)^{<1>}, (g_e \wedge t_7)^{<2>} \\ &= (b' = 0 \wedge s' = 0 \wedge d' = 0), \\ &\qquad (b' < s' + 9 \wedge b'' = b' + 1, d'' = d' \wedge s'' = s), \\ &\qquad (b'' - s'' < -20 \vee b'' - s'' > 20 \wedge b''' = b'' \wedge d''' = d'' \wedge s''' = s'')\end{aligned}$$

where 0 corresponds to automaton location *start*, 1 and 2 to *on-time* and 9 to *error*. The formulae v_0, v_1, v_2 and v_3 are those at 0, 1, 2 and 9. That is, $v_0 = f_0$, $v_1 = f_1$, $v_2 = f_0$ and $v_3 = f_0$. Next, from the unfolded error path, a sequence interpolant $\hat{A}_0, \hat{A}_1, \hat{A}_2, \hat{A}_3$ is calculated, yielding $\hat{A}_0 = \top$, $\hat{A}_1 = (b' - s' < 9) \rightarrow (-21 \leq b' - s' \leq 19)$, $\hat{A}_2 = -20 \leq b'' - s'' \leq 20$ and $\hat{A}_3 = \bot$. Since $\hat{A}_0 = \top \models f_0 = v_0$ and $\hat{A}_1^{<-1>} = (b - s < 9) \rightarrow (-21 \leq b - s \leq 19) \models f_1 = v_1$ then no strengthening occurs of v_0 or v_1. The formula at 2, v_2, $f_0 = \top$, is strengthened to $f_0 \wedge \hat{A}_2^{<-2>} = -20 \leq b - s \leq 20 = f_1$, causing the removal of covering arc (4, 2). The formula at 9, v_3, is strengthened to $\bot$. Since now $v_2 = f_1 \models f_1 = v_1$ and 1 and 2 both correspond to *on-time*, then a covering arc (2, 1) is added. Also the covering arcs (11, 5) and (12, 6), based at child vertices of 2, are removed in line with the technical point mentioned in the previous step. Since 2 is now a covered leaf, its development is now paused.

4.3.7 Step 8: expand 4 and Step 9: refine 14

With the removal of the covering arc (4, 2) in the previous step, vertex 4 is now an uncovered leaf. Thus it is expanded to give error vertex 14, shown in figure 30, which is subsequently refined. Refinement then gives figure 31. The error path is $\pi = (0, g_0 \wedge t_6, 1), (1, g_2 \wedge t_2, 4), (4, g_e \wedge t_7, 14)$ where v_0, v_1, v_2, v_3 are the formulae holding at 0, 1, 4, 14 respectively. Thus $v_0 = f_0$, $v_1 = f_1$, $v_2 = f_0$ and $v_3 = f_0$. The resultant sequence interpolant is the same to above, save $\hat{A}_1 = (-9 < b' - s') \rightarrow (-19 \leq b' - s' \leq 21)$. It holds $\hat{A}_1^{<-1>} \models f_1 = v_1$ so v_1 remains as is. However, $v_2 = f_0 = \top$ is strengthened to f_1 and, since both 4 and 1 correspond to *on-time*

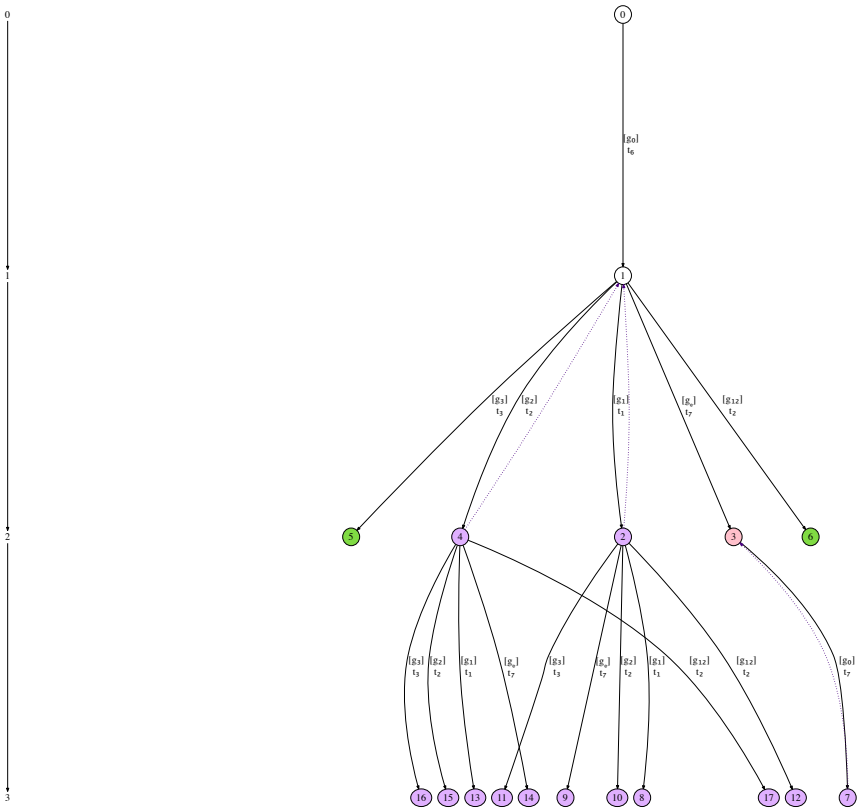


Figure 31: Step 9: Refinement of 14 (see also figure 77)

and $v_1 = f_1$, a covering arc (4, 1) is added. In addition v_3 is strengthened to f_2, but since 4, which the parent of 14, is covered, then it follows 14 is also covered. Expansion is therefore paused for 4 and 14.

4.3.8 Steps 10-15: expand 6, refine 19, ..., refine 25

Steps 10 to 15 interleave expansion and refinement steps repeating the strategy previously illustrated. For brevity, the resultant graphs for only steps 10 and 15 are shown in figures 32 and 37 respectively. Figure 33 summarizes the operations for steps 10 to 15 (and others). In the ops columns, 'e' and 'r' abbreviate expand and refine respectively. The 'set' operation relates to the widening step which is explained in section 4.3.12. For reproducibility, figure 34 shows the calculation of the sequence interpolant and how a branch is refined: the first column is the step and the second column π is a numerical identifier which abbreviates an error

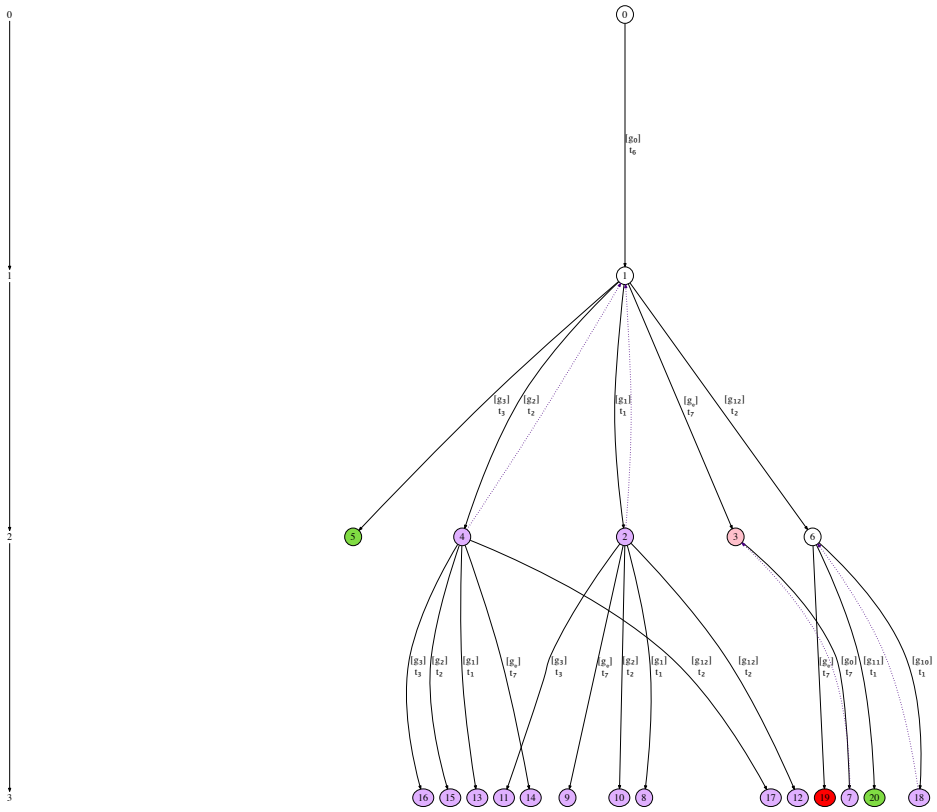


Figure 32: Step 10: Expansion of 6 (see also figure 78)

path. Figure 35 maps the identifier to the triples which make up the path. The v_i row of figure 34 records the formulae along a branch from the root to the error vertex. The $\hat{A}_i^{\langle -i \rangle}$ row gives the unprimed interpolants that make up the sequence interpolant. The f_j and a_k formulae, which appear as interpolants, are defined in figures 21 and 36 respectively. The a_k formulae appear only in a sequence interpolant; they never feature in an unwinding graph. The v_i' formulae record the updated formulae of the branch indicating where strengthening is applied to v_i.

The steps taken up to this point result in the unwinding graph given in figure 37. Three branches have been covered and potentially fully processed. Recall that the covered vertices are purple. One branch has a leaf, 5, corresponding to location *brake*, which is coloured green, indicating it remains to be considered. This is a significant point in the analysis because if this branch is processed, and in doing so does not strengthen the formula at vertex 1, then the analysis is complete. It is

op	step	op	step	op	step	op	step
	0	e 6	10	e 33	20	set 32 g_2	30
e 0	1	r 19	11	r 40	21	r 49	31
e 1	2	e 18	12	e 37	22	set 33 g_2	32
r 3	3	r 22	13	r 45	23	r 50	33
e 3	4	e 20	14	set 32 g_1	24	e 51	34
r 7	5	r 25	15	r 49	25	r 55	35
e 2	6	e 5	16	set 33 g_1	26	e 31	36
r 9	7	r 30	17	r 50	27	r 58	37
e 4	8	e 32	18	e 29	28	e 53	38
r 14	9	r 35	19	r 52	29	r 63	39

where

$$g_1 = (-20 \leq b - s \leq 20) \wedge \forall i \geq 0.((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$
$$g_2 = (-20 \leq b - s \leq 20) \wedge \forall i \geq 0.((d \leq 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$

Figure 33: Operations used in analysis of automaton

worth noting up to this point, only three formula feature in the unwinding graph, $f_0 = \top, f_1 = -20 \leq b - s \leq 20$ and $f_2 = \bot$ (as shown in the table in figure 34). In particular, only f_1 is non-trivial and even this can be represented compactly.

4.3.9 Steps 16-17: expand 5, refine 30

In contrast to the compact formulae seen up until now, the development of vertex 5 for the *brake* location generates formulae which grow progressively larger and, without intervention, impede termination of analysis itself.

To see this consider the expansion of vertex 5 and the subsequent refinement of vertex 30 (and steps which follow on). Expansion is shown in figure 38 and refinement in figure 39. After refinement, f_1 holds at vertex 5 which corresponds to *brake*. Two of the children of vertex 5, vertices 32 and 33, also correspond to *brake* and need to be covered, say by 5. Thus expansion resumes on vertex 32 with this objective in mind.

Step	π	i	0	1	2	3	4	5
2	3	v_i	f_0	f_0	f_0			
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_1	f_2			
		v'_i	f_0	f_1	f_2			
4	7	v_i	f_0	f_1	f_2	f_0		
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_1	f_2	f_2		
		v'_i	f_0	f_1	f_2	f_2		
6	9	v_i	f_0	f_1	f_0	f_0		
		$\hat{A}_i^{\langle -i \rangle}$	f_0	a_3	f_1	f_2		
		v'_i	f_0	f_1	f_1	f_2		
8	14	v_i	f_0	f_1	f_0	f_0		
		$\hat{A}_i^{\langle -i \rangle}$	f_0	a_4	f_1	f_2		
		v'_i	f_0	f_1	f_1	f_2		
10	19	v_i	f_0	f_1	f_0	f_0		
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	f_1	f_2		
		v'_i	f_0	f_1	f_1	f_2		
12	22	v_i	f_0	f_1	f_1	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_5	f_1	f_2	
		v'_i	f_0	f_1	f_1	f_1	f_2	
14	25	v_i	f_0	f_1	f_1	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	f_0	f_1	f_2	
		v'_i	f_0	f_1	f_1	f_1	f_2	
16	30	v_i	f_0	f_1	f_0	f_0		
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	f_1	f_2		
		v'_i	f_0	f_1	f_1	f_2		
18	35	v_i	f_0	f_1	f_1	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_6	f_1	f_2	
		v'_i	f_0	f_1	f_3	f_1	f_2	
20	40	v_i	f_0	f_1	f_3	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_7	f_1	f_2	
		v'_i	f_0	f_1	f_3	f_1	f_2	

Step	π	i	0	1	2	3	4	5
22	45	v_i	f_0	f_1	f_3	f_1	f_0	f_0
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_8	a_6	f_1	f_2
		v'_i	f_0	f_1	f_4	f_3	f_1	f_2
24	49	v_i	f_0	f_1	f_4	f_5	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_9	f_5	f_2	
		v'_i	f_0	f_1	f_6	f_5	f_2	
26	50	v_i	f_0	f_1	f_6	f_5	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_{11}	f_5	f_2	
		v'_i	f_0	f_1	f_6	f_5	f_2	
28	52	v_i	f_0	f_1	f_6	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_{12}	f_1	f_2	
		v'_i	f_0	f_1	f_7	f_1	f_2	
30	49	v_i	f_0	f_1	f_7	f_8	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_{13}	f_8	f_2	
		v'_i	f_0	f_1	f_9	f_8	f_2	
32	50	v_i	f_0	f_1	f_9	f_8	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_{15}	f_8	f_2	
		v'_i	f_0	f_1	f_9	f_8	f_2	
34	55	v_i	f_0	f_1	f_9	f_1	f_0	f_0
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	a_{16}	a_7	f_1	f_2
		v'_i	f_0	f_1	f_9	f_1	f_1	f_2
36	58	v_i	f_0	f_1	f_9	f_0	f_0	
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	f_0	f_1	f_2	
		v'_i	f_0	f_1	f_9	f_1	f_2	
38	63	v_i	f_0	f_1	f_9	f_1	f_0	f_0
		$\hat{A}_i^{\langle -i \rangle}$	f_0	f_0	f_0	f_0	f_1	f_2
		v'_i	f_0	f_1	f_9	f_1	f_1	f_2

Figure 34: Calculating and using the sequence interpolants

π					
3	$(0, g_0 \wedge t_6, 1)$	$(1, g_e \wedge t_7, 3)$			
7	$(0, g_0 \wedge t_6, 1)$	$(1, g_e \wedge t_7, 3)$	$(3, g_0 \wedge t_7, 7)$		
9	$(0, g_0 \wedge t_6, 1)$	$(1, g_1 \wedge t_1, 2)$	$(2, g_e \wedge t_7, 9)$		
14	$(0, g_0 \wedge t_6, 1)$	$(1, g_2 \wedge t_2, 4)$	$(4, g_e \wedge t_7, 14)$		
19	$(0, g_0 \wedge t_6, 1)$	$(1, g_{12} \wedge t_2, 6)$	$(6, g_e \wedge t_7, 19)$		
22	$(0, g_0 \wedge t_6, 1)$	$(1, g_{12} \wedge t_2, 6)$	$(6, g_{10} \wedge t_1, 18)$	$(18, g_e \wedge t_7, 22)$	
25	$(0, g_0 \wedge t_6, 1)$	$(1, g_{12} \wedge t_2, 6)$	$(6, g_{11} \wedge t_1, 20)$	$(20, g_e \wedge t_7, 25)$	
30	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_e \wedge t_7, 30)$		
35	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_5 \wedge t_5, 32)$	$(32, g_e \wedge t_7, 35)$	
40	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_6 \wedge t_2, 33)$	$(33, g_e \wedge t_7, 40)$	
45	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_5 \wedge t_5, 32)$	$(32, g_5 \wedge t_5, 37)$	$(37, g_e \wedge t_7, 45)$
49	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_5 \wedge t_5, 32)$	$(32, g_{14} \wedge t_7, 49)$	
50	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_6 \wedge t_2, 33)$	$(33, g_{14} \wedge t_7, 50)$	
52	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_7 \wedge t_1, 29)$	$(29, g_e \wedge t_7, 52)$	
49	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_5 \wedge t_5, 32)$	$(32, g_{15} \wedge t_7, 49)$	
50	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_6 \wedge t_2, 33)$	$(33, g_{15} \wedge t_7, 50)$	
55	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_7 \wedge t_1, 29)$	$(29, g_6 \wedge t_4, 51)$	$(51, g_{err} \wedge t_7, 55)$
58	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_4 \wedge t_4, 31)$	$(31, g_e \wedge t_7, 58)$	
63	$(0, g_0 \wedge t_6, 1)$	$(1, g_3 \wedge t_3, 5)$	$(5, g_7 \wedge t_1, 29)$	$(29, g_4 \wedge t_4, 53)$	$(53, g_{err} \wedge t_7, 63)$

where

$$
\begin{aligned}
g_{14} &= (\forall i \geq 0.(d + i < 9 \rightarrow (-21 \leq b + i - s \leq 19))) \rightarrow (-20 \leq b - s \vee b - s < 20) \\
g_{15} &= (\forall i \geq 0.(d + i \leq 9 \rightarrow (-21 \leq b + i - s \leq 19))) \rightarrow (-20 \leq b - s \vee b - s < 20)
\end{aligned}
$$

Figure 35: Error paths π used during refining

$$
\begin{array}{lcrcl}
a_0 & = & & & f_0 \\
a_1 & = & & & f_1 \\
a_2 & = & & & f_2 \\
a_3 & = & b < 9 + s & \rightarrow & r_1 \\
a_4 & = & s < b + 9 & \rightarrow & r_5 \\
a_5 & = & b + 1 < s & \rightarrow & r_1 \\
a_6 & = & d < 9 & \rightarrow & r_1 \\
a_7 & = & 1 + s < b & \rightarrow & r_5 \\
a_8 & = & d < 8 & \rightarrow & r_2 \\
a_9 & = & d < 9 & \rightarrow & r_1 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 8 \rightarrow r_4)) \\
a_{10} & = & & & r_0 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 9 \rightarrow r_3)) \\
a_{11} & = & 1 + s < b & \rightarrow & r_5 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 9 \rightarrow r_6)) \\
a_{12} & = & d = 9 & \rightarrow & r_1 \\
a_{13} & = & (s < b \wedge d = 9) & \rightarrow & r_0
\end{array}
$$

where

$$
\begin{array}{lcl}
r_5 & = & -19 \leq b - s \leq 21 \\
r_6 & = & -20 \leq b + i - s \leq 20
\end{array}
$$

Figure 36: Intermediate formulae a_i used in calculation of sequence interpolants where f_i for $i \in [0, 2]$ and r_j for $j \in [0, 4]$ are given in figure 21

4.3.10 Step 18: expand 32 and Step 19: refine 35

Vertex 32 is expanded and error vertex 35 is then refined, shown in figures 40 and 41 respectively. The refinement of 35 results in $\perp$ at 35 and f_1 at 32. However, the back propagation of f_1 at 32, through its transition formula:

$$g_5 \wedge t_5 = (d < 9) \wedge (b' = b + 1 \wedge d' = d + 1 \wedge s' = s)$$

results in the interpolant $d < 9 \rightarrow -21 \leq b - s \leq 19$. Since this does not entail f_1 which holds at vertex 5, the formula at 5 is strengthened to:

$$f_3 = f_1 \wedge (d < 9 \rightarrow -21 \leq b - s \leq 19)$$

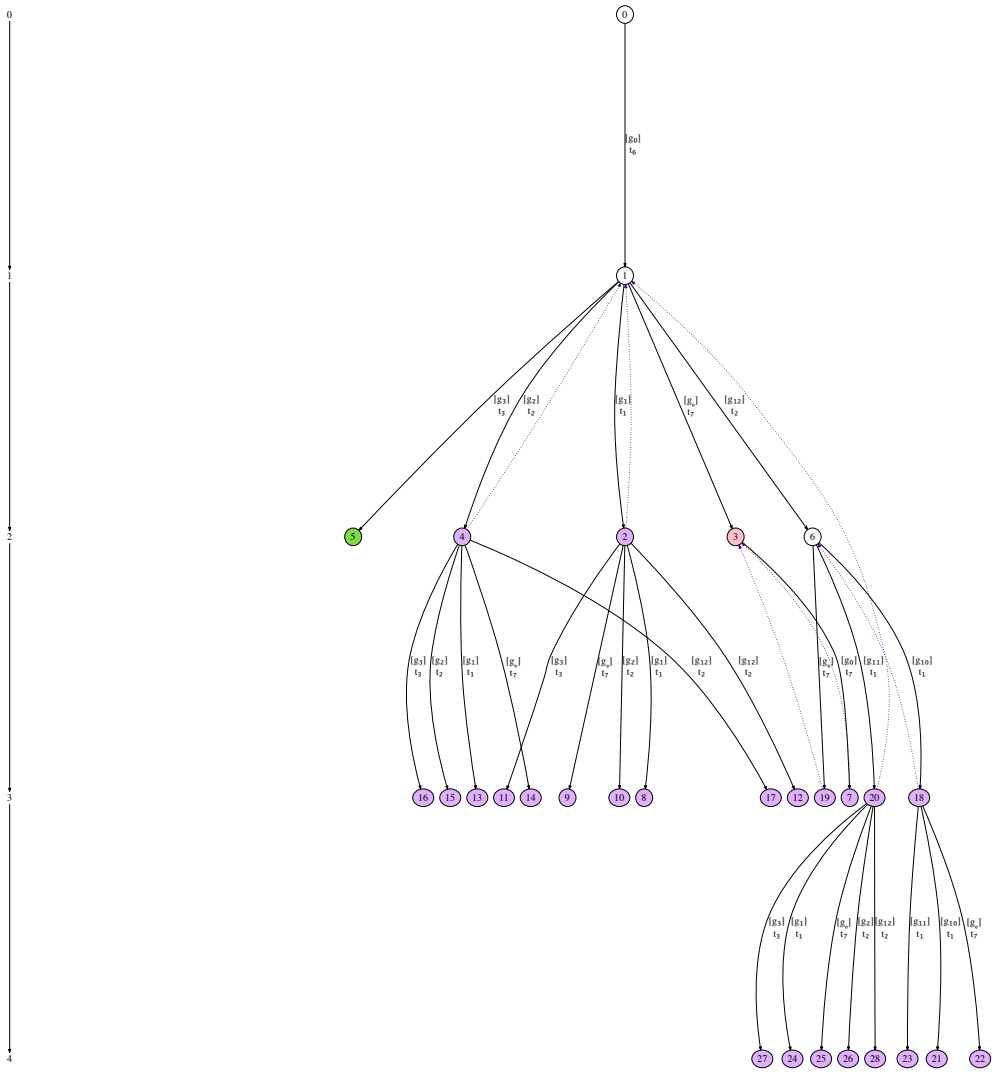


Figure 37: Step 15: Refinement of 25 (see also figure 79)

Thus, post strengthening, it can no longer cover vertex 32 where f_1 holds. Formula f_3 is larger (syntactically) than f_1 since it includes f_1 as a subformula. This marks the beginning of the growth problem at vertex 5 and its children.

4.3.11 Step 20: expand 33 and Step 21: refine 40

The growth problem is not immediately apparent because of a toing and froing action between two branches. Specifically, vertex 33 covers the children of 32.

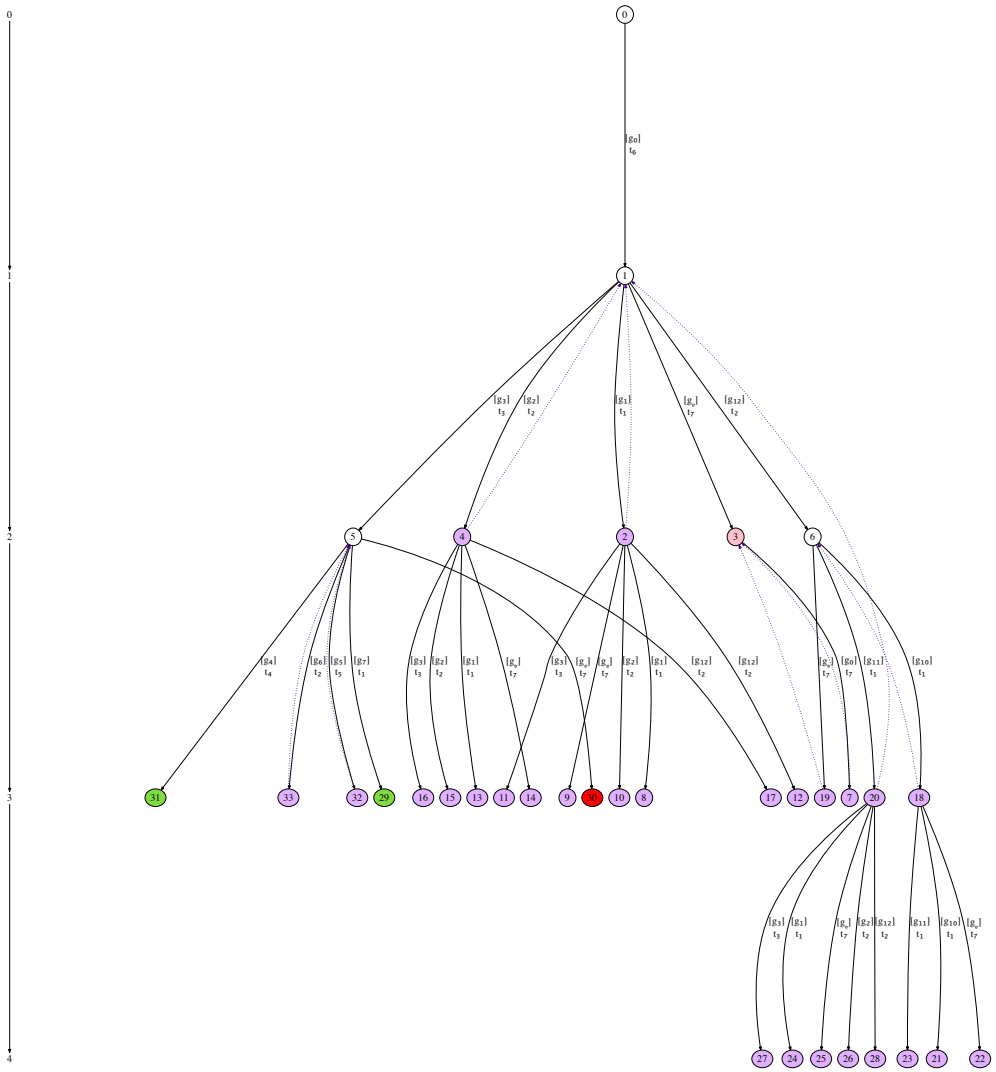


Figure 38: Step 16: Expansion of 5 (see also figure 80)

However, this covering is removed as soon as the error below 33 is refined. Toing and froing repeats as the two *brake* vertices, 32 and 33, are developed. (This behaviour is not core to the growth problem but an artefact of the concrete example under analysis.)

Following the refinement of vertex 35, the unprocessed children of 32 are now covered by the unprocessed children of its parent, vertex 5. Thus, the other *brake* vertex, 33, is expanded and its error vertex, 40, refined. Expansion results in figure 42 and refinement in figure 43. (Vertex 33 is no longer covered by 32 following

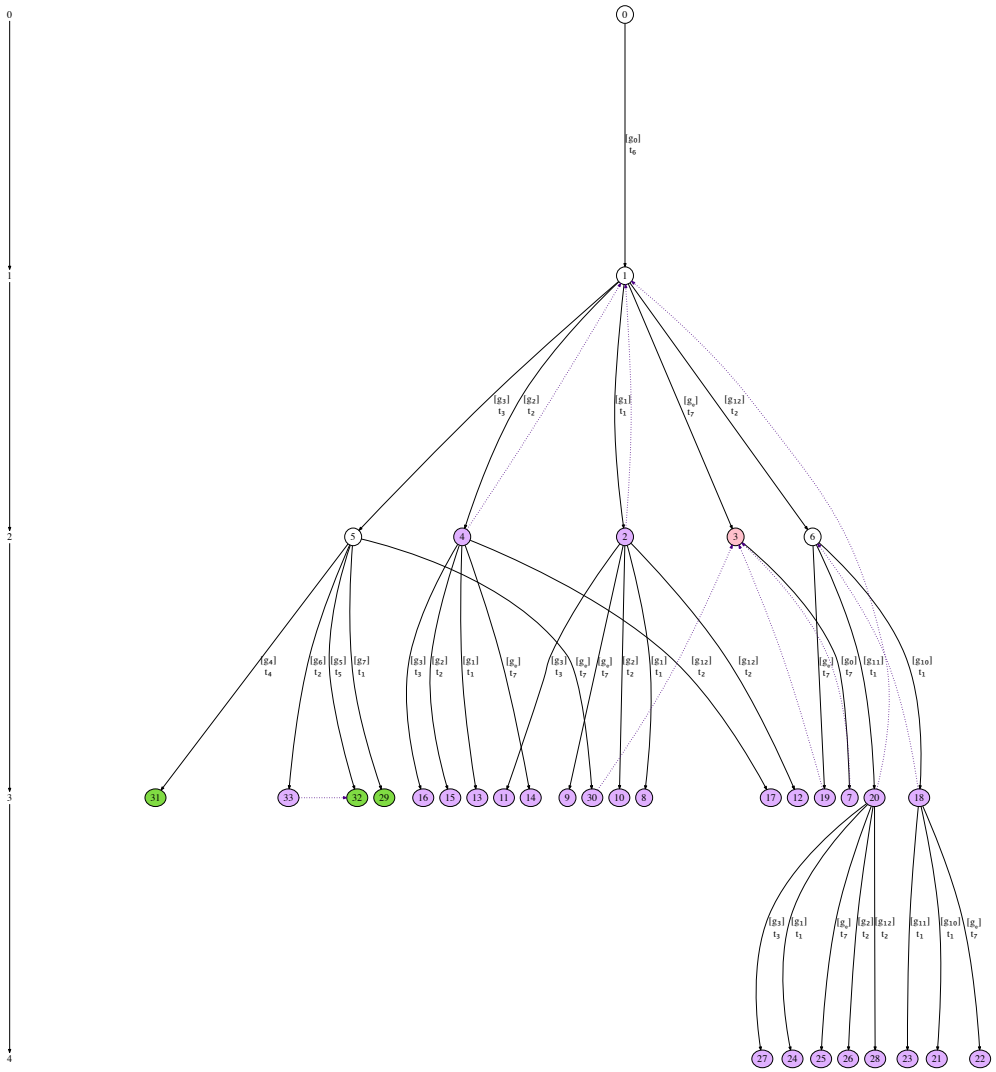


Figure 39: Step 17: Refinement of 30 (see also figure 81)

its refinement). When vertex 40 is refined, the sequence interpolant is the same as in the previous expansion-refinement steps, save for the interpolant calculated at vertex 5 which is:

$$a_7 = (1 + s < b) \rightarrow (-19 \leq b - s \leq 21)$$

This is not stronger than f_3 so f_3 persists at vertex 5. However, the strengthening of the formula to f_1 at 33 results in the uncovering of the children of 32, including

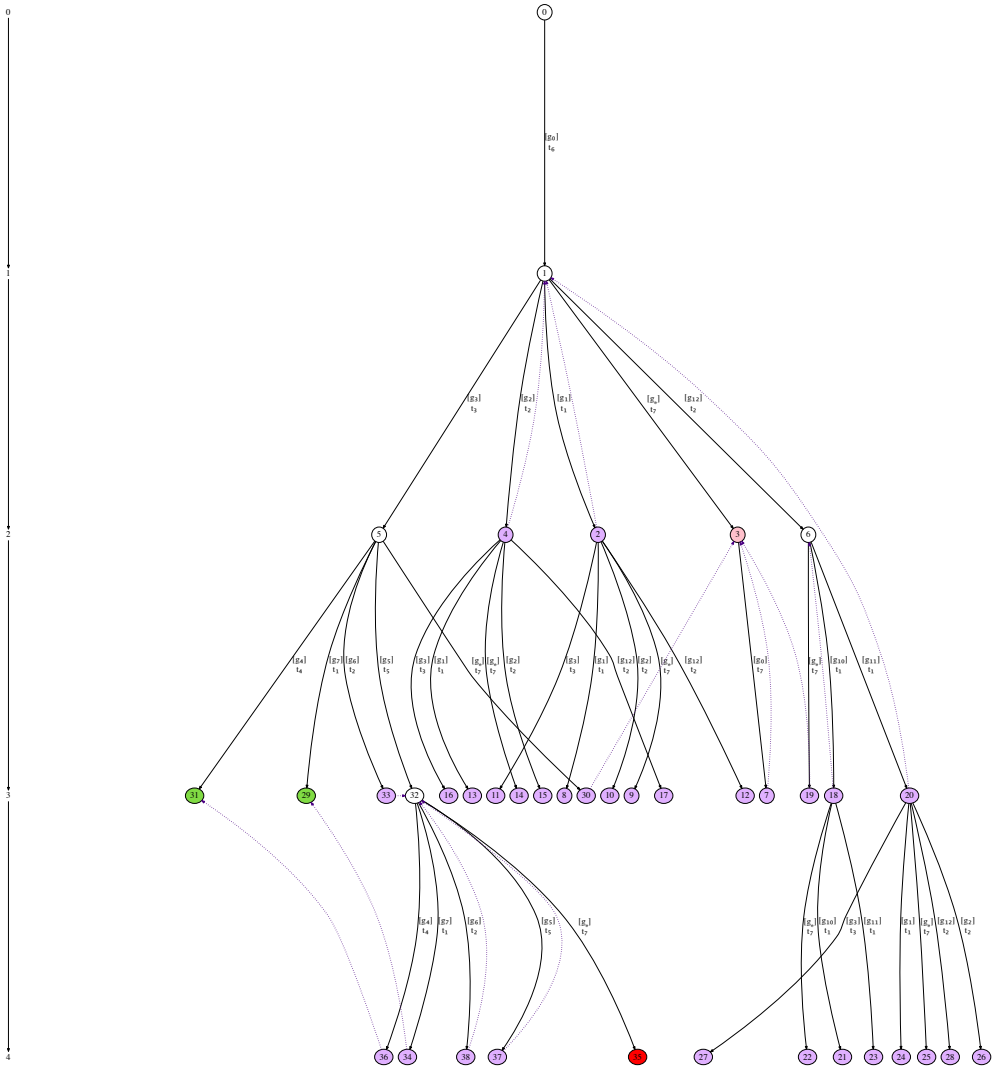


Figure 40: Step 18: Expansion of 32 (see also figure 82)

vertex 37, opening this branch for development.

4.3.12 Step 22: expand 37 and Step 23: refine 45

Once vertex 37 is expanded (figure 44) and error vertex 45 refined (figure 45), the formula at 37 becomes $f_1 = r_0$, the formula at 32 becomes:

$$f_3 = f_1 \wedge (d < 9 \rightarrow r_1)$$

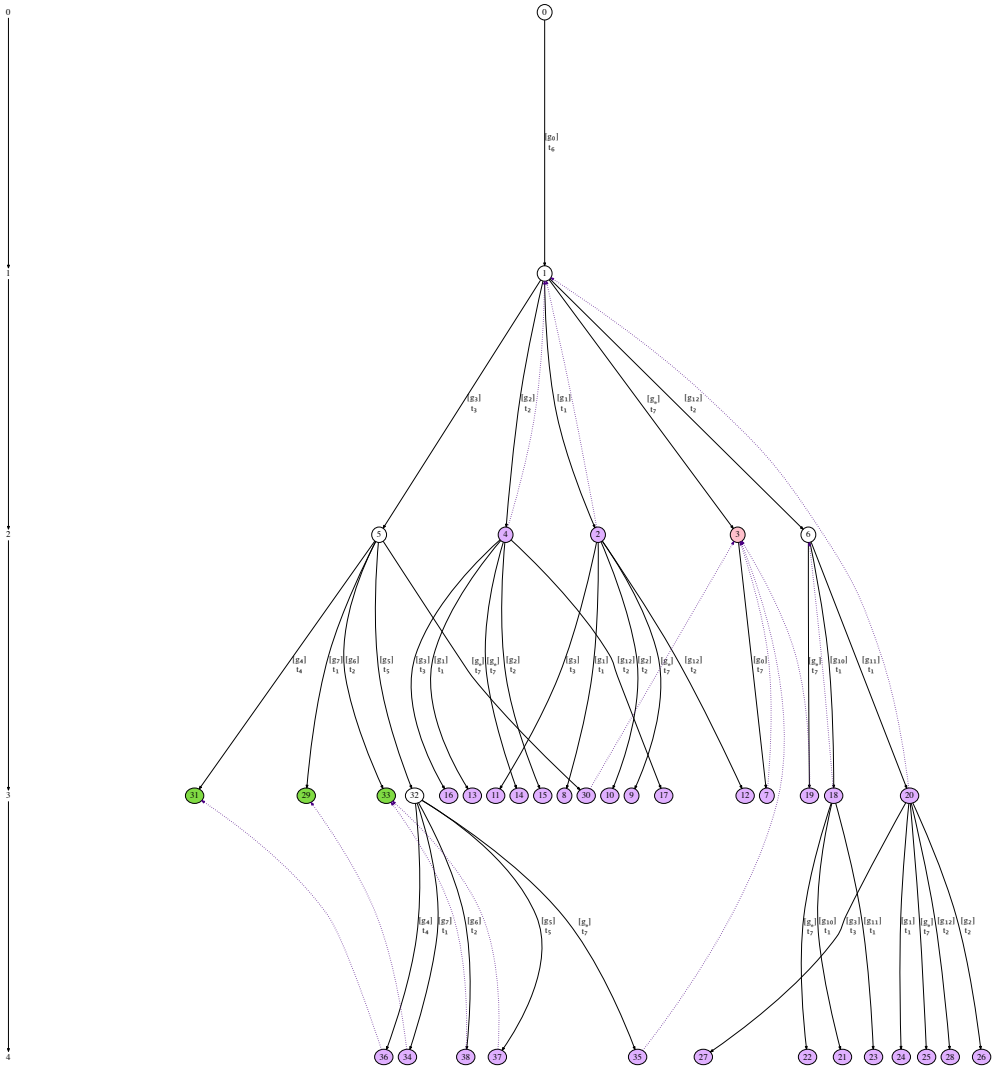


Figure 41: Step 19: Refinement of 35 (see also figure 83)

and the formula at 5 becomes:

$$\begin{aligned} f_4 &= f_3 \wedge (d < 8 \rightarrow r_2) \\ &= f_1 \wedge (d < 9 \rightarrow r_1) \wedge (d < 8 \rightarrow r_2) \\ &= f_1 \wedge (d < 9 \rightarrow -21 \leq b - s \leq 19) \wedge (d < 8 \rightarrow -22 \leq b - s \leq 18) \end{aligned}$$

Observe the most recently added conjunct at 37 is $r_0 = -20 \leq b - s \leq 20$, which, when propagated back through the transition $g_5 \wedge t_5$, generates the conjunct $d < 9 \rightarrow r_1 = d < 9 \rightarrow (-21 \leq b - s \leq 19)$. This conjunct is then added to the

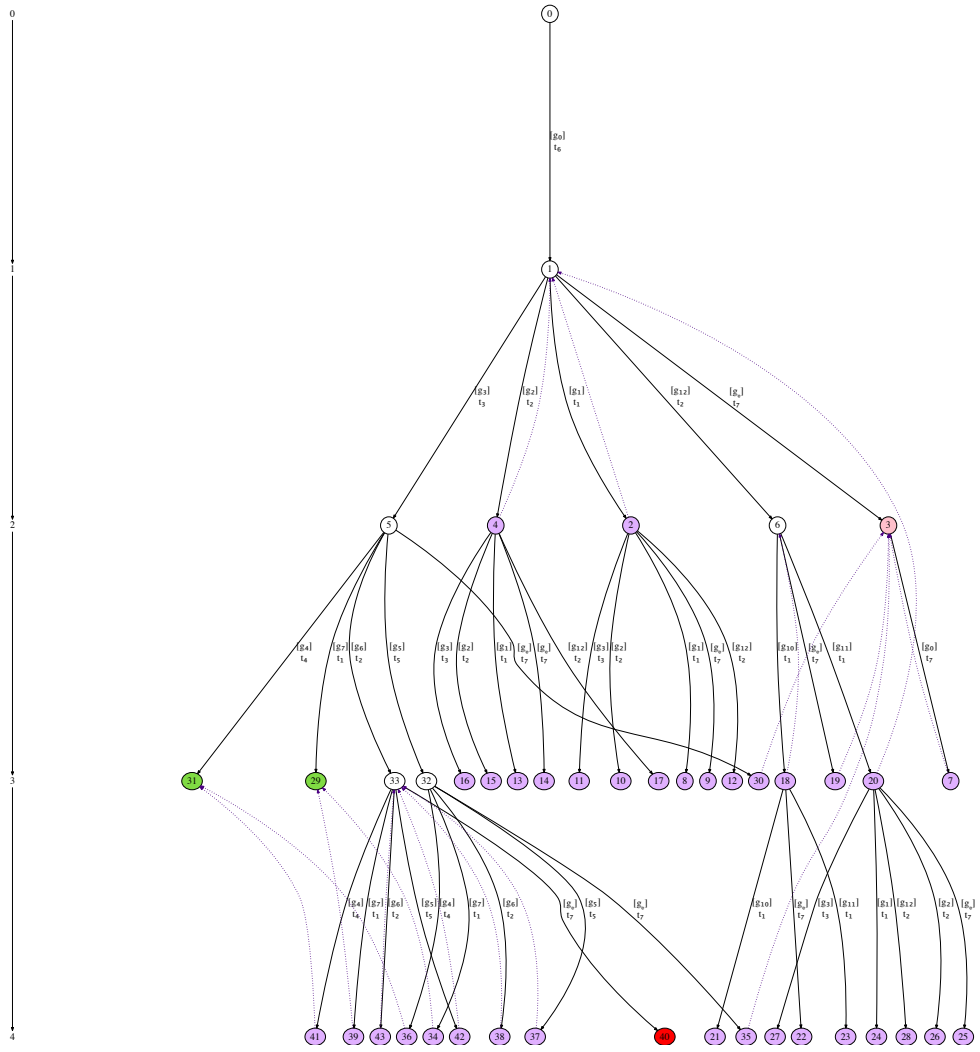


Figure 42: Step 20: Expansion of 33 (see also figure 84)

formula at 32 since the formula currently at 32 does not entail it. And when this additional conjunct is propagated back, the formula at 5 is augmented with $d < 8 \rightarrow r_2 = d < 8 \rightarrow (-22 \leq b - s \leq 18)$. This pattern would continue should the branch be further developed.

For covering to occur, it is necessary for a child vertex of a brake vertex to be itself a brake vertex where its formula is stronger than that of its parent (or a predecessor elsewhere in the unwinding graph). However, whenever a condition is added which prevents the error location being reached, once propagated back

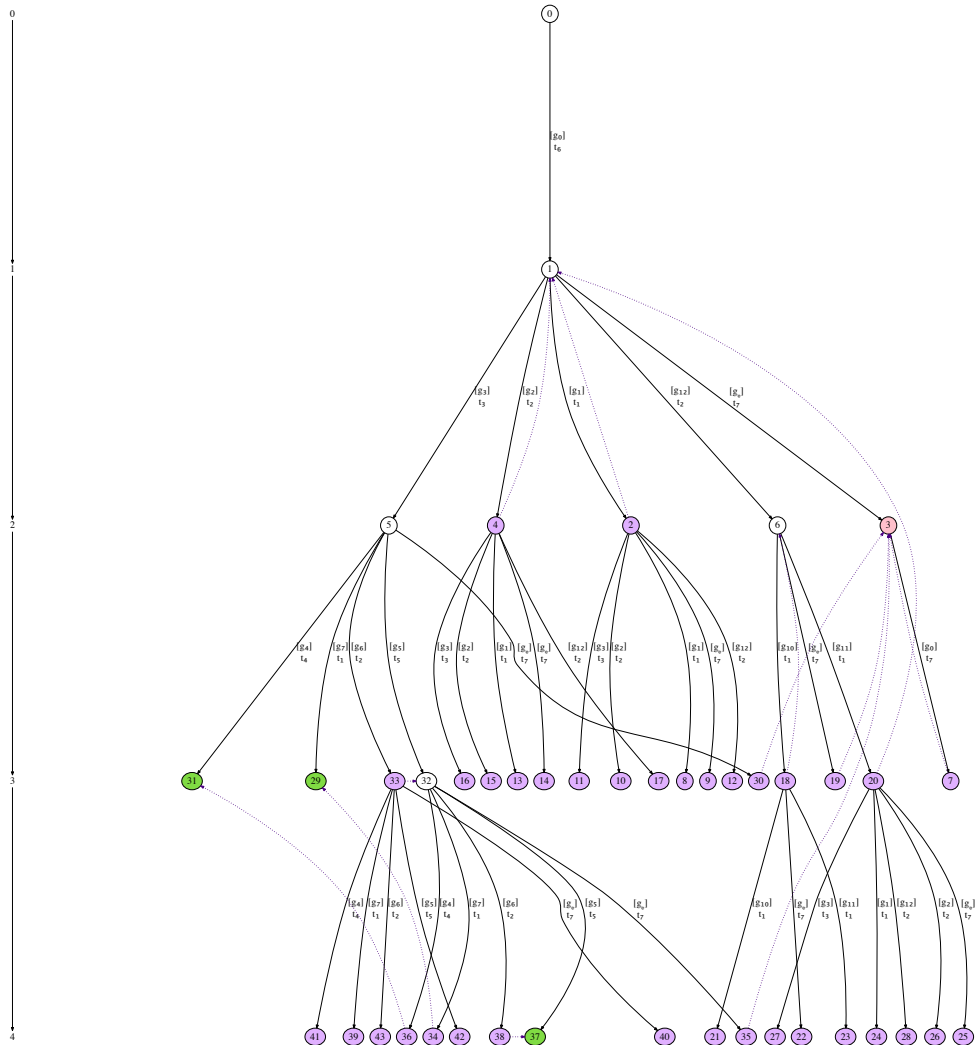


Figure 43: Step 21: Refinement of 40 (see also figure 85)

across this particular transition $g_5 \wedge t_5$, it generates a new conjunct at the parent. This process would continue ad infinitum and is a fundamental problem with IMPACT.

To address this problem, observe that the subformulae $d < 9 \rightarrow r_1$ and $d < 8 \rightarrow r_2$ can be written as:

$$(d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i)$$

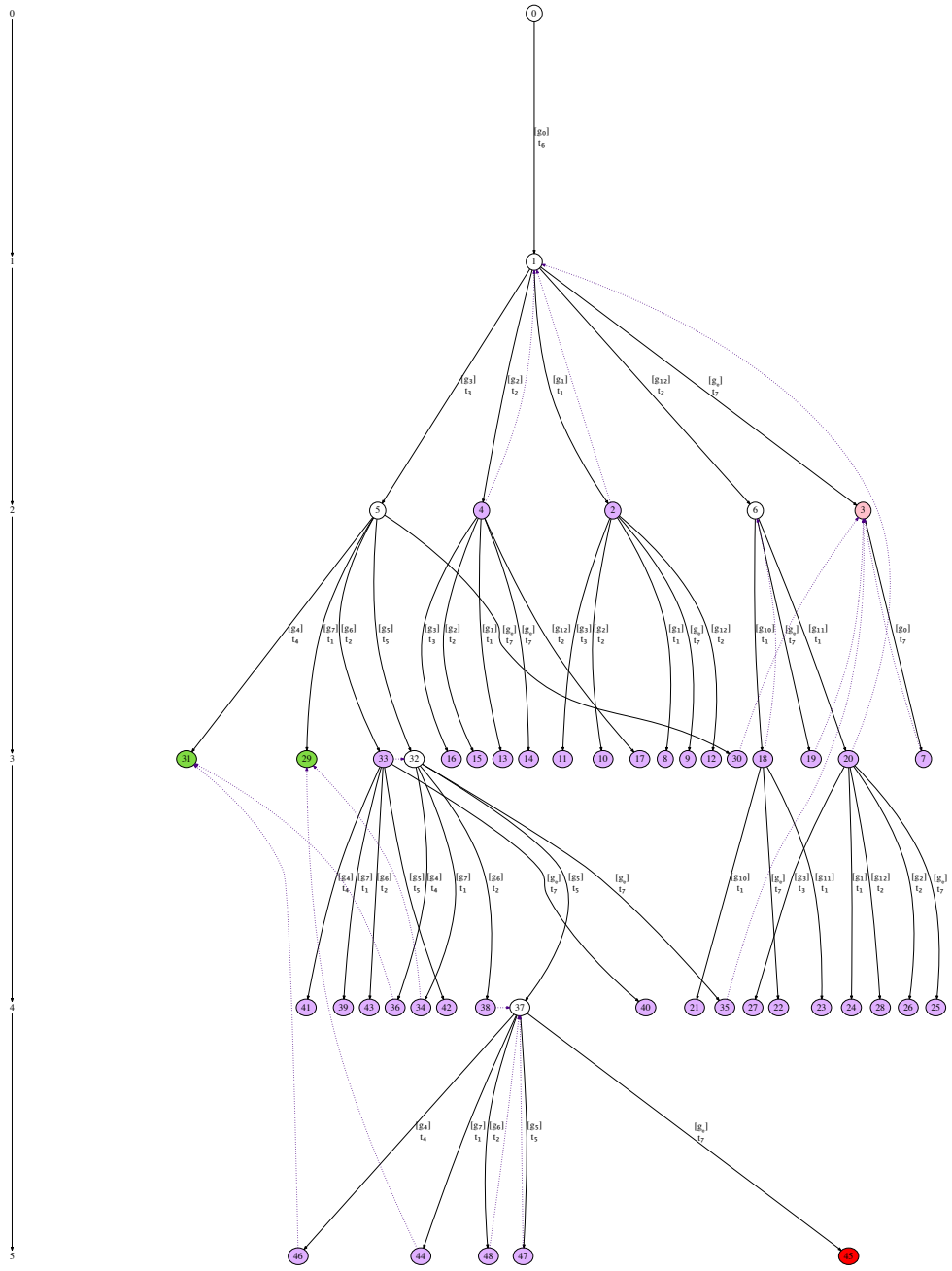


Figure 44: Step 22: Expansion of 37 (see also figure 86)

where $i = 0$ and $i = 1$ respectively. Therefore f_4 can be represented more compactly using quantification:

$$f_4 = f_1 \wedge \forall i \in [0, 1].((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$

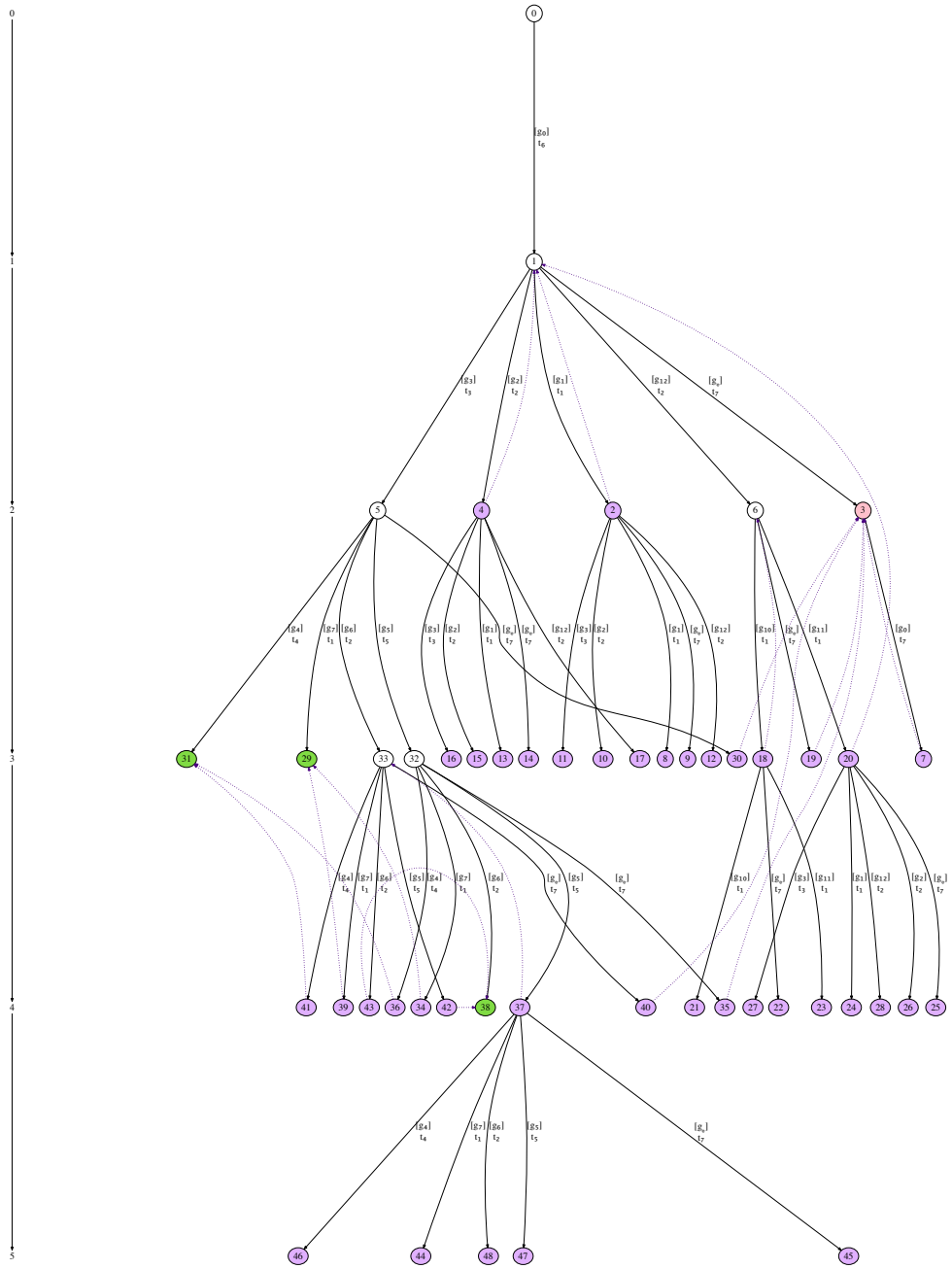


Figure 45: Step 23: Refinement of 45 (see also figure 87)

This suggests that the general form of the formula that holds at vertex 5 is:

$$f_1 \wedge \forall i \in [0, u].((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$

where u is an ever-increasing upper bound.

The above discussion focuses on vertex 5 and the problem of its formula becoming repeatedly strengthened by adding conjuncts. This phenomenon is not unique to vertex 5, but also occurs at vertices 32 and 37. The general form of the formulae that hold at vertex 32 and 37 is respectively:

$$f_1 \wedge \forall i \in [0, u-1].((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$
$$f_1 \wedge \forall i \in [0, u-2].((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$

To curtail this growth, and to induce termination, the formula at 32 can be strengthened in such a way that covering can be applied, and will continue to apply, during development of the unwinding graph. This is achieved by relaxing the upper bound of the interval to infinity which gives a formula that is stable in the sense that its upper bound cannot be further extended. For vertex 32, this amounts to strengthening its formula to:

$$g_1 = f_1 \wedge \forall i \geq 0.((d < 9 - i) \rightarrow (-21 - i \leq b - s \leq 19 - i))$$

as given in the list of operations in figure 33 as a set operation. This is akin to widening in abstract interpretation, but instead of relaxing the abstraction, the formula is restricted by enlarging bounds. This strengthening and subsequent backwards propagation of the new formula at 32 constitutes the next two steps.

4.3.13 Step 24: set 32 g_1 and Step 25: refine 49

Since g_1 persists in the unwinding graph, it is assigned the name f_5. Having strengthened vertex 5 (figure 46), it is necessary to propagate f_5 up the branch, in the same way as when a vertex is strengthened to rule out an error. To do so, an error location 49 is added whose transition formula is $\neg g_1 \wedge t_7$ (where $g_1 = f_5$). When error 49 is refined (figure 47), the formula f_5 at 32 is not strengthened before it is propagated backwards. Once f_5 has been propagated through the transition $g_5 \wedge t_5$, the formula at 5 is updated to:

$$h = f_0 \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 0 \rightarrow (d < 8 - i \rightarrow r_4))))$$

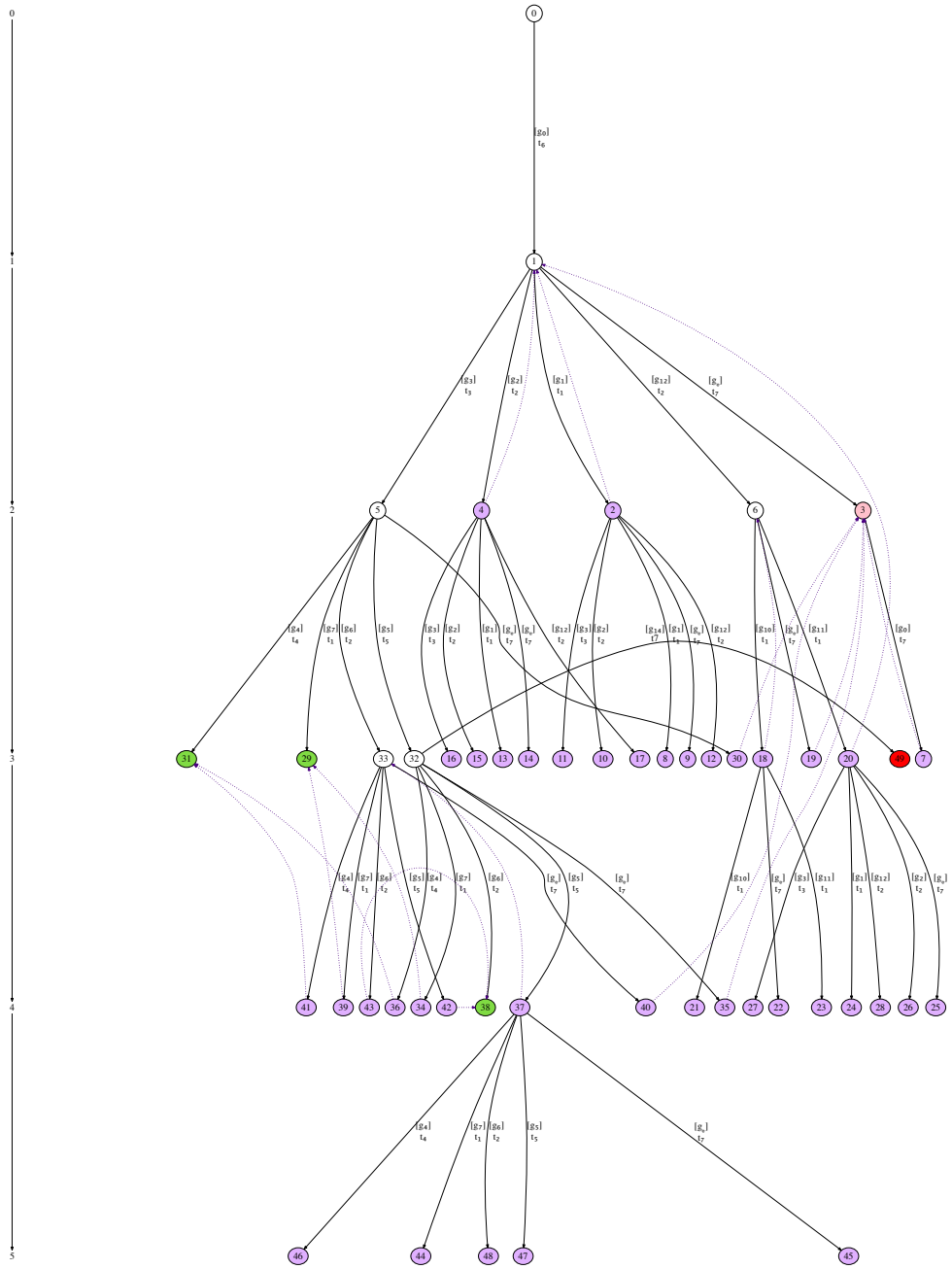


Figure 46: Step 24: Set g_1 at 32 (see also figure 88)

Although not obvious, $f_5 \models h$. The force of this entailment is that it is sufficient for covering, hence termination. Prior to this intervention, the formula at vertex 5 entailed that at 32 ($f_4 \models f_3$) but now the formula at 32 entails that at 5 ($f_5 \models h$).

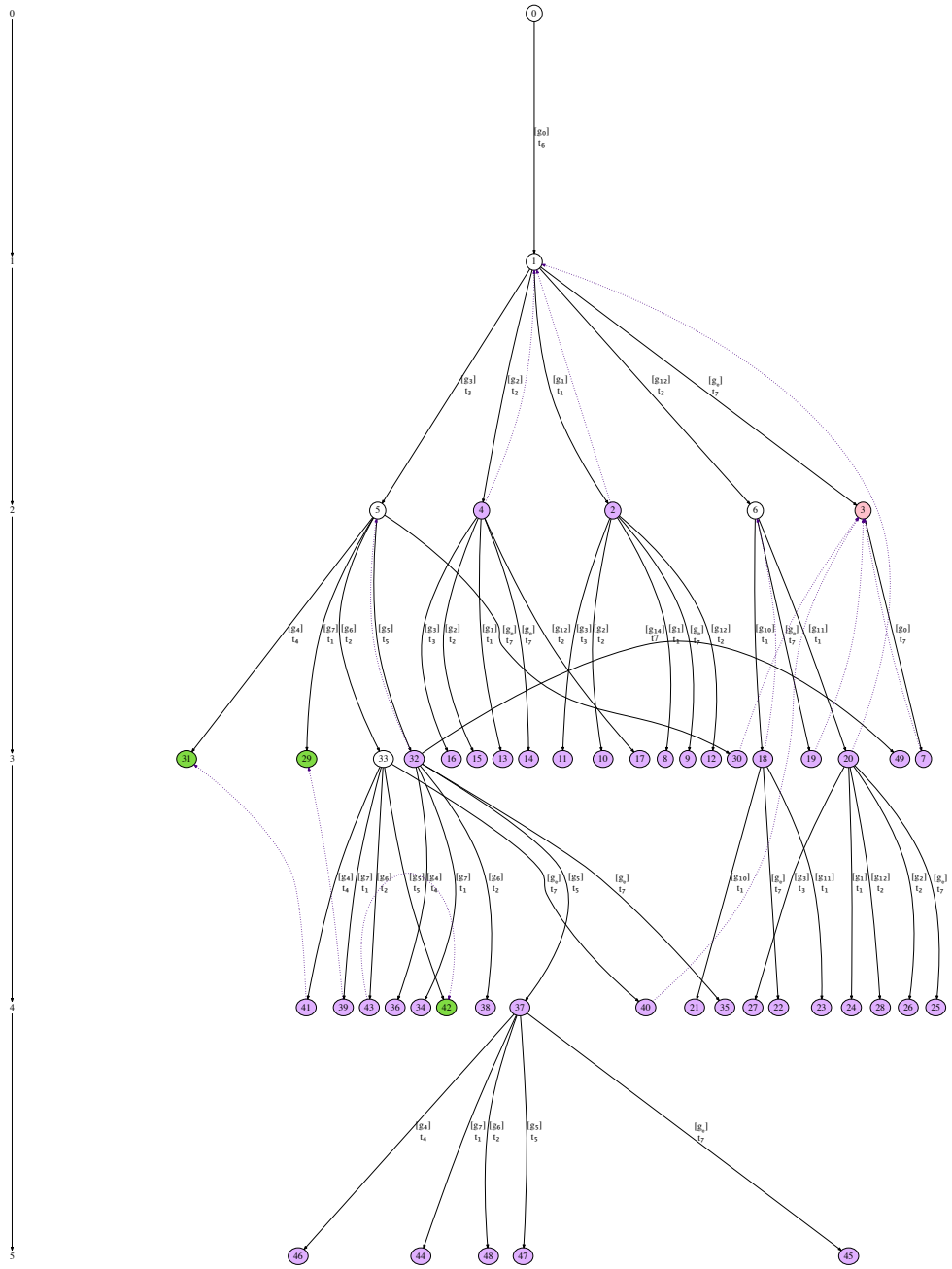


Figure 47: Step 25: Refinement of 49 (see also figure 89)

To see that this entailment holds, observe that h can be rewritten as:

$$h' = f_0 \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 1 \rightarrow (d < 9 - i \rightarrow r_3))))$$

where r_3 is given in figure 21. Next, $d < 9$ is distributed over the implication of h' to give the following:

$$\begin{aligned} h'' &= f_0 \wedge (d < 9 \rightarrow r_1) \wedge (d < 9 \rightarrow (\forall i.i \geq 1 \rightarrow (d < 9 - i \rightarrow r_3))) \\ &= f_0 \wedge (d < 9 \rightarrow r_1) \wedge (\forall i.i \geq 1 \rightarrow (d < 9 \rightarrow (d < 9 - i \rightarrow r_3))) \\ &= f_0 \wedge (d < 9 \rightarrow r_1) \wedge (\forall i.i \geq 1 \rightarrow (d < 9 - i \rightarrow (d < 9 \rightarrow r_3))) \end{aligned}$$

Since $d < 9 - i \models d < 9$ for all $i \geq 1$, it follows that h'' can be simplified to:

$$h''' = f_0 \wedge (d < 9 \rightarrow r_3) \wedge (\forall i.i \geq 1 \rightarrow (d < 9 - i \rightarrow r_3))$$

Furthermore, h''' can be expressed as:

$$h'''' = f_0 \wedge (\forall i.i \geq 0 \rightarrow (d < 9 - i \rightarrow r_3)) = r_0 \wedge (\forall i.i \geq 0 \rightarrow (d + i < 9 \rightarrow r_3))$$

which is syntactically equal to f_5, in particular $f_5 \models h''''$, as required. A covering arc can then be inserted from vertex 32 to vertex 5, closing off the branch.

4.3.14 Step 26: set 33 g_2 and Step 27: refine 26

Vertex 33 is another child of vertex 5 and like vertex 32, corresponds to the location *brake*. However, it can be strengthened using the same technique, and in fact, even to the same formula, hence a commentary is not provided on its development. These two steps result in figure 48 and then figure 49. After these two steps, the branch ending with vertex 33 is also closed.

4.3.15 Step 28: expand 29 and Step 29: refine 52

There are two child leaf vertices of vertex 5: vertex 29 and vertex 31. Vertex 29, which represents automaton location *stop*, is arbitrarily chosen for development. Once vertex 29 is expanded (figure 50) and error vertex 52 refined (figure 51), the formula at 29 is strengthened to $f_1 = r_0$. However, once this is propagated through the transition $g_7 \wedge t_1 = (d = 9) \wedge (b' = b + 1 \wedge d' = d \wedge s' = s)$, the conjunct $d = 9 \rightarrow r_1$ is added to the formula at vertex 5, resulting in its formula

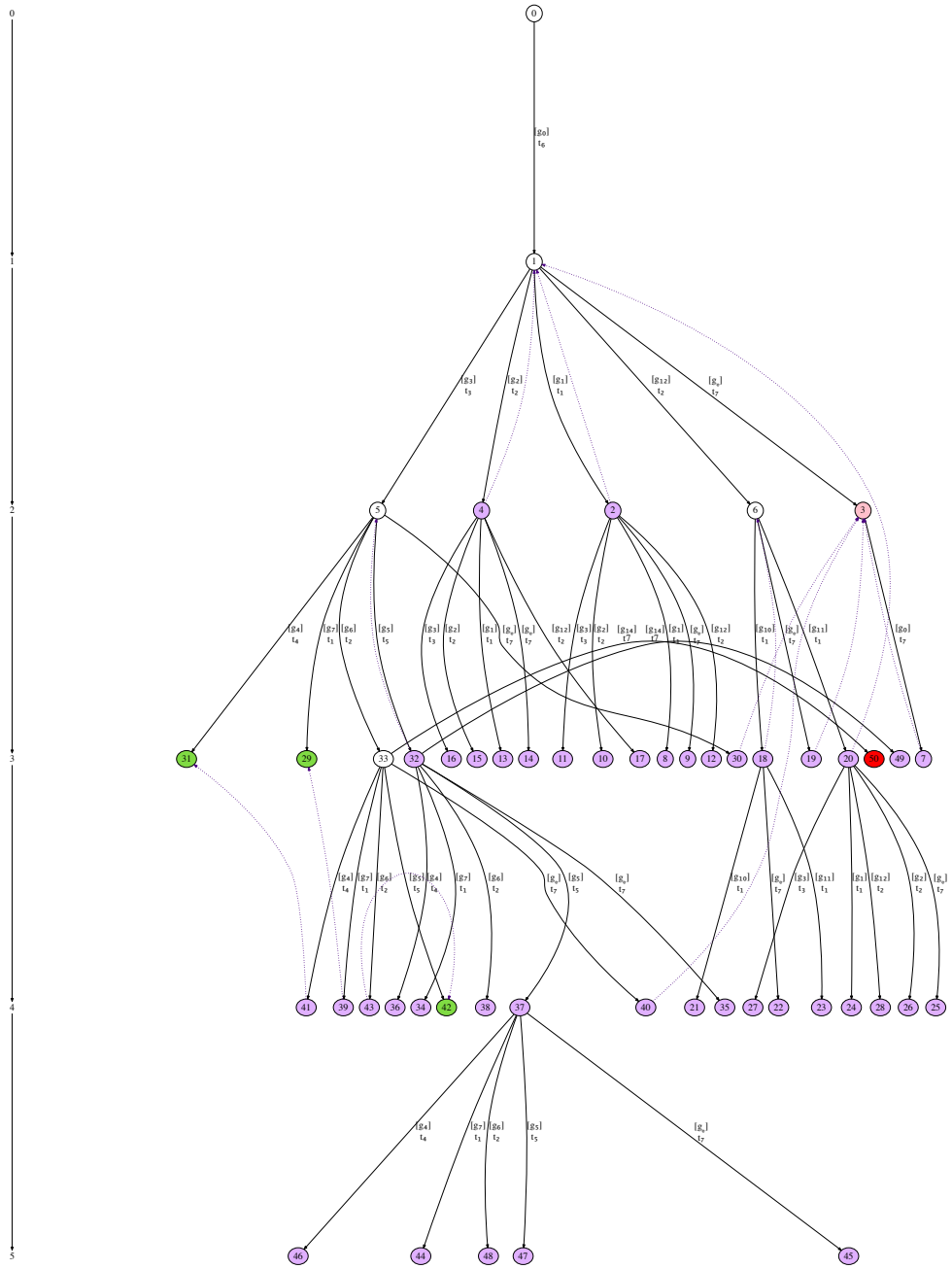


Figure 48: Step 26: Set g_1 at 33 (see also figure 90)

strengthening from f_6 to f_7. This strengthening invalidates the covering arcs in place between 32 and 5, and between 33 and 5, and these are removed.

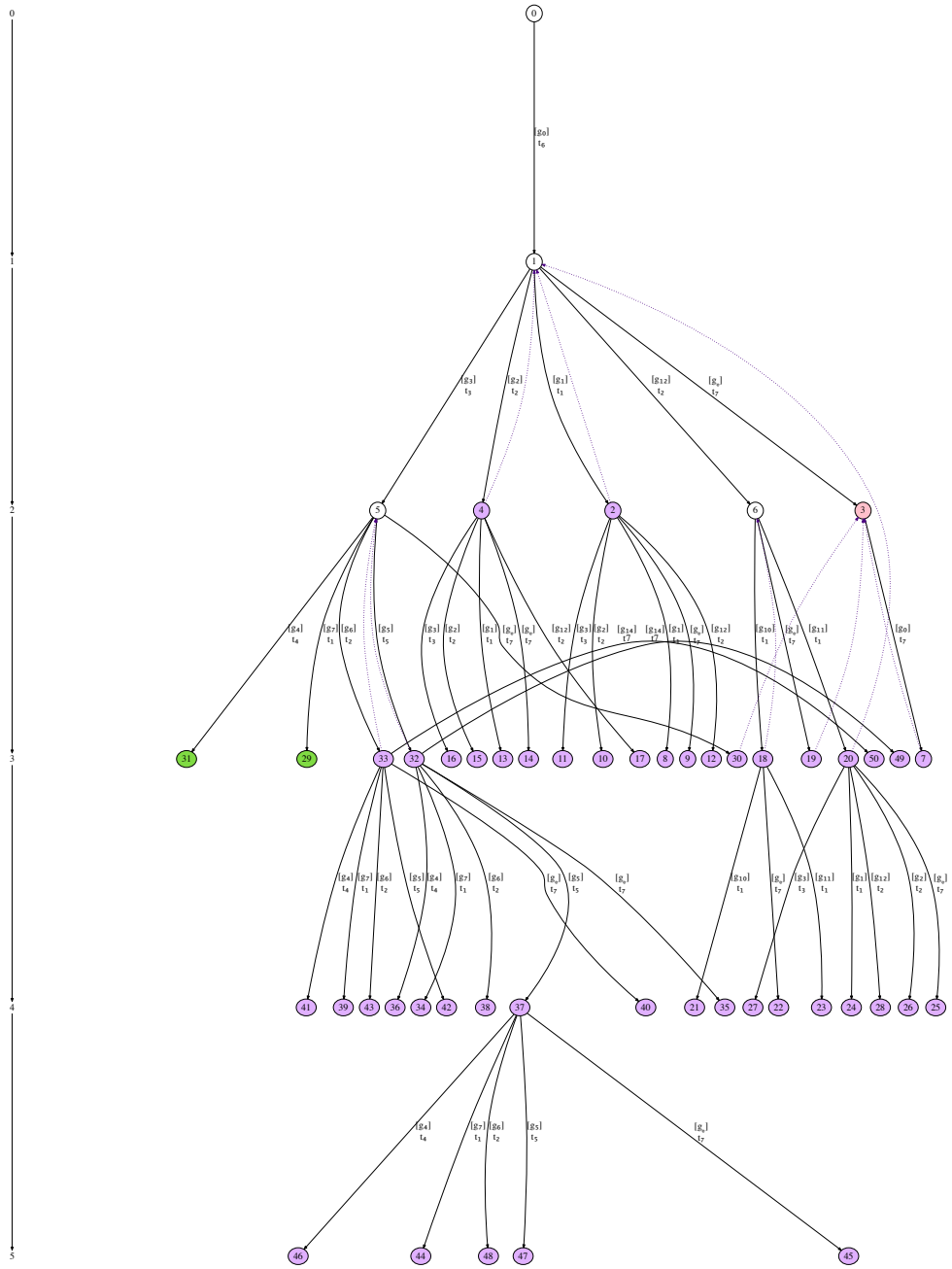


Figure 49: Step 27: Refinement of 50 (see also figure 91)

4.3.16 Step 30: set 32 g_2 and Step 31: refine 49

Now that the vertices at 32 and 33 are no longer covered by vertex 5, the formula at 32, f_5, is further strengthened to g_2 in order to reclose its branch. Strengthening

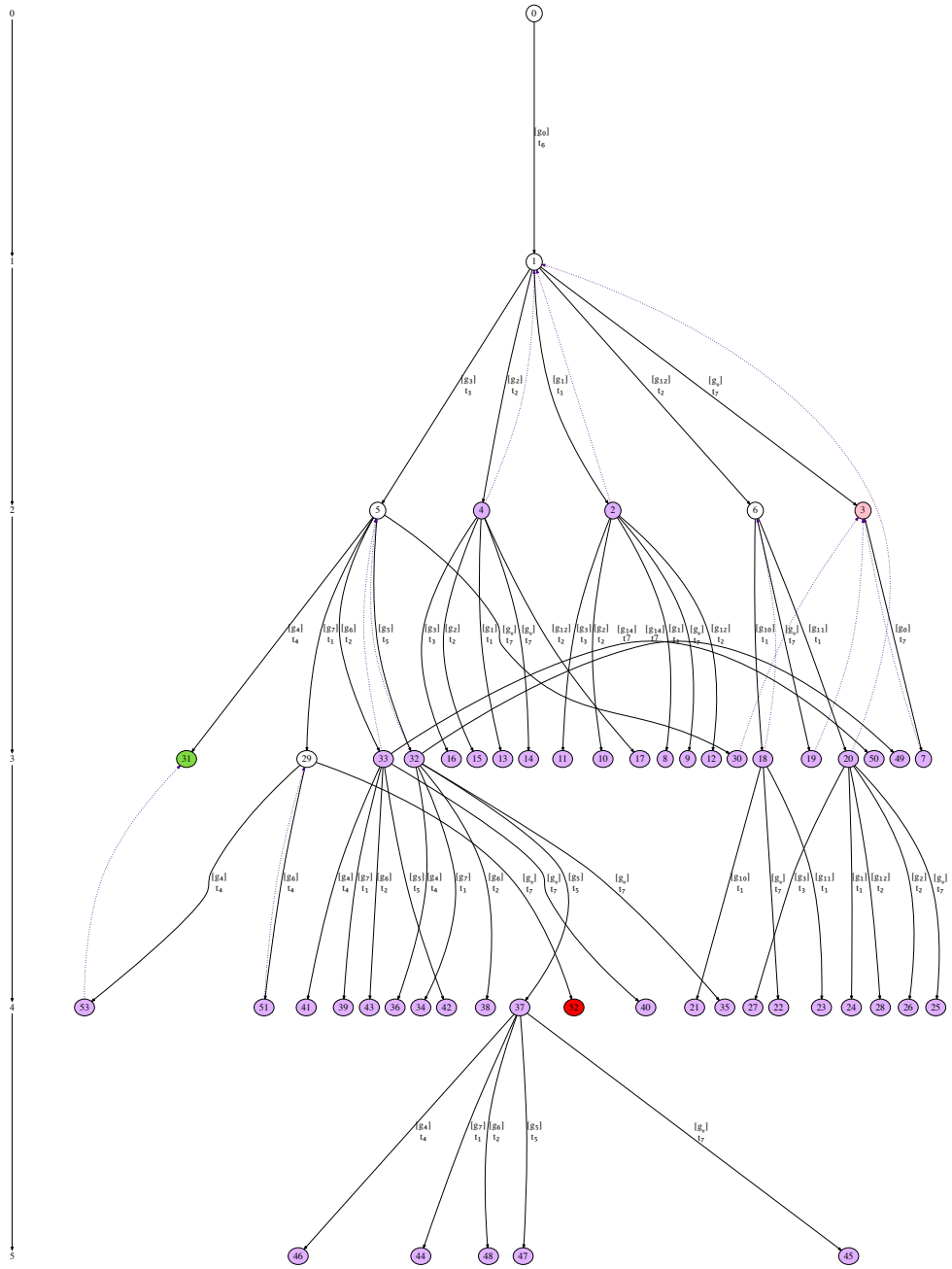


Figure 50: Step 28: Expansion of 29 (see also figure 92)

gives figure 52. Since g_2 will persist in the unwinding graph, it is assigned the name f_8. When f_8 is propagated backwards, through the refinement of error vertex 49

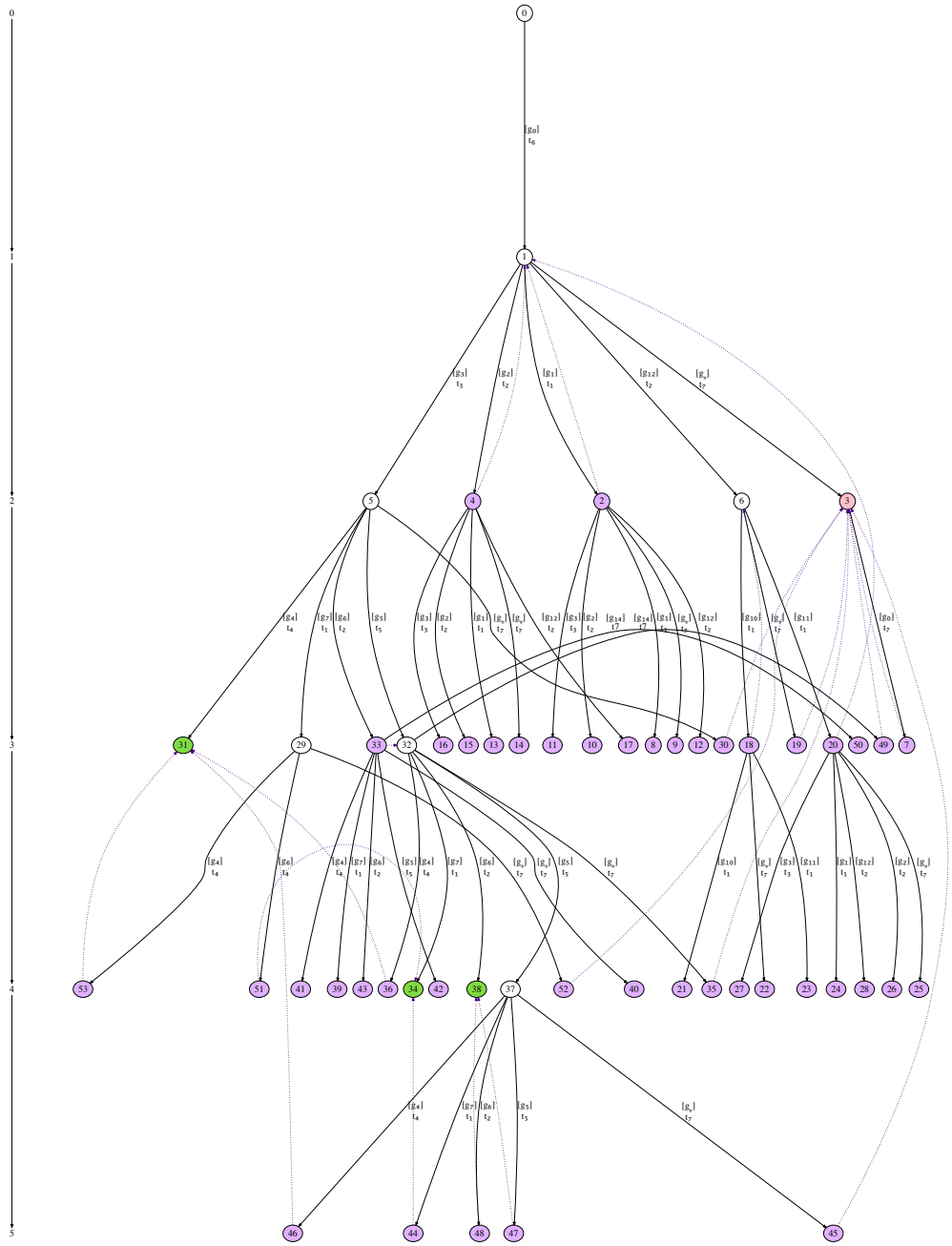


Figure 51: Step 29: Refinement of 52 (see also figure 93)

(figure 53), the formula at 5 is strengthened to:

$$f_9 = r_0 \wedge (d = 9 \rightarrow r_1) \wedge (d < 9 \rightarrow (r_1 \wedge (\forall i.i \geq 0 \rightarrow ((8 < d + i) \vee r_4))))$$

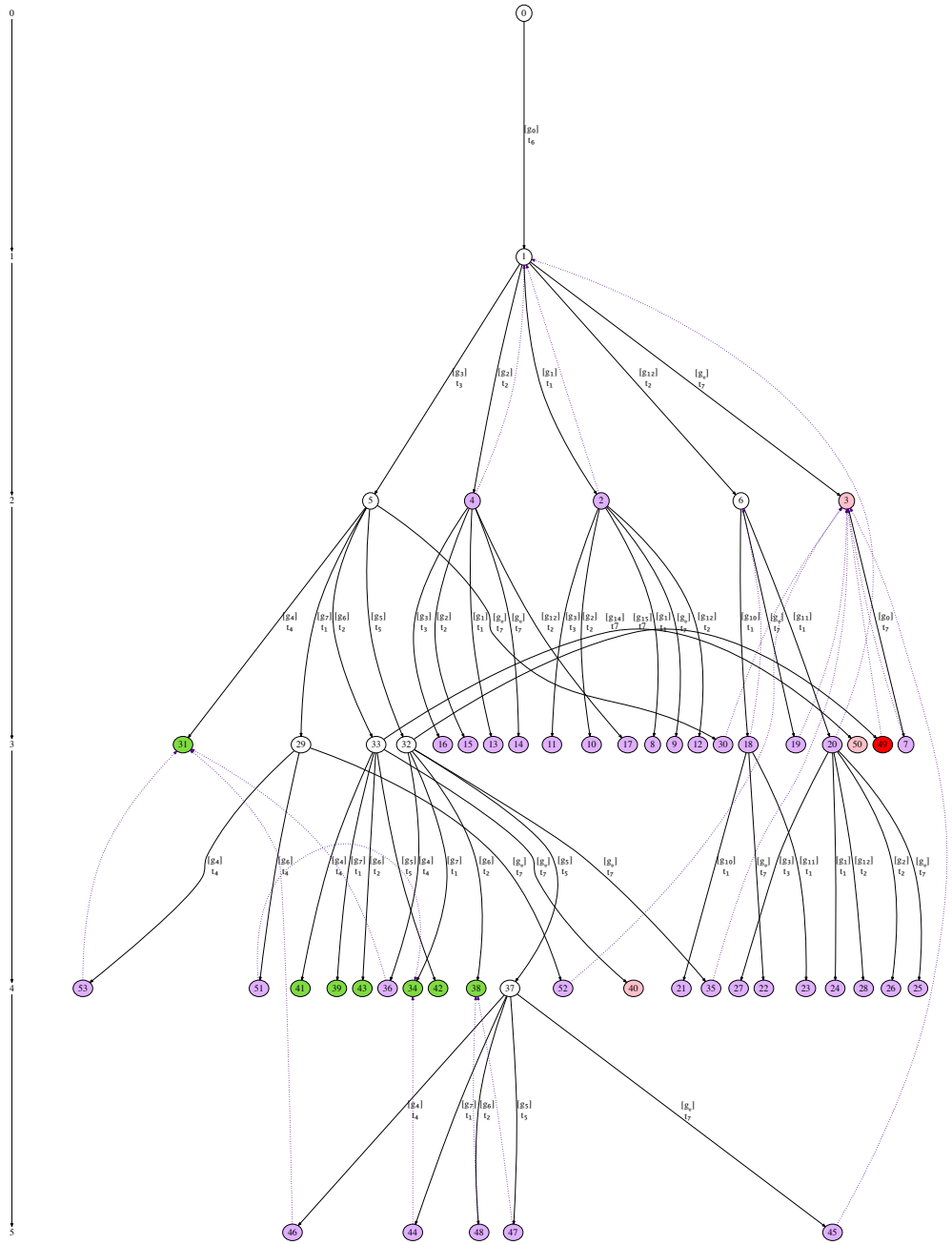


Figure 52: Step 30: Set g_2 at 30 (see also figure 94)

As a result, $f_8 \models f_9$ and (32, 5) is reinstated. To show entailment, a number of formula rewrites can be performed on f_9. First, $d < 9$ is distributed over the implication. Second, the disjunct $((8 < d + i) \vee r_4)$ is expressed as implication.

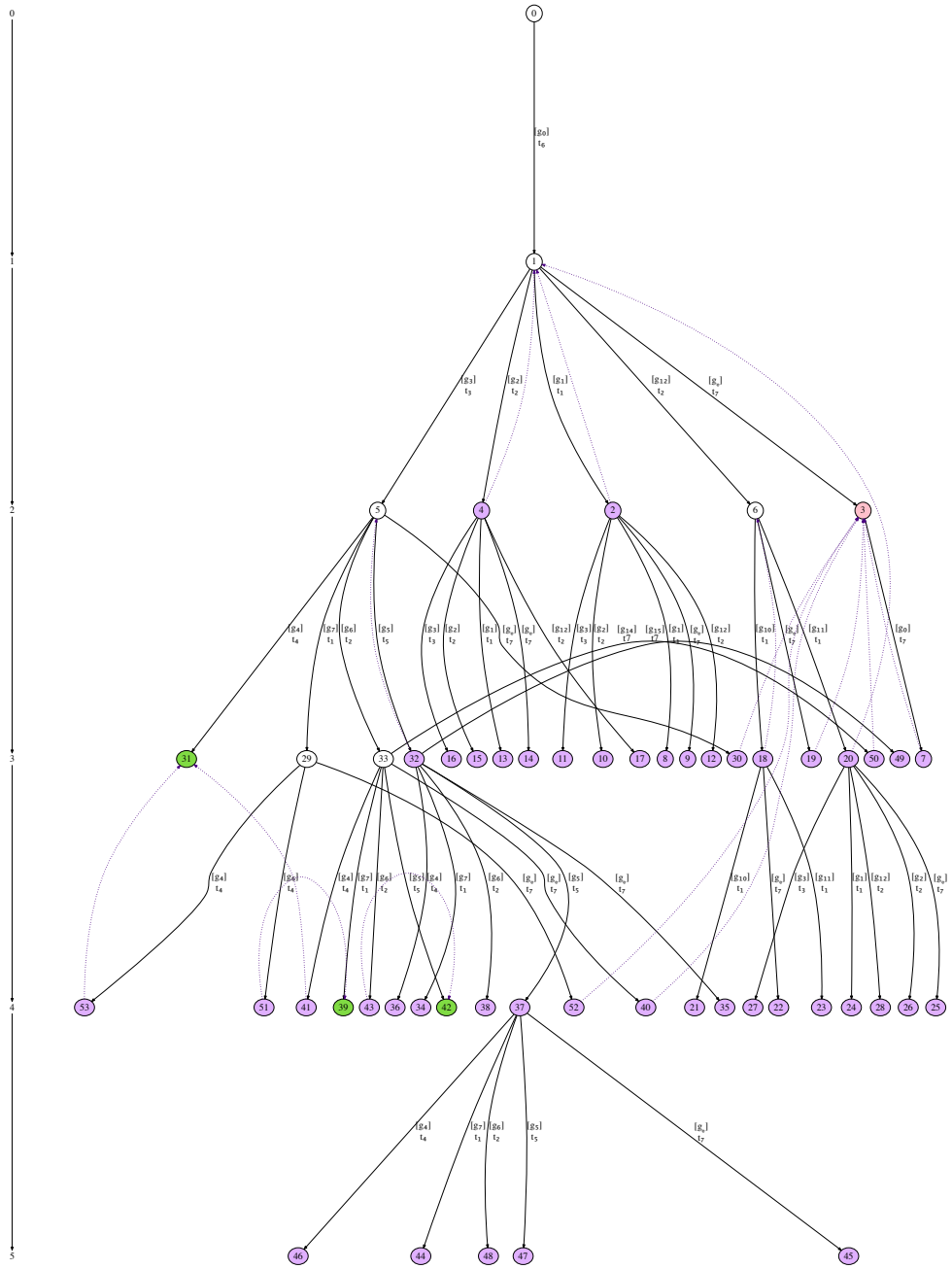


Figure 53: Step 31: Refinement of 49 (see also figure 95)

Third, the range of the quantified variable is adjusted to start at 1. Fourth and fifth, the premises of the nested implications are reordered. Sixth, the $d < 9$ premise is dropped since $d + i \leq 9 \models d < 9$ for all $i \geq 1$. Seventh, the $d = 9 \rightarrow r_1$

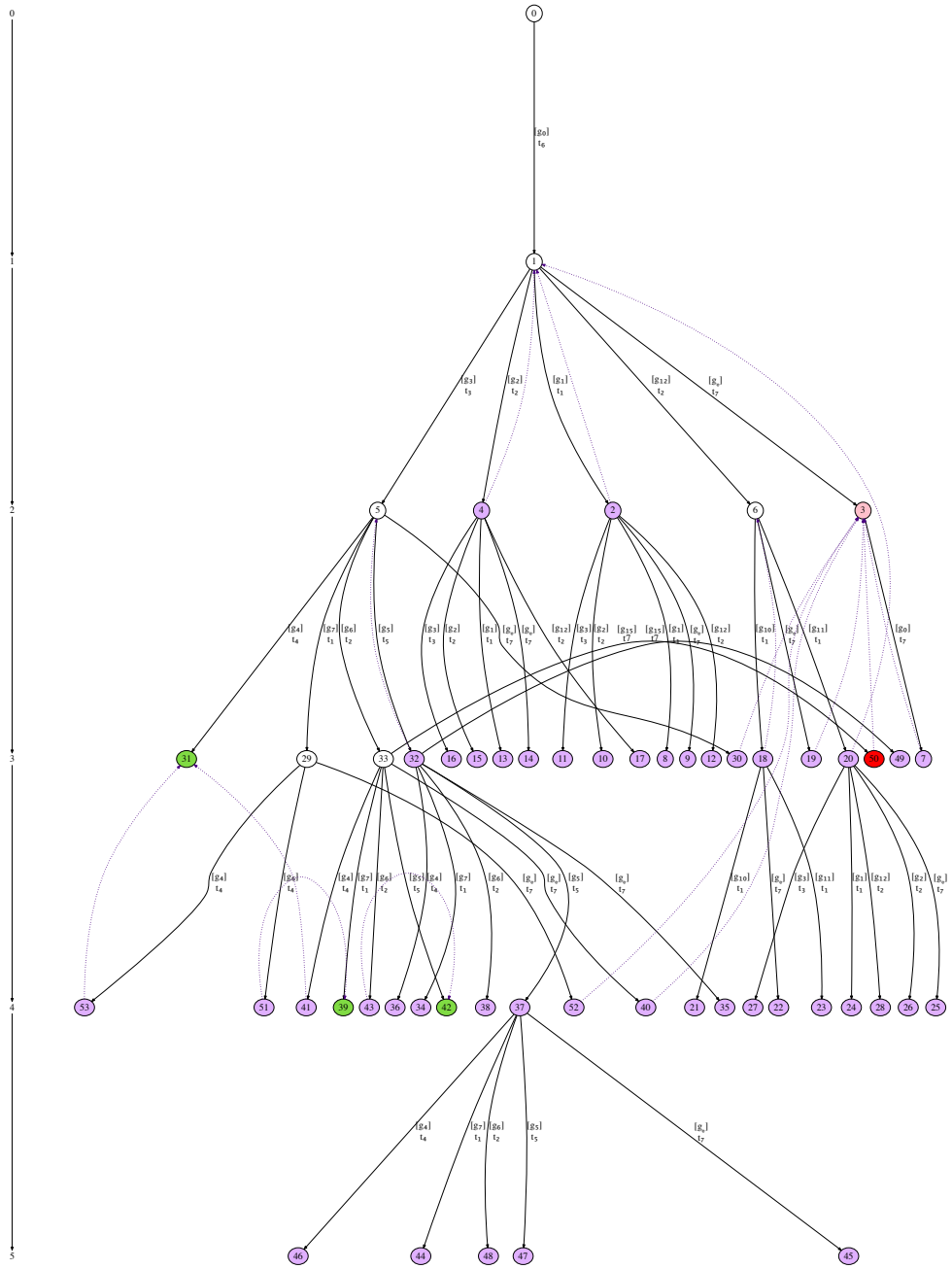


Figure 54: Step 32: Set g_2 at 32 (see also figure 96)

and $d < 9 \to r_1$ conjuncts are expanded to $d + 0 = 9 \to r_1$ and $d + 0 < 9 \to r_1$. Eighth, the two conjuncts are reduced to $d + 0 \leq 9 \to r_1$. Ninth, the conjunct is incorporated into the quantified formula. This gives:

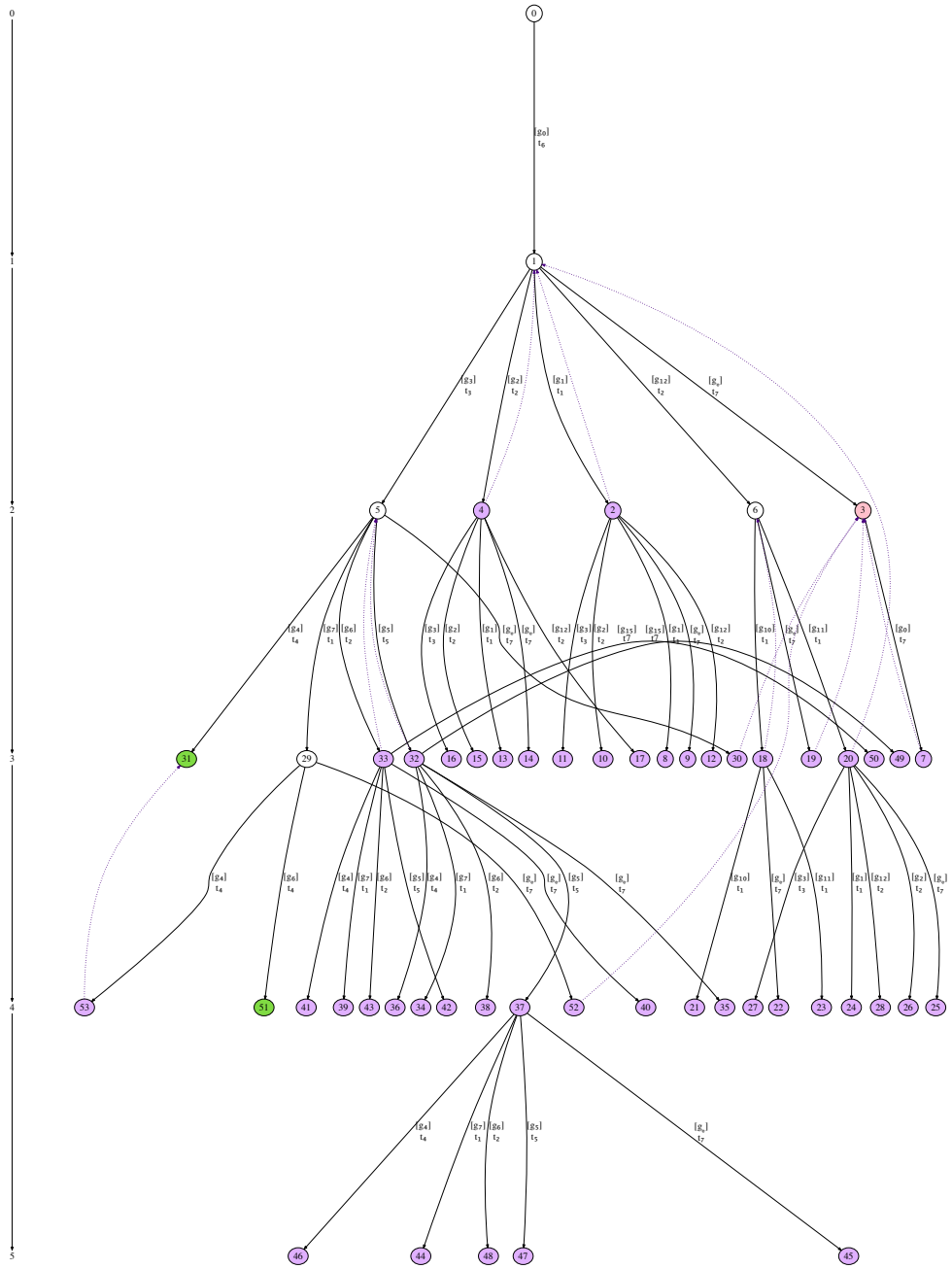


Figure 55: Step 33: Refinement of 50 (see also figure 97)

$$
\begin{aligned}
f_9 &= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to (r_1 \wedge (\forall i.i \geq 0 \to ((8 < d + i) \vee r_4)))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (d < 9 \to (\forall i.i \geq 0 \to ((8 < d + i) \vee r_4))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (d < 9 \to (\forall i.i \geq 0 \to ((d + i \leq 8) \to r_4))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (d < 9 \to (\forall i.i \geq 1 \to ((d + i \leq 9) \to r_3))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (\forall i.i \geq 1 \to (d < 9 \to ((d + i \leq 9) \to r_3))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (\forall i.i \geq 1 \to (d + i \leq 9 \to (d < 9 \to r_3))) \\
&= r_0 \wedge (d = 9 \to r_1) \wedge (d < 9 \to r_1) \wedge (\forall i.i \geq 1 \to (d + i \leq 9 \to r_3)) \\
&= r_0 \wedge (d + 0 = 9 \to r_1) \wedge (d + 0 < 9 \to r_1) \wedge (\forall i.i \geq 1 \to (d + i \leq 9 \to r_3)) \\
&= r_0 \wedge (d + 0 \leq 9 \to r_1) \wedge (\forall i.i \geq 1 \to (d + i \leq 9 \to r_3)) \\
&= r_0 \wedge (\forall i.i \geq 0 \to (d + i \leq 9 \to r_3))
\end{aligned}
$$

The purpose of these rewrites is to convince the reader that the entailment $f_8 \models f_9$ holds. ImpactExplorer uses an SMT solver to prove $f_8 \models f_9$. The check $f_8 \models f_9$ can be realized as the unsatisfiability check $f_8 \wedge \neg f_9$. An unsatisfiability argument may be easier to mechanize but rewrites are easier for human comprehension.

4.3.17 Step 32: set 33 g_2 and Step 33: refine 50

The analogous steps to those above are performed on vertex 33 and its subsequent error location, vertex 50, which results in a covering arc between 33 and 5. These steps give figures 54 and 55 respectively.

4.3.18 Step 34: expand 51 and Step 35: refine 55

Now that the branches of vertex 32 and 33 have been closed again, the development of the branch containing vertex 29 continues. Vertex 51 corresponds to automaton location *stop* and is the only uncovered leaf in the branch. It is expanded (figure 56), and its error vertex, 55, refined (figure 57). After doing so, the formula at 51 strengthens from f_0 to f_1 and, in doing so, a covering arc can now be added between 51 and 29. Since all children of 29 are now covered, the branch is closed.

4.3.19 Step 36: expand 31 and Step 37: refine 58

Now one final branch remains for development. Its leaf vertex 31 corresponds to *on-time*. Once it is expanded (figure 58) and its error vertex, 58, refined (figure 59), vertex 31 is covered by vertex 1. However, the previously undeveloped vertex 53 was being covered by vertex 31, but is no longer, so focus switches to 53.

4.3.20 Step 38: expand 53 and Step 39: refine 63

Vertex 53 is a leaf corresponding to automaton location *on-time*. It is expanded and its error location (figure 60), vertex 63, refined (figure 61). The formula at 53 is strengthened to f_1 which allows it to be covered by vertex 1. Thus, a covering arc is added from 53 to 1. Following this step, all leaves are covered and the formula

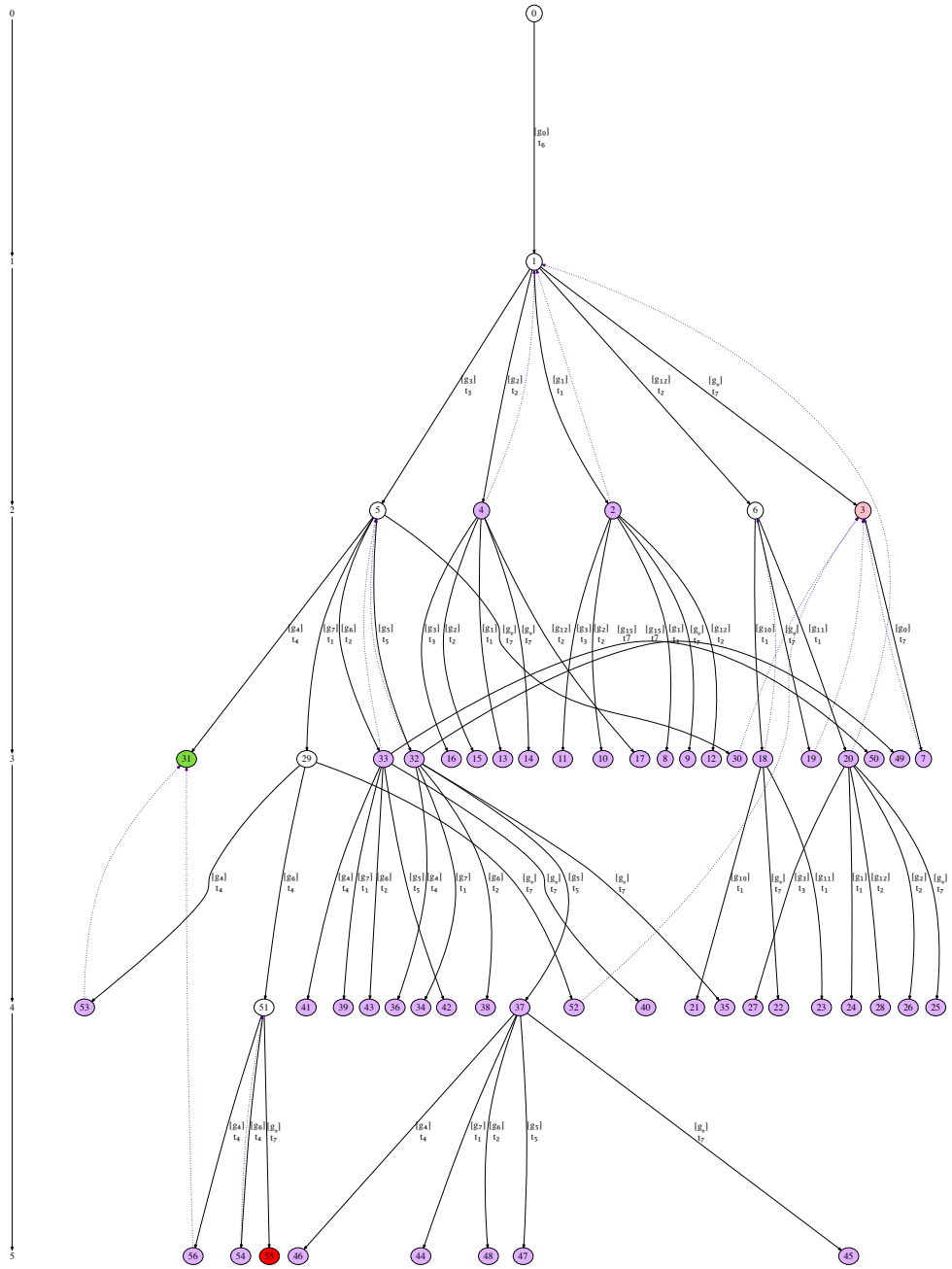


Figure 56: Step 34: Expansion of 51 (see also figure 98)

at vertex 0 remains $\top$. Vertex 0 is coloured gold to highlight the significance of this step. There are no conditions under which an error location can be reached, hence the automaton is proven safe (in a finite number of steps).

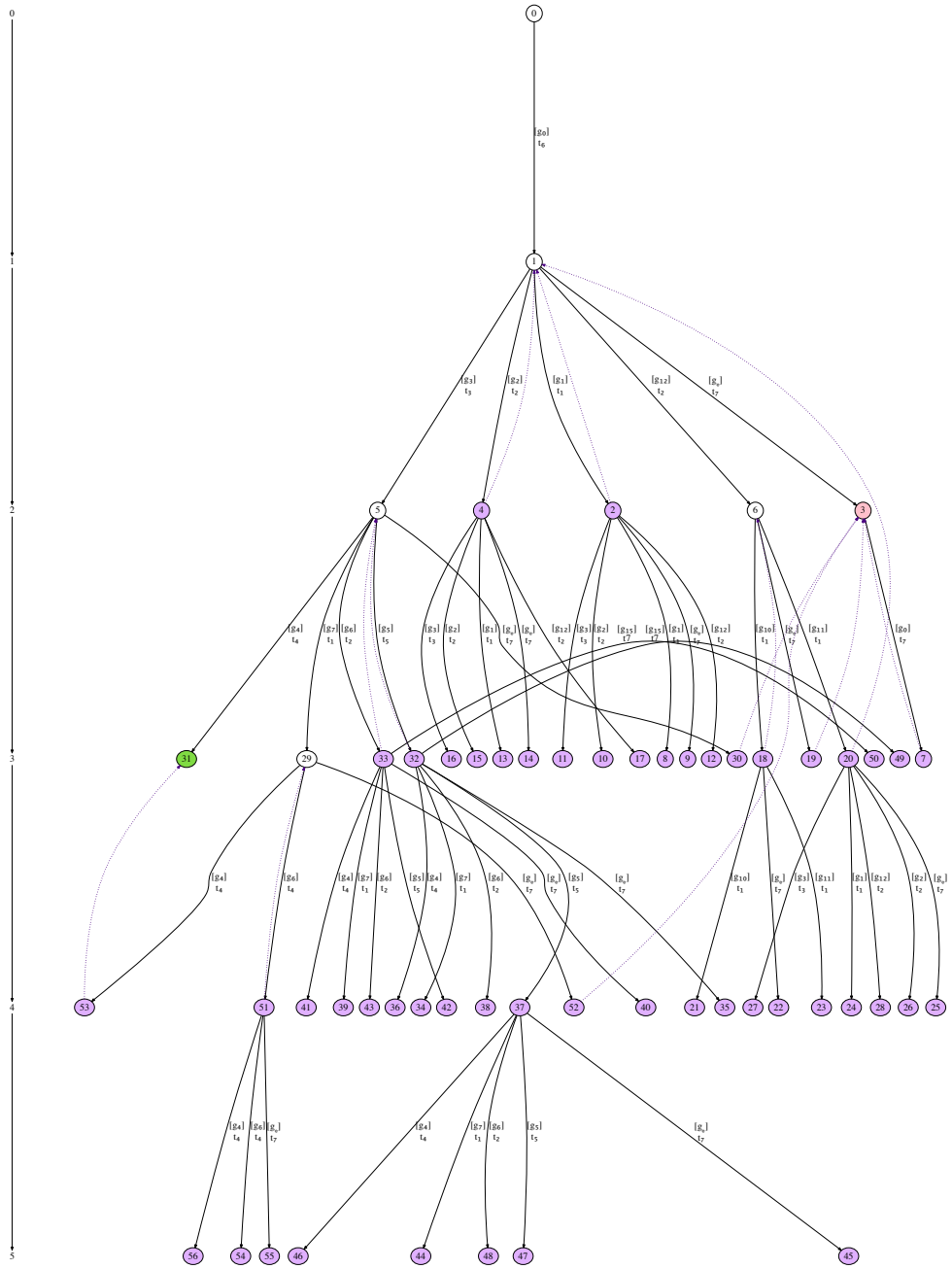


Figure 57: Step 35: Refinement of 55 (see also figure 99)

To summarize, relaxing the interval bounds of a quantified formula provides a natural way to strengthen a conjunctive formula and thereby derive a stable covering relation. This is key to termination and the derivation of a finite safety

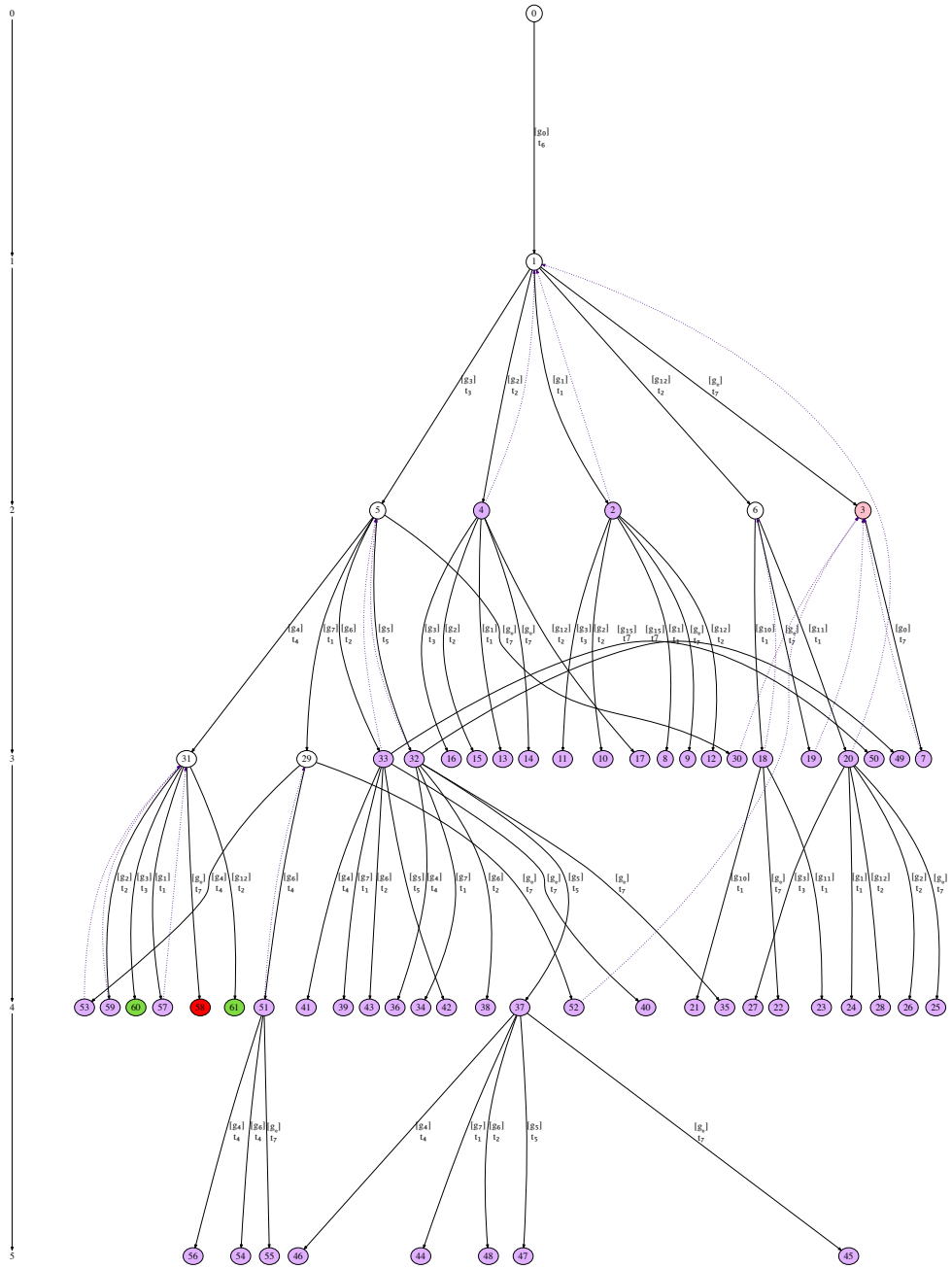


Figure 58: Step 36: Expansion of 31 (see also figure 100)

proof. Furthermore, the method does not require an interpolating SMT solver over a richer theory: the quantified formula is derived merely by recognizing structural similarity in repeating conjuncts. Moreover, it should be emphasized that the

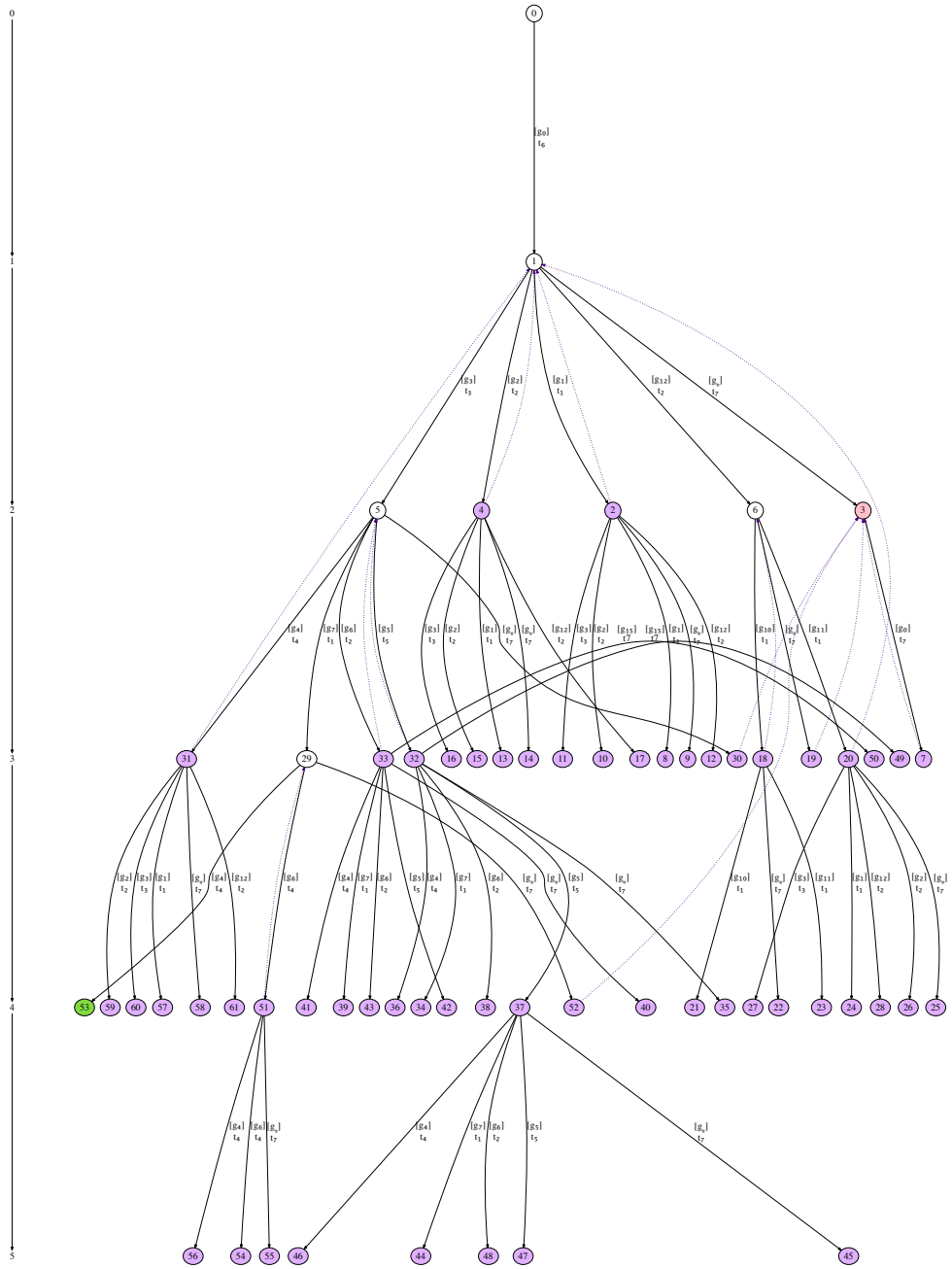


Figure 59: Step 37: Refinement of 58 (see also figure 101)

complex (nine-step) symbolic manipulation of section 4.3.16 was provided solely as a way to convince the reader that the entailment $f_8 \models f_9$ holds. The completeness of modern SMT solvers provides a decision procedure for satisfiability, hence

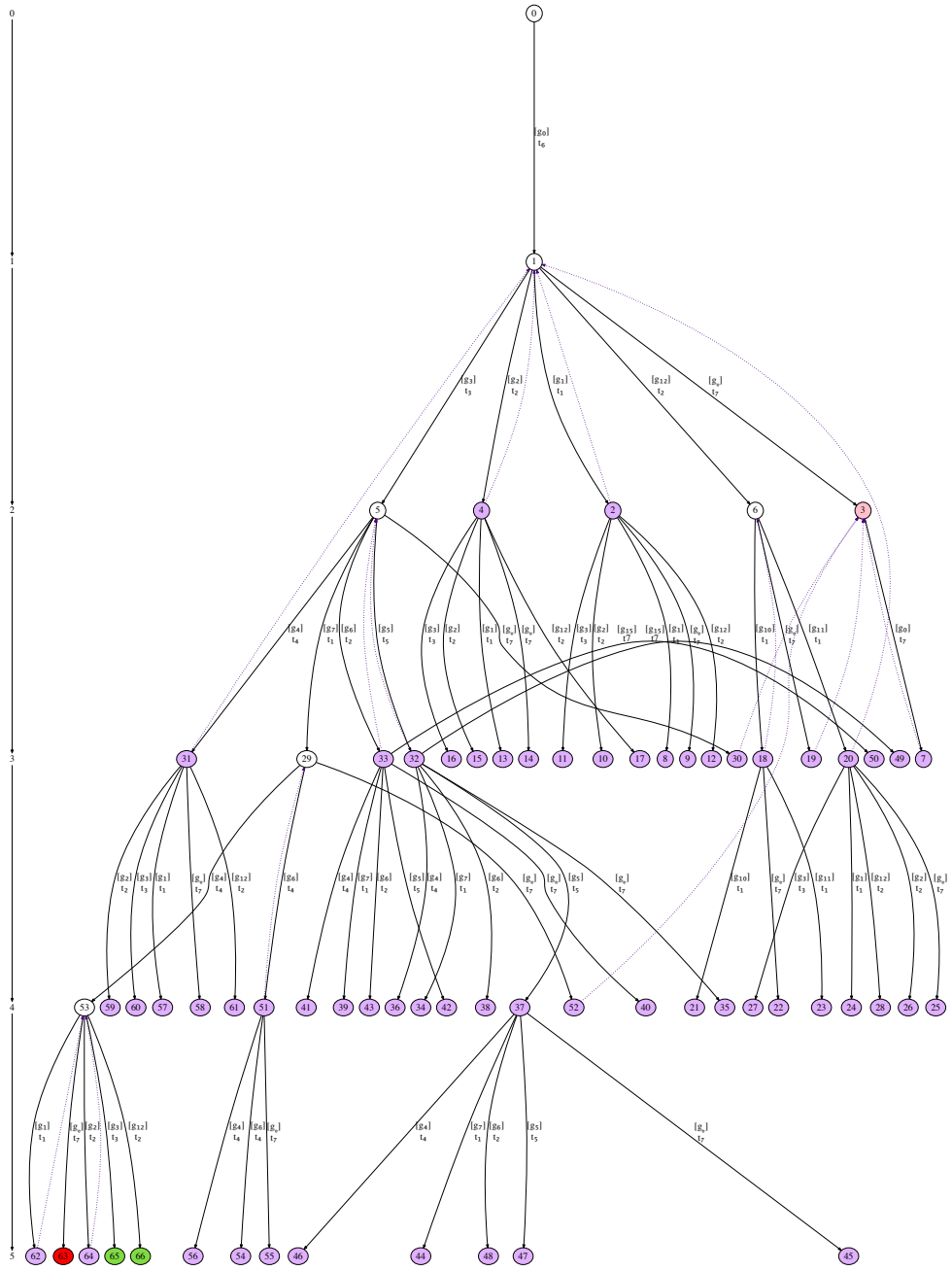


Figure 60: Step 38: Expansion of vertex 53 (see also figure 102)

unsatisfiability, hence entailment, which fully automates this step.

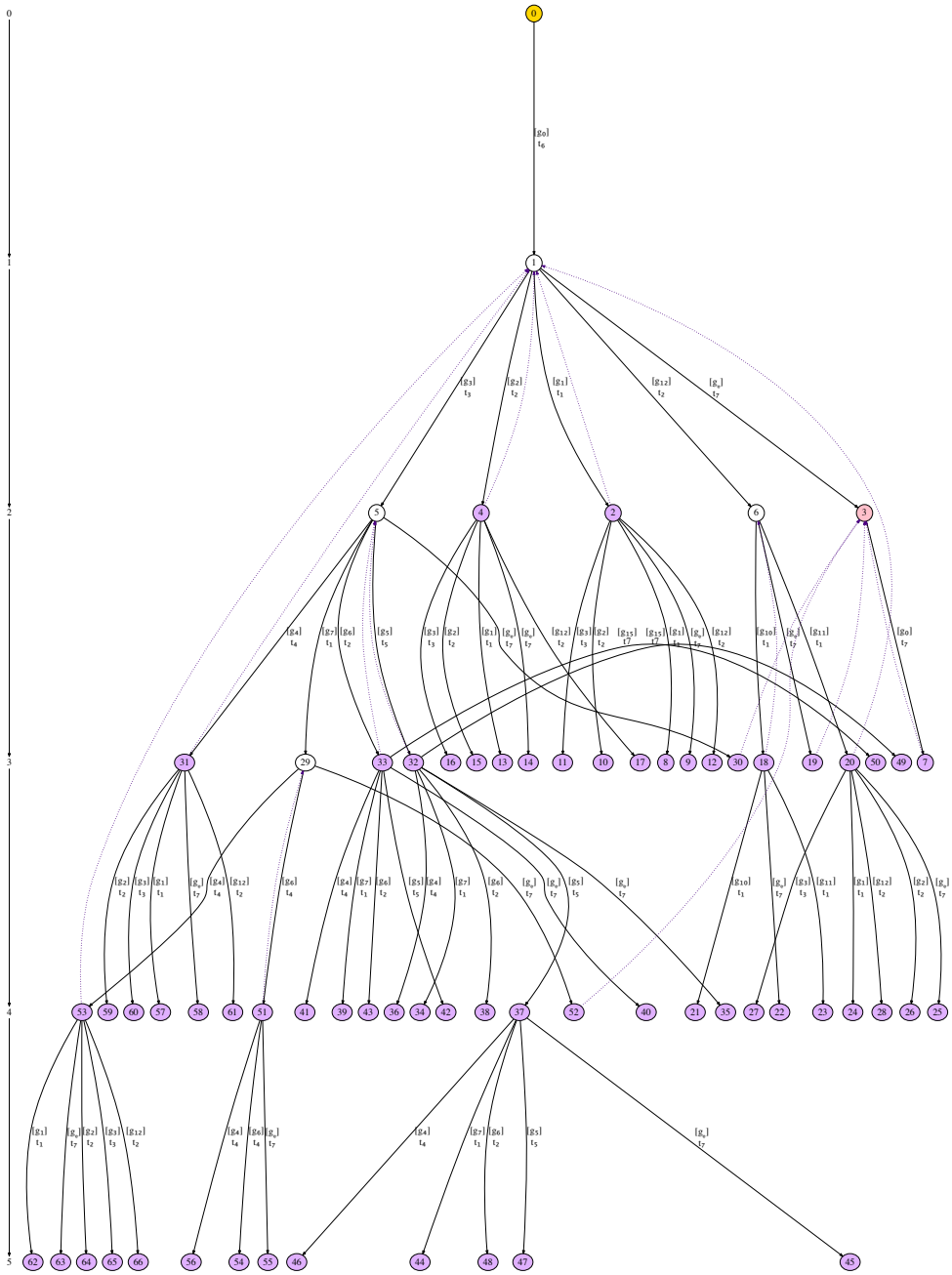


Figure 61: Step 39: Refinement of 63 (see also figure 103)

4.4 Concluding discussion

This discussion reflects both on the IMPACT algorithm, and further afield to techniques that combine widening with interpolation-based model checking.

It is arguable that the lack of suitable visualization tools has impeded the development of IMPACT since, without them, it would seem impossible to diagnose the termination problem that manifests as extra conjuncts are added to formulae through strengthening. Understanding formula growth is a level beyond understanding the mechanics of the algorithm, which are themselves non-obvious. It is telling that the IMPACT paper [52] contains a subtle error in that the guard in the cover method omits the check that w is not already covered. Furthermore, the corresponding code base [6] actually corrects this error but gives '???' against the correction, suggesting the implementors are not sure themselves. It is only by visualizing runs of the algorithm and studying the development of the formulae and covering relations that these points become clear.

The IMPACT algorithm [52] leaves the choice on how to develop the unwinding graph open; that is, the order in which the children of a vertex are expanded is not prescribed. Actually, in general, there is latitude when covering and refinement is applied. Covering should be applied as eagerly as possible. This is to prevent a vertex being developed into a subgraph when the vertex can be covered, rendering the development unnecessary. Equally, the refinement of an error vertex should also be eager since refinement will propagate conditions up the path to the root vertex. Strengthening a formula could well compromise covering but it will immediately focus the algorithm on vertices which do need development, namely the uncovered vertices.

Widening is a classical technique for accelerating convergence [25, section 9.1.3]; first demonstrated for intervals, where a binary widening operator is repeatedly applied to give an increasing chain of abstractions that is not strictly increasing. Widening can be followed by narrowing [25, section 9.3.4] which derives a decreasing chain of abstractions, starting at a safe abstraction (a post-fixpoint). Widening thus performs extrapolation by dropping information to finitely derive a post-fixpoint; narrowing then refines the post-fixpoint to recover lost information, again by only calculating a finite number of iterations. In the setting of this

thesis, the formulae that develop at vertices of the unwinding graph are progressively strengthened. Then, when a bound on a quantified variable is removed, the formula is further strengthened by enlarging the domain over which the quantified variable ranges. This is often sufficient to induce covering which is the rationale behind the tactic.

The widening used in this thesis can be refined to a staged widening [28] where bounds are relaxed to increasingly large predefined thresholds. It is even conceivable that these thresholds could be inferred themselves [61]. The formula strengthening technique introduced in this thesis resonates with work on widening recurrence equations over dependence-graph abstractions [62] where the objective is to introduce a new variable to express a closed-form recurrence.

Cousot interprets interpolation as a form of abstract interpretation [24], observing that formula strengthening, as induced by refinement, is a form of narrowing. McMillan [53], in a talk abstract, comments that the key difference between widening/narrowing and interpolation is that the former drop information and then attempt to recover it, whereas the latter generates a so-called parsimonious proof [39]; that is, a proof stripped of superfluous detail. Neither of these works show how IMPACT can be improved with widening.

Chapter 5

Widening Formulae

5.1 Introduction

Recall that IMPACT is compromised by termination problems that stem from conjunctive formulae where the conjuncts differ only in numeric constants. This chapter provides a methodology for deriving quantified formulae that compactly summarize conjunctive formulae. Not only are quantified formulae more compact, but they suggest how a range can be relaxed to derive a formula which is, in general, necessary for termination within IMPACT. To illustrate the concept, consider the formula f defined thus:

$$f \;=\; (2x + 3 = 0 \rightarrow y = 12) \;\wedge\; (2x + 5 = 0 \rightarrow y = 24) \;\wedge\; (2x + 7 = 0 \rightarrow y = 36)$$

This chapter shows how to apply marches to derive the quantified formula:

$$\forall i \in [-2, 0].\, (2x + (7 + 2i) = 0 \rightarrow y = (36 + 12i))$$

which is semantically equivalent to f. Relaxing the interval to $[-2, \infty]$, for example, then suggests a quantified formula, akin to a loop invariant, which is sufficient for covering, hence termination in IMPACT. Our methodology for deriving these quantified formulae is multi-stage. First, we consider the problem of how to abstract a set of distinct polynomials, which differ in their numeric constants, into a single summary polynomial. These polynomials stem from different expressions

which arise in the conjuncts of a conjunctive formula. The summary polynomial is defined over an enlarged set of variables and acts as a template polynomial. When values are provided for these additional variables, namely the distinct numeric constants, the original polynomials are then recovered by the application of staged polynomial evaluation. This justifies the template.

A template polynomial provides a compact summary of a set of polynomials. With an eye to deriving compact representations of formulae, the technique is lifted from a set of polynomials to the problem of abstracting a set of formulae as a template formula. Analogous to before, staged evaluation of formulae justifies the use of a template. The act of summarizing formulae as a template introduces a new variable which, in the context of a conjunctive formula, is bound by a universal quantifier. The quantified variable ranges over an interval. The methodology is completed by relaxing the interval over which the quantified formula is defined.

It is interesting to observe that the universally quantified formula thus derived can be often rendered more compactly by using the conjuncts derived by substituting in the minimal and maximal values from the interval. Even if termination is not a problem, keeping formulae as simple as possible is attractive to aid human comprehension and can only enhance the performance of IMPACT. Therefore this chapter studies formula compression based on this technique.

5.2 Parametric polynomials

Before describing parametric polynomials, it is useful to clarify the structure of polynomials themselves. Let $\vec{x} = \langle x_1, \ldots, x_n \rangle$ be a vector of variables. A monomial over $\vec{x}$ is an expression $\vec{x}^{\vec{\alpha}} = x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ where $\vec{\alpha} = \langle \alpha_1, \ldots, \alpha_n \rangle \in \mathbb{N}^n$ where $\mathbb{N}$ is taken to include zero. A term over $\vec{x}$ is an expression $t = c\vec{x}^{\vec{\alpha}}$ where $c \in \mathbb{Z}$ is the coefficient and $\vec{x}^{\vec{\alpha}}$ the monomial of t. A polynomial over $\vec{x}$ is an expression $t_1 + \cdots + t_s$ where each t_i is a term over $\vec{x}$, the case $s = 0$ corresponding to the 0 polynomial. Let $\mathbb{Z}[\vec{x}]$ denote the set of polynomials with integer coefficients over the variables $\vec{x}$.

Suppose $\vec{a} \in \mathbb{Z}^n$ and $p \in \mathbb{Z}[\vec{x}]$ where $|\vec{x}| = n$. Then let $[\![p]\!]_{\vec{x}}(\vec{a})$ denote evaluating p at $\vec{a}$, henceforth called polynomial evaluation, by substituting a_i for x_i in p, and calculating the resulting arithmetical expression.

Example 34. To illustrate polynomial evaluation, consider $\vec{x} = \langle a, b, c\rangle$ and the (linear) polynomial $p = 2a + b + 3c$. Observe $p \in \mathbb{Z}[\vec{x}]$. Then the act of evaluating p on $\vec{a} = \langle 1, 5, 5\rangle$ is encapsulated by $[\![p]\!]_{\vec{x}}(\vec{a}) = 2(1) + 5 + 3(5) = 22$.

Polynomial evaluation lifts to a vector of polynomials $\vec{p} = \langle p_1, \ldots, p_m\rangle \in \mathbb{Z}[\vec{x}]^m$ as follows: $[\![\vec{p}]\!]_{\vec{x}}(\vec{a}) = \langle [\![p_1]\!]_{\vec{x}}(\vec{a}), \ldots, [\![p_m]\!]_{\vec{x}}(\vec{a})\rangle$. It further lifts to a set of vectors $A \subseteq \wp(\mathbb{Z}^n)$ by $[\![\vec{p}]\!]_{\vec{x}}(A) = \{[\![\vec{p}]\!]_{\vec{x}}(\vec{a}) \mid \vec{a} \in A\}$. Polynomial evaluation also induces a notion of semantic equivalence:

Definition 21 (Semantic equivalence for polynomials)**.** Let $p_1, p_2 \in \mathbb{Z}[\vec{x}]$ where $|\vec{x}| = n$. Then $p_1 \equiv p_2$ iff $[\![p_1]\!]_{\vec{x}}(\vec{a}) = [\![p_2]\!]_{\vec{x}}(\vec{a})$ for all $\vec{a} \in \mathbb{Z}^n$.

Example 35. To illustrate semantic equivalence, consider $p_1 = c + a + b + a - c$ and $p_2 = 2a + b$ which are syntactically distinct. To observe equivalence, let $\vec{a} = \langle a_1, a_2, a_3\rangle$. Then

$$[\![p_1]\!]_{\vec{x}}(\vec{a}) = a_3 + a_1 + a_2 + a_1 - a_3 = 2a_1 + a_2 = [\![p_2]\!]_{\vec{x}}(\vec{a})$$

The distinction between semantic and syntactic manifests in the formulation of staged evaluation.

The polynomial evaluation $[\![p]\!]_{\vec{x}}(\vec{a})$ evaluates p to give an integral value which is derived by substituting each variable of $\vec{x}$ with the corresponding integer of $\vec{a}$. Consider now a polynomial over two disjoint vectors of variables, $\vec{x}$ and $\vec{y}$. A generalization of polynomial evaluation is staged polynomial evaluation where the variables of $\vec{x}$ are substituted with integral values but the variables of $\vec{y}$ remain free. Since the result is parametric on $\vec{y}$, the result of staged polynomial evaluation is a polynomial over $\vec{y}$.

Definition 22 (Staged evaluation for polynomials)**.** Given $p \in \mathbb{Z}[\vec{x}{:}\vec{y}]$ where $|\vec{x}| = n$ and $|\vec{y}| = m$ and $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. If $\vec{a} \in \mathbb{Z}^n$, then $p[\vec{x} \to \vec{a}] \in \mathbb{Z}[\vec{y}]$ where $[\![p[\vec{x} \to \vec{a}]]\!]_{\vec{y}}(\vec{b}) = [\![p]\!]_{\vec{x}:\vec{y}}(\vec{a}{:}\vec{b})$ for all $\vec{b} \in \mathbb{Z}^m$.

Staged evaluation of polynomials is formulated in terms of the semantics of the polynomial $p[\vec{x} \to \vec{a}]$, rather than how it is actually derived from the polynomial p. Staged polynomial evaluation, that is $p[\vec{x} \to \vec{a}]$, is considered a mapping for a given p, $\vec{x}$ and $\vec{a}$ – in effect it chooses from the many candidate polynomials which satisfy the semantic equivalence requirement.

Example 36. Let p_1 and p_2 be as in example 35. Suppose $\vec{x} = \langle a, c\rangle$ and $\vec{y} = \langle b\rangle$ so $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. Given $\vec{a} = \langle 4, 2\rangle$, then $p_1[\vec{x} \to \vec{a}] = q_1$ where $q_1 \equiv 8 + b$. Likewise $p_2[\vec{x} \to \vec{a}] = q_2$ where $q_2 \equiv 8 + b$.

Ultimately the aim is to abstract a set of distinct polynomials, which differ in their numeric constants, into a single summary polynomial. This summary polynomial is defined over an enlarged set of variables and acts as a template polynomial. When values are provided for these additional variables, namely the distinct numeric constants, the original polynomials are then recovered by the application of staged polynomial evaluation. This process, being the dual of creating the template, is akin to concretization in abstract interpretation (section 2.5.2).

Example 37. Consider summarizing the set of polynomials $Q = \{q_1, q_2, q_3\}$, where

$$q_1 = 2x + y + 3 \qquad q_2 = 2x - y + 5 \qquad q_3 = 2x - 3y + 7$$

Observe $q_1, q_2, q_3 \in \mathbb{Z}[\vec{y}]$ where $\vec{y} = \langle x, y\rangle$. Let $p = ax + by + c$ and $\vec{x} = \langle a, b, c\rangle$. To observe that $p \in \mathbb{Z}[\vec{x}{:}\vec{y}]$ serves as a template formula for Q, consider the following staged polynomial evaluations:

$$p[\vec{x} \to \langle 2, 1, 3\rangle] \equiv q_1 \qquad p[\vec{x} \to \langle 2, -1, 5\rangle] \equiv q_2 \qquad p[\vec{x} \to \langle 2, -3, 7\rangle] \equiv q_3$$

5.3 Parametric formulae

A template polynomial provides a compact summary of a set of polynomials. In order to derive a compact representation of a formula, we lift the technique for abstracting a set of polynomials to the problem of abstracting a set of formulae into a template formula. Sets of formulae naturally arise as conjuncts so the technique provides a way of deriving a compact symbolic representation of a conjunctive formula. Analogous to before, the act of summarizing formulae in a template introduces new variables which, in the context of a conjunctive formula, become bound by a universal quantifier. To present this technique, first the formulae over which this method operates is clarified:

Definition 23. Given a vector of variables $\vec{v}$, the set of formulae, $\mathsf{Form}[\vec{v}]$ is the

least set such that:

$$
\begin{array}{rcl}
\top \in \mathsf{Form}[\vec{v}] & & \\
\bot \in \mathsf{Form}[\vec{v}] & & \\
p < 0 \in \mathsf{Form}[\vec{v}] & \text{if} & p \in \mathbb{Z}[\vec{v}] \\
p > 0 \in \mathsf{Form}[\vec{v}] & \text{if} & p \in \mathbb{Z}[\vec{v}] \\
p \leq 0 \in \mathsf{Form}[\vec{v}] & \text{if} & p \in \mathbb{Z}[\vec{v}] \\
p \geq 0 \in \mathsf{Form}[\vec{v}] & \text{if} & p \in \mathbb{Z}[\vec{v}] \\
p = 0 \in \mathsf{Form}[\vec{v}] & \text{if} & p \in \mathbb{Z}[\vec{v}] \\
\neg f \in \mathsf{Form}[\vec{v}] & \text{if} & f \in \mathsf{Form}[\vec{v}] \\
f_1 \rightarrow f_2 \in \mathsf{Form}[\vec{v}] & \text{if} & f_1, f_2 \in \mathsf{Form}[\vec{v}] \\
f_1 \leftrightarrow f_2 \in \mathsf{Form}[\vec{v}] & \text{if} & f_1, f_2 \in \mathsf{Form}[\vec{v}] \\
\wedge \wr f_1, \ldots, f_n \int \in \mathsf{Form}[\vec{v}] & \text{if} & \{f_1, \ldots, f_n\} \subseteq \mathsf{Form}[\vec{v}] \\
\vee \wr f_1, \ldots, f_n \int \in \mathsf{Form}[\vec{v}] & \text{if} & \{f_1, \ldots, f_n\} \subseteq \mathsf{Form}[\vec{v}] \\
\forall v_i \in I.\, f \in \mathsf{Form}[\vec{v}] & \text{if} & I \in \mathsf{Int} \text{ and } f \in \mathsf{Form}[\vec{v}] \\
\exists v_i \in I. f \in \mathsf{Form}[\vec{v}] & \text{if} & I \in \mathsf{Int} \text{ and } f \in \mathsf{Form}[\vec{v}]
\end{array}
$$

Observe that in the above definition, for full generality, a conjunctive formula is formulated as a multiset of formulae which are conjoined, rather than as a set of formulae. This multiset formulation mirrors the syntactic form of formulae, which can include duplicate conjuncts. Although the primary aim is to reason about universally quantified formulae, for completeness, the class $\mathsf{Form}[\vec{v}]$ also includes existentially quantified formulae. Note that $v_i \in I$ is actually syntactic sugar for $v_i \in \gamma(I)$, hence v_i is integral (see section 3.2).

Since formulae express relationships between polynomials, we lift polynomial evaluation to the concept of formula evaluation. Like polynomial evaluation, formulae evaluation is a map operating on an input vector of integers. Unlike polynomial evaluation, however, formula evaluation derives a truth value.

Definition 24. Given $f \in \mathsf{Form}[\vec{v}]$, evaluation $[\![f]\!]_{\vec{v}} : \mathbb{Z}^{|\vec{v}|} \to \{\top, \bot\}$ is defined:

$$\begin{aligned}
[\![\top]\!]_{\vec{v}}(\vec{a}) &= \top \\
[\![\bot]\!]_{\vec{v}}(\vec{a}) &= \bot \\
[\![p < 0]\!]_{\vec{v}}(\vec{a}) &= ([\![p]\!]_{\vec{v}}(\vec{a}) < 0) \\
[\![p > 0]\!]_{\vec{v}}(\vec{a}) &= ([\![p]\!]_{\vec{v}}(\vec{a}) > 0) \\
[\![p \leq 0]\!]_{\vec{v}}(\vec{a}) &= [\![\neg(p > 0)]\!]_{\vec{v}}(\vec{a}) \\
[\![p \geq 0]\!]_{\vec{v}}(\vec{a}) &= [\![\neg(p < 0)]\!]_{\vec{v}}(\vec{a}) \\
[\![p = 0]\!]_{\vec{v}}(\vec{a}) &= [\![\wedge \wr p \leq 0, p \geq 0 \int]\!]_{\vec{v}}(\vec{a}) \\
[\![\neg f]\!]_{\vec{v}}(\vec{a}) &= \neg [\![f]\!]_{\vec{v}}(\vec{a}) \\
[\![f_1 \to f_2]\!]_{\vec{v}}(\vec{a}) &= [\![f_1]\!]_{\vec{v}}(\vec{a}) \to [\![f_2]\!]_{\vec{v}}(\vec{a}) \\
[\![f_1 \leftrightarrow f_2]\!]_{\vec{v}}(\vec{a}) &= [\![f_1 \to f_2 \wedge f_2 \to f_1]\!]_{\vec{v}}(\vec{a}) \\
[\![\wedge \wr f_1, \ldots, f_n \int]\!]_{\vec{v}}(\vec{a}) &= \wedge_{i=1}^{n} [\![f_i]\!]_{\vec{v}}(\vec{a}) \\
[\![\vee \wr f_1, \ldots, f_n \int]\!]_{\vec{v}}(\vec{a}) &= \vee_{i=1}^{n} [\![f_i]\!]_{\vec{v}}(\vec{a}) \\
[\![\forall v_i \in I.\, f]\!]_{\vec{v}}(\vec{a}) &= \wedge_{a' \in \gamma(I)} [\![f]\!]_{\vec{v}}(\langle a_1, \ldots, a_{i-1}, a', a_{i+1}, \ldots, a_n \rangle) \\
[\![\exists v_i \in I.f]\!]_{\vec{v}}(\vec{a}) &= \vee_{a' \in \gamma(I)} [\![f]\!]_{\vec{v}}(\langle a_1, \ldots, a_{i-1}, a', a_{i+1}, \ldots, a_n \rangle)
\end{aligned}$$

where $\vec{a} = \langle a_1, \ldots, a_n \rangle$.

Note that $\top$ and $\bot$ represent universally satisfiable and unsatisfiable formulae, hence evaluate to the truth values $\top$ and $\bot$ respectively, regardless of the vector $\vec{a}$. Semantic evaluation of a universally quantified formula, which is quantified over the interval I, can be considered a two-part process. First, the formula f is evaluated, replacing the i-th component of the vector $\vec{a}$ with each of the integers described by the interval (where the index i is determined by the position of the variable v_i in the vector $\vec{v}$). Second, the truth values are composed using conjunction. Existential quantified formulae are defined analogously using disjunction.

Formula evaluation lifts to a vector of formulae $\langle f_1, \ldots, f_n \rangle \in \mathsf{Form}[\vec{v}]^n$ as follows: $[\![\langle f_1, \ldots, f_n \rangle]\!]_{\vec{v}}(\vec{a}) = \langle [\![f_1]\!]_{\vec{v}}(\vec{a}), \ldots, [\![f_n]\!]_{\vec{v}}(\vec{a}) \rangle$. Like before, formula evaluation induces a notion of semantic equivalence. However, unlike integer values, truth values are related by logical implication, which additionally induces a notion of semantic entailment:

Definition 25 (Semantic equivalence for formulae)**.** Let $f_1, f_2 \in \mathsf{Form}[\vec{v}]$. Then $f_1 \equiv_{\vec{v}} f_2$ iff $[\![f_1]\!]_{\vec{v}}(\vec{a}) = [\![f_2]\!]_{\vec{v}}(\vec{a})$ for all $\vec{a} \in \mathbb{Z}^{|\vec{v}|}$.

Semantic entailment for formulae is defined as a lifting of the standard entailment ordering on truth values, namely: $\bot \models \bot$, $\bot \models \top$ and $\top \models \top$.

Definition 26 (Semantic entailment for formulae)**.** Let $f_1, f_2 \in \mathsf{Form}[\vec{v}]$. Then $f_1 \models_{\vec{v}} f_2$ iff $[\![f_1]\!]_{\vec{v}}(\vec{a}) \models [\![f_2]\!]_{\vec{v}}(\vec{a})$ for all $\vec{a} \in \mathbb{Z}^{|\vec{v}|}$.

Analogous to before, staged evaluation of formulae is formulated in terms of two distinct sets of variables, $\vec{x}$ and $\vec{y}$, where the variables $\vec{x}$ are replaced with the values of the integral vector $\vec{a}$:

Definition 27 (Staged evaluation for formulae)**.** Given $f \in \mathsf{Form}[\vec{x}{:}\vec{y}]$ where $|\vec{x}| = n$ and $|\vec{y}| = m$ and $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. If $\vec{a} \in \mathbb{Z}^n$, then $f[\vec{x} \to \vec{a}] \in \mathsf{Form}[\vec{y}]$ where $[\![f[\vec{x} \to \vec{a}]]\!]_{\vec{y}}(\vec{b}) = [\![f]\!]_{\vec{x}:\vec{y}}(\vec{a}{:}\vec{b})$ for all $\vec{b} \in \mathbb{Z}^m$.

5.4 Staged symbolic evaluation

Staged evaluation of formulae mirrors that of polynomials. As before, $f[\vec{x} \to \vec{a}]$ is considered a mapping for a given f, $\vec{x}$ and $\vec{a}$ – in effect it chooses from the many candidate formulae which satisfy the stipulated requirements for semantic equivalence. An integer is merely a degenerate polynomial and therefore staged evaluation can be generalized to a symbolic counterpart where a variable x_i is bound to a polynomial p_i, as follows:

Definition 28 (Staged symbolic evaluation for formulae)**.** Given $f \in \mathsf{Form}[\vec{x}{:}\vec{y}]$ where $|\vec{x}| = n$, $|\vec{y}| = m$ and $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. If $\vec{p} = \langle p_1, \ldots, p_n \rangle$ where $p_i \in \mathbb{Z}[\vec{z}]$ and $vars(\vec{x}{:}\vec{y}) \cap vars(\vec{z}) = \emptyset$, then $f[\vec{x} \to \vec{p}] \in \mathsf{Form}[\vec{y}{:}\vec{z}]$ where $[\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{z}}(\vec{b} : \vec{c}) = [\![f]\!]_{\vec{x}:\vec{y}}([\![\vec{p}]\!]_{\vec{z}}(\vec{c}) : \vec{b})$ for all $\vec{b} \in \mathbb{Z}^m$ and $\vec{c} \in \mathbb{Z}^{|\vec{z}|}$.

This form of staged evaluation is truly symbolic since p_i is parametric in a vector of variables, here $\vec{z}$. It a moot point whether this constitutes a form of evaluation at all since each x_i is substituted with a polynomial expression p_i, rather than an integer value. As an aide-mémoire for the reader, the concept has been thus named to resonate with symbolic evaluation of programs, where the values of variables are modelled with symbolic expressions.

Example 38. Let $\vec{x} = \langle a, b, c\rangle$, $\vec{y} = \langle x, y\rangle$ and $\vec{z} = \langle i, j\rangle$. Suppose $f = (q \geq 0)$ where $q = ax + by + c \in \mathbb{Z}[\vec{x}{:}\vec{y}]$ so that $f \in \mathsf{Form}[\vec{x}{:}\vec{y}]$. Note $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. Let $\vec{p} = \langle p_1, p_2, p_3\rangle$ where:

$$p_1 = 2 + 4i + j, \quad p_2 = 3 + 3i + j \quad \text{and} \quad p_3 = 7 + j$$

so that $p_i \in \mathbb{Z}[\vec{z}]$ and $vars(\vec{x}{:}\vec{y}) \cap vars(\vec{z}) = \emptyset$. Then $f[\vec{x} \to \vec{p}] = f'$ where $f' \equiv \big((2 + 4i + j)x + (3 + 3i + j)y + (7 + j) \geq 0\big)$. Using $\vec{a} = \langle a_1, a_2\rangle$ and $\vec{c} = \langle c_1, c_2\rangle$, where the a_i and c_i are symbolic values, it can be demonstrated that $[\![f']\!]_{\vec{y}:\vec{z}}(\vec{b} : \vec{c}) = [\![f]\!]_{\vec{x}:\vec{y}}([\![\vec{p}]\!]_{\vec{z}}(\vec{c}) : \vec{b})$.

Staged symbolic evaluation provides a way to extend staged evaluation to a symbolic description of a set of vectors. To this end, the following proposition is provided as a bridging result between the concepts of staged evaluation for formulae and their staged symbolic evaluation:

Proposition 10. Let $f \in \mathsf{Form}[\vec{x}{:}\vec{y}]$ where $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. Let $\vec{p} \in \mathbb{Z}[i]^n$ and $I \in \mathsf{Int}$. Furthermore, let $A = \{\vec{a}_1, \ldots, \vec{a}_\ell\}$ and $A \subseteq [\![\vec{p}]\!]_{\vec{i}}(\gamma(I))$ where $\vec{i} = \langle i\rangle$ and $i \notin vars(\vec{x}{:}\vec{y})$. Then, for all $\vec{b} \in \mathbb{Z}^{|\vec{y}|}$

- $[\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \models [\![\wedge_{j=1}^{\ell} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b})$
- $[\![\wedge_{j=1}^{\ell} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) \models [\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0})$ if $A = [\![\vec{p}]\!]_{\vec{i}}(\gamma(I))$

Observe the left side of the first entailment is a formula which is a conjunction of staged evaluations, that differ only in the constant vectors $\vec{a}_j$, which are contained within the image of the mapping $[\![\vec{p}]\!]_{\vec{i}}$ on the set $\gamma(I)$. The right side is a universally quantified formula, the quantifier of which, i, is introduced by staged symbolic evaluation. Recall that the formula $\forall i \in I.\, f[\vec{x} \to \vec{p}]$ is evaluated by replacing the last component of the vector $\vec{b} : \vec{0}$ with each of the integers from the interval I. Hence, $\vec{0}$ plays no substantive role in the evaluation, other than padding $\vec{b}$ to be dimensionally consistent with $\vec{y} : i$. The disjointedness of $\vec{x}$ and $\vec{y}$ and the fresh variable i is a necessary requirement for staged and staged symbolic evaluation.

The force of this proposition is that is gives a way of collapsing a potentially large conjunction into a compact formula which is quantified over a fresh variable which ranges over an interval, as the following example illustrates:

Example 39. Consider the derivation of a compact representation for the conjunctive formula

$$g = (3x - 12y < 0) \wedge (5x - 24y < 0) \wedge (7x - 36y < 0)$$

where $g \in \mathsf{Form}[\vec{y}]$ and $\vec{y} = \langle x, y \rangle$. Let $b \in \mathbb{Z}^2$ and $\vec{x} = \langle r, s \rangle$. The commonality between the three conjuncts can be factored out by expressing g using staged evaluation as follows:

$$[\![g]\!]_{\vec{y}}(\vec{b}) = [\![\wedge_{j=1}^{3}(rx + sy < 0)[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b})$$

where

$$\vec{a}_1 = \langle 3, -12 \rangle \qquad \vec{a}_2 = \langle 5, -24 \rangle \qquad \vec{a}_3 = \langle 7, -36 \rangle$$

To aid the application of the proposition, put $f = (rx + sy < 0)$ and observe $f \in \mathsf{Form}[\vec{x}{:}\vec{y}]$. Note $vars(\vec{x}) \cap vars(\vec{y}) = \emptyset$. To illustrate the application of the proposition, consider $\vec{p} = \langle 7 + 2i, -36 - 12i \rangle \in \mathbb{Z}[i]$, noting $i \notin vars(\vec{x}{:}\vec{y})$. Observe

$$\vec{a}_1 = [\![\vec{p}]\!]_{\vec{i}}(-2) \qquad \vec{a}_2 = [\![\vec{p}]\!]_{\vec{i}}(-1) \qquad \vec{a}_3 = [\![\vec{p}]\!]_{\vec{i}}(0)$$

Let $I = [-2, 0] \in \mathsf{Int}$ so that $[\![\vec{p}]\!]_{\vec{i}}(\gamma(I)) = \{\vec{a}_1, \vec{a}_2, \vec{a}_3\}$. Then the application of proposition 10 gives:

$$\begin{aligned} &= [\![\wedge_{j=1}^{3} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) \\ &= [\![\forall i \in [-2, 0].\, f[\vec{x} \to \langle 7 + 2i, -36 - 12i \rangle]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \\ &= [\![\forall i \in [-2, 0].\, ((7 + 2i)x + (-36 - 12i)y < 0)]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \end{aligned}$$

with the final equality following by an application of staged symbolic evaluation. In conclusion,

$$[\![g]\!]_{\vec{y}}(\vec{b}) = [\![\forall i \in [-2, 0].\, ((7 + 2i)x + (-36 - 12i)y < 0)]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0})$$

though at this stage we omit the derivation of the polynomial $\vec{p}$ from the constants $\vec{a}_1$, $\vec{a}_2$ and $\vec{a}_3$. Note how this universally quantified formula is devoid of the fixed constants $\vec{a}_1$, $\vec{a}_2$ and $\vec{a}_3$; the coefficients of x and y are fully parametric in the

quantified variable i.

The force of proposition 10 is that the vector of polynomials, $\vec{p}$, can be mechanically derived, thereby providing a computational strategy for deriving a universally quantified formula. As an intermediate step, the following lemma asserts that a single point can be represented as a march without loss of information. The example which follows the lemma illustrates how the join operator of a march can then be applied to derive $\vec{p}$.

Lemma 29. Let $\vec{a} \in \mathbb{Z}^n$ and $m = \vec{a} + 0[1,1]\vec{1} \in \mathsf{March}^n$. Then $\gamma(m) = \{\vec{a}\}$.

Example 40. To illustrate how the $\vec{p}$ and the kI of the previous example can be found, consider $A = \{\vec{a}_1, \vec{a}_2, \vec{a}_3\}$ where $\vec{a}_1 = \langle 3, -12\rangle$, $\vec{a}_2 = \langle 5, -24\rangle$ and $\vec{a}_3 = \langle 7, -36\rangle$. Construct $m_1 = \vec{a}_1 + 0[1,1]\vec{1}$, $m_2 = \vec{a}_2 + 0[1,1]\vec{1}$ and $m_3 = \vec{a}_3 + 0[1,1]\vec{1}$. Observe $\gamma(m_j) = \{\vec{a}_j\}$ where $j \in \{1,2,3\}$. Using proposition 9, the following join is calculated:

$$m = m_1 \sqcup m_2 \sqcup m_3 = \langle 7, -36\rangle + 1[-2,0]\langle 2, -12\rangle$$

For future reference, put $\vec{a} = \langle 7, -36\rangle$, $kI = 1[-2,0]$ and $\vec{v} = \langle 2, -12\rangle$ so that $m = \vec{a} + kI\vec{v}$. Observe $|\gamma(m)| = |\gamma(1[-2,0]\langle 2, -12\rangle)| = |\gamma(-2,0)| = 3$. Moreover since $\vec{a}_1, \vec{a}_2, \vec{a}_3 \in \gamma(m)$ and $|\gamma(m)| = 3$, it follows $\gamma(m) = A$, as desired. The polynomial $\vec{p} \in \mathbb{Z}[i]^2$ can now be read off from m to give:

$$\vec{p} = \langle 7 + 2i, -36 - 12i\rangle$$

the construction merely being to put $\vec{p} = \vec{a} + ik\vec{v}$. This construction also provides the interval, I, suitable for the application of proposition 10.

Observe how example 40 dovetails with example 39 to derive a universally quantified formula from a conjunctive formula in a systematic, mechanized way. In this manner, it is possible to collapse a conjunctive formula, possibly involving many conjuncts, into a compact quantified formula. The quantified formula uses the parameter i to derive a concise representation, whilst preserving the same semantics. Observe that each component of a march $\vec{s} + kI\vec{v}$ is used in this construction: the quantified range is derived from I and the polynomial used to construct the

formula is itself derived from $\vec{s}$, k and $\vec{v}$. A march provides everything that is needed but no more. The domain can be seen as a vehicle for deriving a symbolic linear description of a finite set of points. However, it derives a particular form of symbolic description that is primed for widening. In fact, the march abstraction is designed so that widening is localized to its interval component, for which a number of widening methods have been proposed and studied (for instance, widening with thresholds [9, section 7.1.2]). The other components of the march abstraction, by design, continue to fulfil their role unaffected by interval widening. Thus the march domain summarizes points but also sets them up for generalization using standard machinery. Furthermore, when this tactic is not applicable, a march abstraction of $\top$ is returned (as described in section 3.6), indicating that the points cannot be duly summarized.

Example 41. To illustrate how a quantified formula is derived from a march with a non-unit k-multiplier, consider now:

$$g = (-12x + 51y \geq 0) \wedge (-18x + 75y < 0) \wedge (-24x + 99y < 0) \wedge (-30x + 123y < 0)$$

where $g \in \mathsf{Form}[\vec{x}]$ and $\vec{x} = \langle x, y \rangle$. Let $b \in \mathbb{Z}^2$. Consider $A = \{\vec{a}_1, \vec{a}_2, \vec{a}_3, \vec{a}_4\}$ and observe

$$[\![g]\!]_{\vec{y}}(\vec{b}) = [\![\wedge_{j=1}^{4}(rx + sy \geq 0)[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b})$$

where $\vec{a}_1 = \langle -12, 51 \rangle$, $\vec{a}_2 = \langle -18, 75 \rangle$, $\vec{a}_3 = \langle -24, 99 \rangle$ and $\vec{a}_4 = \langle -30, 123 \rangle$. Define $m_1 = \vec{a}_1 + 0[1,1]\vec{1}$, $m_2 = \vec{a}_2 + 0[1,1]\vec{1}$, $m_3 = \vec{a}_3 + 0[1,1]\vec{1}$, $m_4 = \vec{a}_4 + 0[1,1]\vec{1}$ and calculate:

$$m = m_1 \sqcup m_2 \sqcup m_3 \sqcup m_4 \equiv \langle -30, 123 \rangle + 6[0,3]\langle 1, -4 \rangle$$

Put $\vec{p} = \vec{a} + ik\vec{v} = \langle -30 + 6i, 123 - 24i \rangle \in \mathbb{Z}[i]$ and observe

$$\vec{a}_1 = [\![\vec{p}]\!]_{\vec{i}}(3) \qquad \vec{a}_2 = [\![\vec{p}]\!]_{\vec{i}}(2) \qquad \vec{a}_3 = [\![\vec{p}]\!]_{\vec{i}}(1) \qquad \vec{a}_4 = [\![\vec{p}]\!]_{\vec{i}}(0)$$

so that by proposition 10:

$$[\![g]\!]_{\vec{y}}(\vec{b}) = [\![\forall i \in [0,3].\ ((-30 + 6i)x + (123 - 24i)y \geq 0)]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0})$$

thus the resultant quantified formula is expressed in the very simplest terms, using merely an interval, even if it is derived from a march with a non-unit k-multiplier.

5.5 Collapsing formulae

In what follows, we explore how proposition 10 suggests a general tactic for compactly representing conjunctive formulae.

Example 42. One might expect from proposition 10 that it would be possible to find a formula where its semantics differs to those of the derived quantified summary formula. To this end, consider the derivation for the conjunctive formula:

$$h = (5x \geq 0) \wedge (10x - 2y \geq 0) \wedge (20x - 6y \geq 0)$$

where $h \in \mathsf{Form}[\vec{y}]$ and $\vec{y} = \langle x, y \rangle$. Let $b \in \mathbb{Z}^2$ and $\vec{x} = \langle r, s \rangle$. Again, the commonality between the three conjuncts can be factored out by expressing h using staged evaluation as follows:

$$[\![h]\!]_{\vec{y}}(\vec{b}) = [\![\wedge_{j=1}^{3}(rx + sy \geq 0)[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b})$$

where

$$\vec{a}_1 = \langle 5, 0 \rangle \qquad \vec{a}_2 = \langle 10, -2 \rangle \qquad \vec{a}_3 = \langle 20, -6 \rangle$$

Now put $m_1 = \vec{a}_1 + 0[1,1]\vec{1}$, $m_2 = \vec{a}_2 + 0[1,1]\vec{1}$ and $m_3 = \vec{a}_3 + 0[1,1]\vec{1}$ and calculate their join:

$$m = m_1 \sqcup m_2 \sqcup m_3 = \langle 20, -6 \rangle + 1[-3, 0]\langle 5, -2 \rangle$$

Thus $\vec{s} = \langle 20, -6 \rangle$, $kI = 1[-3, 0]$ and $\vec{v} = \langle 5, -2 \rangle$. Observe in this case $|\gamma(m)| = |\gamma(-3, 0)| = 4 > |A|$ where $A = \{\vec{a}_1, \vec{a}_2, \vec{a}_3\}$. Indeed observe $\langle 15, -4 \rangle \in \gamma(m)$ but $\langle 15, -4 \rangle \notin A$. Nevertheless, reading off the polynomial $\vec{p} \in \mathbb{Z}[i]^2$ gives:

$$\vec{p} = \langle 20 + 5i, -6 - 2i \rangle$$

Observe, by the first case of proposition 10:

$$\begin{aligned}
& [\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \\
= \; & [\![\forall i \in [-3, 0].\, (rx + sy \geq 0)[\vec{x} \to \langle 20 + 5i, -6 - 2i\rangle]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \\
= \; & [\![(5x \geq 0) \wedge (10x - 2y \geq 0) \wedge (15x - 4y \geq 0) \wedge (20x - 6y \geq 0)]\!]_{\vec{y}}(\vec{b}) \\
\models \; & [\![(5x \geq 0) \wedge (10x - 2y \geq 0) \wedge (20x - 6y \geq 0)]\!]_{\vec{y}}(\vec{b}) \\
= \; & [\![\wedge_{j=1}^{3} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) \\
= \; & [\![h]\!]_{\vec{y}}(\vec{b})
\end{aligned}$$

Since $A \neq [\![\vec{p}]\!]_{\vec{i}}(\gamma(I))$, the second case of proposition 10 is not guaranteed to hold. However, the quantified formula, which acts as a summary, has exactly the same number of solutions as the formula which was summarized. This can be verified with an SMT solver.

Example 43. Previously the quantified formula equates to a conjunction of four conjuncts. Now observe $\gamma(I) = \{-3, -2, -1, 0\}$. Surprisingly, if the first and the last conjuncts are used, that is $i = -3$ or $i = 0$, equivalence still holds:

$$[\![\forall i \in \{-3, 0\}.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) = [\![\wedge_{j=1}^{3} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = [\![h]\!]_{\vec{y}}(\vec{b})$$

This equivalence can also be verified symbolically using an SMT solver.

Example 44. One might suspect that the equivalence reported in the previous example holds because of the convexity of systems of linear equalities. To counter this idea, the construction is repeated with h redefined as follows:

$$h = (5x^2y \geq 0) \wedge (10x^2y - 2y^2 \geq 0) \wedge (20x^2y - 6y^2 \geq 0)$$

This again gives $kI = 1[-3, 0]$ and $\vec{p} = \langle 20 + 5i, -6 - 2i\rangle$. Surprisingly, however, the following equivalence holds:

$$[\![\forall i \in \{-3, 0\}.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) = [\![\wedge_{j=1}^{3} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = [\![h]\!]_{\vec{y}}(\vec{b})$$

Again this equivalence can be verified symbolically using an SMT solver. In fact the equivalence can be shown directly since if $5x^2y \geq 0$ and $20x^2y - 6y^2 \geq 0$ hold, then $(2/3)(5x^2y) + (1/3)(20x^2y - 6y^2) = 10x^2y - 2y^2 \geq 0$ holds.

$$\frac{g \to g'}{f[g] \to f[g']}$$

$$\neg(\neg f) \to f$$

$$\bigwedge \wr \S \to \top \qquad \bigvee \wr \S \to \bot$$

$$\bigwedge \wr f \S \to f \qquad \bigvee \wr f \S \to f$$

$$\frac{\bigwedge \wr f_i \S_{i=2}^n \models f_1}{\bigwedge \wr f_i \S_{i=1}^n \to \bigwedge \wr f_i \S_{i=2}^n} \qquad \frac{f_1 \models \bigvee \wr f_i \S_{i=2}^n}{\bigvee \wr f_i \S_{i=1}^n \to \bigvee \wr f_i \S_{i=2}^n}$$

$$\frac{f_1 = \bigwedge \wr g_i \S_{i=1}^m}{\bigwedge \wr f_i \S_{i=1}^n \to \bigwedge (\wr g_i \S_{i=1}^m \cup \wr f_i \S_{i=2}^n)} \qquad \frac{f_1 = \bigvee \wr g_i \S_{i=1}^m}{\bigvee \wr f_i \S_{i=1}^n \to \bigvee (\wr g_i \S_{i=1}^m \cup \wr f_i \S_{i=2}^n)}$$

$$\frac{\bigwedge \wr f_i \S_{i=1}^n \models \bot}{\bigwedge \wr f_i \S_{i=1}^n \to \bot} \qquad \frac{\bigvee \wr f_i \S_{i=1}^n \models \top}{\wr f_i \S_{i=1}^n \to \top}$$

Figure 62: Rewrite rules

These examples suggest a general tactic for collapsing conjunctive formulae, irrespective of whether $A = \{\vec{a}_1, \ldots, \vec{a}_\ell\}$ or $A \subseteq \{\vec{a}_1, \ldots, \vec{a}_\ell\}$: a quantified formula is derived and then entailment checking is applied to see if it can be reduced to only the first and last conjunct.

5.6 Concluding discussion

Thus far, we have assumed that the conjuncts of a conjunctive formula differ only in numeric constants. Suppose, however, that there are two conjuncts whose propositional skeletons differ. If the conjuncts can be rewritten so that their propositional skeletons coincide, so that they only differ in numeric constants, then it becomes possible to derive a quantified formula using the methodology presented previously. This is the rationale behind the rewrite rules give in figure 62. The first rule is expressed with the aid of a hole: $f[g]$ denotes a formula f with a subformula g. Then if g rewrites to g', then $f[g]$ rewrites to $f[g']$. The second rule eliminates double negation. The next four rules handle conjuncts and disjuncts which are either empty or singular. The next two rules catch redundant conjuncts

and redundant disjuncts. The subsequent two rules either lift a conjunct which is itself a conjunction or lift a disjunct which is itself a disjunction. The final two rules detect inconsistent conjuncts and tautologous disjuncts. Note how the multiset formulation of conjunction and disjunction aids in the formulation of these rules. It should be emphasized that these rules are incomplete but nevertheless serve to extend the applicability of the methodology.

The above rewrite rules extend the generality of the method presented in this chapter by rewriting formulae into a form which aids the discovery of repeating patterns of coefficients. The rules for handling redundant conjuncts resemble rewrite systems for tightening systems of constraints, the most notable of which are the rewrite rules for strengthening bounds in difference bound matrices (DBMs), as used when model checking timed automata. In fact, the connection between these model checking algorithms and IMPACT itself is closer that one might initially think. The key step in achieving termination in IMPACT is the discovery of covering relationships between locations, the same concept occurring, even called covering, in the model checking of timed automata [5, page 342]. Since the latter does not deploy widening, one has to ask why? In the case of timed systems, where timing differences are expressed as DBMs, the set of all DBMs which are reachable from an initial state characterized as a DBM, is finite [31, Theorem 4]. This is because all the entries of the DBMs are drawn from a finite fixed set of values, bounding the number of distinct DBMs overall. This property occurs because of the way timers progress together over the lifetime of the model checking problem, so only the differences between timers need to be preserved. In effect, the DBMs perfectly match the model checking problem for timed systems because they discard superfluous information and do so in a way which induces finiteness. This illustrates how certain classes of abstraction are suitable for particular classes of problem. However, not all problems can be modelled with timed automata, hence the need to induce termination through other means, namely widening. Interestingly, Dill recognizes, even in his seminal work [31], that in general there is a need to apply "approximate and heuristic methods [to induce termination]" though widening was barely known at this point [26].

Chapter 6

ImpactExplorer

6.1 Introduction

Model checking algorithms aspire to verify programs in a completely automatic way and therefore there has been little work on animating these algorithms to provide insight into the mechanics of the algorithms to the user. IMPACT is not so much an algorithm but a framework that can be instantiated in many ways, according to the order in which the key operations of expansion, covering and refinement are applied. Yet efficiency is affected and termination can be compromised if these operations are not applied in a judicious order. In particular, if an instance of IMPACT does not terminate, it is not clear why. The subtlety is that there are causal dependencies between the commands. For example, if a node is covered, then it no longer needs to be expanded. The unwinding graph is particularly sensitive to choices made early in its development. It seems surprising therefore that there not been any work on animating IMPACT as this seems key to understanding it and successfully deploying it on challenging applications. To this end, IMPACTEXPLORER has been designed as a visualization tool that can step through a run of the IMPACT framework. This allows the user to study how the unwinding graph grows, as well as providing functionality to control its growth.

This chapter describes the development and structure of IMPACTEXPLORER: first, presenting the rationale for the coding medium; second, overviewing its architecture, including a description of the underlying data structures; third, reviewing

the commands available to a user. The chapter concludes with a discussion on the interplay of mathematical development and software development.

6.2 Language choice

ImpactExplorer is programmed entirely in Scala. The main data structures are realized mainly through Scala classes, concrete and abstract, traits (analogous to an interface in Java) and singleton objects (which group methods that would be static in Java). Scala was chosen for the following reasons:

Declarative nature The polynomial, formulae and unwinding graph transformations and, likewise, the domain operations can all be expressed in code that precisely mirrors their mathematical formulation (stateless), even to the point of using unicode to name functions, linking them with their mathematical counterparts.

Concise code Scala was designed to be less verbose than Java. Its high-level nature allows data structures to be mapped over or filtered with predicates without considering indexes and other lower level details. This makes it easier to validate the correctness of the code through manual inspection. It also reduces the burden of reengineering, which is so common in a research prototype.

The net effect is that ImpactExplorer is just c5,300 lines of code, which includes the significant portion dedicated to visualization and validation.

6.3 Architecture of ImpactExplorer

Package structure Figure 63 is a schematic diagram of the architecture of ImpactExplorer. The main elements of ImpactExplorer are indicated in green, whereas blue denotes the input/output files and external libraries, notably the Z3 SMT solver (maintained by Microsoft). The green bar for the impact package encompasses the subpackages modelling and harness. The modelling subpackage supports data structures and operations on unwinding graphs for analyzing automata. The harness subpackage provides the overarching real-eval-print loop

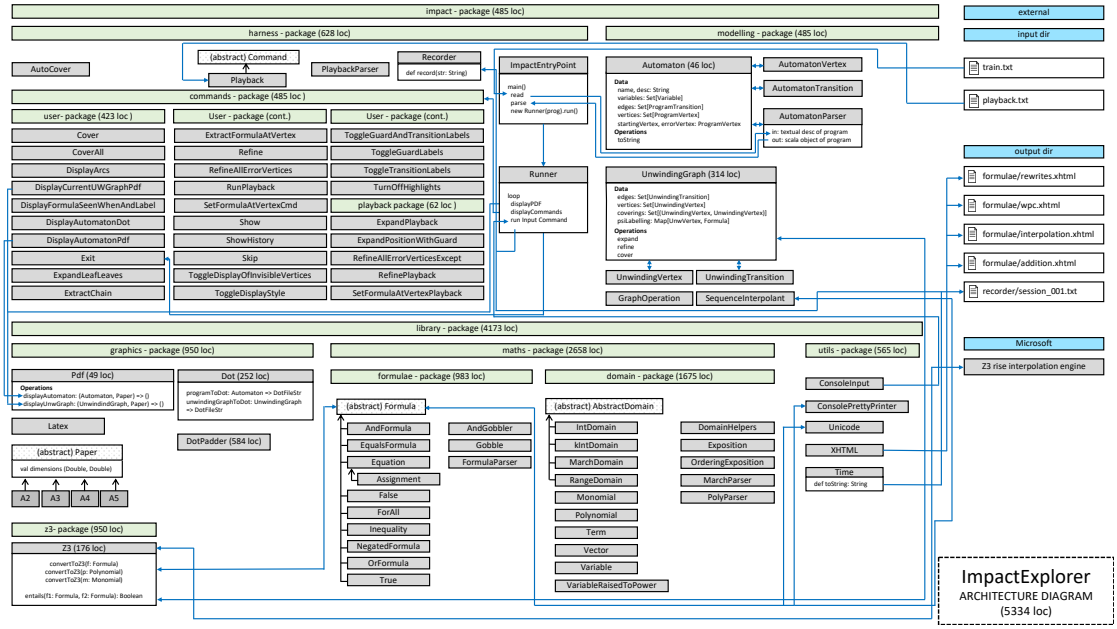


Figure 63: Architecture of IMPACTEXPLORER

functionality which drives the way the unwinding graph develops under IMPACT. The library package encompasses a number of subpackages which provide additional functionality needed for visualization; quantifier introduction; and debugging and validation. These subpackages are respectively named graphics, maths and utils.

Subpackage structure The graphics subpackage renders the unwinding graph. The maths package itself subdivides into a package for manipulating symbolic formulae, named formulae, and the package domain which provides the domain operations for marches. The domain package also provides operations for ranges and intervals that themselves support marches as well as vector arithmetic which underpins all the domain operations. The utils package provides functionality for validating operations on formulae. For example, XHTML generates a table in interpolation.xhtml to aid the validation of the sequence interpolants. Each row in the table corresponds to a sequence interpolant which arises during refinement; each column gives an individual interpolant which makes up the sequence and also provides the formulae from which the interpolant is derived. Likewise wpc.xhtml

records the derivation of each weakest precondition. Similarly, rewrites.xhtml logs the formulae simplification rewrites, as described in figure 62. In a similar manner, addition.xhtml records all polynomial arithmetic. The utils package also contains code for pretty printing when using the console, notably colouring different formulae, as to distinguish them. It also contains a number of unicode symbols which are used within formulae to closely match the normal presentation of formulae.

File structure The grey boxes indicate classes whereas the white boxes with a file icon indicate plain text or xhtml files. Boxes with a grey dotted background distinguish abstract classes from their concrete counterparts. The blue boxes indicate the file structure (actually directory names) used for input/output and for external libraries. The blue arrows indicate information flow. For example, the arrow from the ImpactEntryPoint grey box, emanating from read, to the ProgramParser grey box, indicates a flow of a list of strings that represents the contents of train.txt.

6.3.1 Underlying data structures

6.3.1.1 Data structures of the impact package

AutomatonTransition An automaton is a graph composed of vertices and edges. The vertices are interpreted as locations and the edges model transitions between them. The AutomatonTransition class represents a transition. It is instantiated by providing a name, the two automaton locations, and the guard and action formulae along the edge between them. (The formulae are defined over a predefined set of variables). Edges are named so that the user can identify an edge in the output with an edge given in the input text file. It also stores the unique identifiers for the guard and action formulae. As with the unwinding graphs, these can be used in place of full formulae to give a more compact representation of an automaton.

AutomatonVertex The AutomatonVertex class represents a vertex of an automaton. Each vertex is assigned a name for the purposes of identification.

Automaton The Automaton class contains a name of an automaton and free-form text describing it. The class also contains the set of variables over which the transition formulae is defined, as well as the vertices and edges of the automaton. In addition, it distinguishes two special vertices, the starting vertex and the error vertex.

AutomatonParser The AutomatonParser class is a context-free parser that converts a textual description of an automaton into an Automaton object which takes a role akin to an abstract syntax tree.

Command The Command abstract class represents a command entered either by a user in an interactive REPL session, or one invoked via playback. It has two abstract members, description and action, which prescribe methods to be defined in subclasses. The description method describes what the action the command performs and is displayed to the user when ImpactExplorer lists all its commands. The action method maps an input unwinding graph to an output unwinding graph. Command also has two variables, continue and displayPdf, which prescribe default boolean values. The variable continue determines whether the REPL loop continues after executing a command. The variable displayPdf indicates whether a pdf should be displayed automatically after executing a command. The sole concrete member, apply, applies the action method to the current unwinding graph.

GraphOperation The GraphOperation abstract class merely groups together concrete operations on the unwinding graph, namely Initialization, Expansion, Covering, Refinement and SetFormulaAtVertex.

SequenceInterpolant The SequenceInterpolant class represents a sequence interpolant. It is initialized using two items: a list of UnwindingTransition objects, and a list of Formula objects. The first element of the list of Formula objects is True. The ith Formula object, when conjoined with the guard and update formulae at the ith UnwindingTransition object, logically entails the $(i+1)$th element of the formula list, following the definition of a sequence interpolant [52].

UnwindingGraph The UnwindingGraph class both models the unwinding graph as specified in IMPACT and supports the additional functionality of IMPACTEXPLORER. It is parameterized with three distinct sets of data. First, it takes data that specifies header details such as the automaton under analysis and the graph operation which led to its creation. Second, it has the data structures specified in IMPACT such as sets of vertices, edges, covering arcs and formula labelling. Third, it has flags which prescribe its presentation. This and the other data and operations of the UnwindingGraph class be categorized in three areas:

- *IMPACT methods* The three methods expand, refine and cover implement the three basic unwinding steps as detailed in [52].
- *IMPACTEXPLORER methods* The two methods setFormulaAtVertex and expandPositionWithGuard are methods specific to IMPACTEXPLORER. The coverPairs method returns a list of candidates for covering and is used in an interactive session when the autocover setting is off. This differs from IMPACT where the application of covering is fully prescribed. The expandPositionWithGuard method implements the command of the same name, as detailed in the previous section.
- *Public fields and methods* The public fields and methods other than those mentioned above such as epsilon, leafMappingsGroupedByProgLoc, vertexIsCovered and levelToVertices provide information to external classes. For example, the graphics package uses the levelToVertices map, which lists the vertices located at the various depths of an unwinding graph, to display all sibling vertices at the same level.

UnwindingTransition This class models an edge in an unwinding graph augmented with a transition formula, comprised of a guard and an update, and their short-form identifiers.

UnwindingVertex This abstract class represents a vertex in the unwinding graph. A concrete unwinding vertex is instantiated by providing a unique integer identifier.

6.3.1.2 Data structures of interest in library package

Dot The Dot object contains two main public methods, automatonToDot and unwindingGraphToDot, which generate a dot file from which an automaton and unwinding graph can be rendered respectively.

DotPadder The DotPadder object takes a list of strings that represents the vertices, edges and covering arcs in dot file format of the current unwinding graph. These strings are generated by the Dot object, described above. It then performs set differencing operations against those of a specific reference unwinding graph: one for the vertices, one for the edges and another for the covering arcs. This information is used to create a new unwinding graph where any missing vertices, edges and covering arcs are added, but not displayed, to fix the absolute positions of vertices, edges and arcs in the new graph. This amounts to applying a series of string editing operations on the lines of the dot file. This is so that across a series of graphs, when rendered, the subgraphs do not move around, aiding with subsequent analysis.

Pdf The Pdf object takes an UnwindingGraph object, on which is runs Dot and DotPadder to generate a dot file, and then executes a shell command calling the dot application with the dot file as input. Finally, it opens the pdf for viewing. It performs the same sequence of steps to display the automaton (but without the need to call DotPadder, which is only relevant for unwinding graphs).

6.3.1.3 Data structures of interest in maths package

AbstractDomain The AbstractDomain trait is inherited by the abstract domains used in ImpactExplorer. It declares the abstract methods $\sqsubseteq$, $\sqcup$ and $\sqcap$. Other operations are implemented on top: $a \equiv b$ iff $a \sqsubseteq b \wedge b \sqsubseteq a$ and $a \sqsubset b$ iff $a \sqsubseteq b \wedge \neg(a \equiv b)$.

AndGobbler The AndGobbler trait, mixed in to the AndFormula class, converts a list of conjuncts into a compact representation for human comprehension.

For example, the conjuncts $x + y \geq 5$ and $-y > x - 10$ would be rendered as the single string $5 \leq x + y < 10$.

Assignment The Assignment class extends from the Equation class, which itself is a concrete instance of the Formula abstract class. In an assignment, the left side of an equation is a variable and the right side is a polynomial.

DomainHelpers The DomainHelpers object provides auxiliary methods so that the codebase supports the notation used in the mathematics. The first is γ, which, once in scope, can be used to write $\gamma(d)$ rather than $d.concretize$, where d is an instance of an abstract domain. It also defines a number of implicit methods that extend the functionality of existing classes. For example, $|$ defines the divides method for an integer, the $*$ method allows scalar multiplication against a vector, and the $\cdot$ and $\otimes$ methods allow scalar multiplication against abstract domains. DomainHelpers also extends the functionality of a vector of integers, defining methods such as $|$, divisors, and isIrreducible. The divisors method finds all integers that divide all the components of a vector and isIrreducible is a predicate which is true when its divisors are only ± 1.

Formula The Formula abstract class is the superclass of all the logical formulae used in ImpactExplorer: namely AndFormula, Assignment, EqualsFormula, Equation, False, ForAll, Inequality, NegatedFormula and OrFormula. The abstract method implements a number of common methods that include substitute, primeNtimes and entails. The substitute method substitutes a list of assignments into a formula. This is used when calculating weakest preconditions. The primeNtimes method replaces each variable in the formula x with a primed counterpart x''', when $n = 3$, for instance. This is used to name formulae apart when calculating sequence interpolants. The entails method determines whether the current formula logically entails another. Its companion object contains a rewrite method that performs the formula simplifications as given in figure 62, the wpc method that calculates weakest preconditions and various factory methods for creating formulae. The Formula object caches formula rewrites, as many similar rewrites happen over the course of a session. All formula operations are recorded to their

own output log files for independent validation.

Variable The Variable class represents a variable with a numeric subscript. It also stores a primingLevel integer count, which records how many primes are on the variable.

Vectors The Vectors class supports vector arithmetic. It has a method which finds the common divisors of a set of integers, namely those integers which constitute the elements a vector. It determines whether a vector divides another, and calculates the gcd of two or more vectors, in the sense described in section 2.

6.3.1.4 Data structures of interest in z3 package

Z3 The Z3 object is the main interface between ImpactExplorer and the external Z3 SMT library. Formulae, and the polynomials contained within, are converted to Z3 format by a number of conversion methods. The other main use of the Z3 object is when the Formula class requests entailment checks. Entailment checking is frequent but many of the entailment checks are the same hence memoization is applied to cache entailment checks. It was found to improve performance by three orders of magnitude (when regenerating a session without human interaction).

6.3.1.5 Data structures of interest in utils package

ConsolePrettyPrinter The ConsolePrettyPrinter trait contains methods which print a string in various console colours. This functionality is used by the formulae package to distinguish the brackets and operator in one subformula from the brackets and operator in another at a different level. This aids the comprehension of complicated, structured formulae.

ConsoleInput The ConsoleInput object in the utils package is used by the Runner and the individual command objects to aid interpreting and processing commands. ConsoleInput has two pairs of methods which deal with console interaction with the user. These methods either get a single integer value or a list of integers

Command	Syntax
Cover	`c` cp_i
CoverAll	`c all`
ExpandLeaves	`e` $[v_1 v_2 \ldots v_n]$
ExpandPositionWithGuard	`e` p g `[x` t `]`
Refine	`r` v
RefineAllErrorVertices	`r all`
SetFormulaAtVertex	`set` v f

Figure 64: Commands specific to IMPACT

from the console, representing an index of a command, or multiple commands, to be run from the list of user commands. ConsoleInput also has methods for checking whether the entered integers correspond with a set of valid options, as given by the current unwinding graph object. For example, in the case of expansion, it would check an entered integer is the identifier of an expandable leaf vertex.

CustomShowImplicits The CustomShowImplicits trait, which is available across IMPACTEXPLORER, enables sets, vectors, lists and arrays to be printed to the console in a custom format. The custom format implemented in IMPACTEXPLORER closely matches the mathematical syntax for sets and vectors and the Lisp format for lists.

XHTML The XHTML object reads an output csv file that is generated by an operation, such as polynomial addition, and converts it to an xhtml file. Presentation in xhtml format enables a browser to be used to view large tables.

The following section details the typical flow of a session of analysis:

6.4 Using ImpactExplorer

Figure 64 presents the commands which relate to core IMPACT operations, such as expanding a leaf, adding a covering arc or refining an error location. IMPACTEXPLORER also provides commands for setting a formula at a vertex to a stronger formula (a quantified formula) for the purposes of inducing termination. Another

novel command is for expansion. If a given program location with guard corresponds to a unique uncovered vertex in the unwinding graph, then the vertex is expanded, and if uniqueness again holds, then the expansion is reapplied up to n times in total. Repeated expansion is useful for gaining insight into the growth of the unwinding graph.

The following list contains details on the commands, beginning with the name of the command in boldface, followed by a description of operation the command performs.

- **Cover** Invoking this command brings up a list of coveree-coverer vertex pairs (cp_i), that is, pairs of unwinding vertices whose individual vertices correspond to the same automaton location, where the formula label of the coveree vertex entails the formula label of the coverer vertex. Each coveree-coverer candidate pair is identified with a unique index. The user can then enter an index and a covering arc will be inserted from the coveree to the coverer. Note that ImpactExplorer has a configurable flag, AutoCover, which, when enabled, automatically identifies opportunities for covering and inserts the appropriate covering arc. In this case, a manual invocation of the Cover command would identify no remaining candidates.

- **CoverAll** Rather than insert a user-selected coveree-coverer vertex pair, this command inserts a covering arc whenever possible in an unwinding graph. It does so by ordering covering arc pairs according to the level of a coveree, and then its coverer, in the unwinding tree. It then inserts a covering arc for the highest ranked pair and then reranks to insert the next pair, until no candidates remain. Setting AutoCover to true makes CoverAll redundant.

- **ExpandLeaves** This command takes a list of vertices and expands them in turn, even if, at the time, the latter vertices do not yet exist. This command can be used to recreate an unwinding graph up to a particular step, for the user to experiment with a different sequence of actions. Should, however, there only be one leaf vertex in the unwinding graph, this vertex will be expanded automatically without prompting for a vertex. When used in playback mode, the list of vertices is not displayed to the user since the vertices to be expanded are read from a playback file.

- **ExpandPositionWithGuard** Suppose a vertex, v, corresponds to a particular automaton location, p, with an incoming edge with guard g, is expanded to produce a new vertex, v', also at p. Then an expansion of v' can itself provide another candidate for expansion v'', and so on. The formula at v is then strengthened when v' or v'', or any other child vertex, is refined. Multiple expansions, followed by a refinement, can result in the formula at v developing a repeating structure. This command is used to mine for formulae with repeating structure as these hint at where quantification can be introduced. The command takes an automaton location p and a guard formula g and an optional t entry (which, if not given, defaults to 1) that prescribes the number of expansion steps. It then searches for a unique leaf vertex at p which has an inward edge with guard g. It then expands the leaf, representing a transition along the automaton edge. It does this repeatedly, t times, preparing the graph for refinement.

- **Refine** This command refines an error vertex that, as yet, remains unconsidered. Unrefined error vertices are displayed to the user in a list, and one is chosen by the user for refinement.

- **ExpandPlayback** The command behaves analogously to the **ExpandLeaves** user command but ImpactExplorer calls it when replaying commands from a playback file. Thus there is no need for a user to enter the list of vertices.

- **RefineAll** This command will refine all unrefined error vertices in an unwinding graph.

- **SetFormulaAtVertex** This command accepts, via the console, a vertex followed by a textual representation of a formula. It then strengthens the formula of the given vertex with the prescribed formula. A requirement of the tool is that the formula must be strictly stronger than that already at the vertex: entering the same formula that already holds at a vertex would have no impact, whereas entering a weaker formula would invalidate existing coverings. Notice that the f_1 in the command `set 32` f_1 of figure 33 is actually entered at the command line as:

Command	Syntax
ExtractChain	`chain` $v_1 v_2 \ldots v_n$
ToggleDisplayOfInvisibleVertices	`tog invis`
ToggleDisplayStyle	`tog style`
ToggleGuardAndTransitionLabels	`tog guardtran`
ToggleGuardLabels	`tog guard`
ToggleTransitionLabels	`tog tran`

Figure 65: Commands for visualization

```
(-20 <= b-s && b-s <= 20)
  &&
forall i in [0, inf] .
  ((d < 9-i) => (-21-i <= b-s && b-s <= 19-i))
```

although in actuality `forall` is preceded by an escape character to distinguish it from a variable. When used in playback mode, the vertex and formula are read from a playback file.

6.4.1 Commands for visualization

Figure 65 outlines the commands for visualization. Since an unwinding graph can become large, and even a formula at any vertex can itself be unwieldy, the challenge is to present an abstraction of the unwinding graph that is comprehensible to the user (and the developer). However, even when the number of vertices is large, the number of distinct formulae is often small. ImpactExplorer exploits this to achieve a dense representation of an unwinding graph, by mapping each formula to a unique identifier, which is displayed in a vertex. An auxiliary table then decodes the identifier. This representation results in similarly-sized vertices that makes the layout more uniform and comprehensible to the user. (This style is used in the unwinding graphs in Appendix D. A further reduced form which solely displays a vertex identifier is used in the graphs in Chapter 4.) It also highlights the reoccurrence of common formulae, which can hint at tactics for curtailing the growth of the unwinding graph. However, to study a particular branch of the unwinding graph in detail (rather than study the shape of the unwinding graph

as a whole), it is useful to display the formula in place at each vertex down the branch. A refinement of this form of visualization is to show some of the vertices on a branch rather than all the vertices down on a branch (this is useful for studying how interpolants propagate preconditions). This reduces the cognitive load on the user.

- **ExtractChain** This command accepts a list of vertices and produces a pdf document of the current unwinding graph, but only showing the vertices present in the list, hiding all others along with their associated covering arcs. Importantly, the vertices assume the same position in the rendered unwinding graph even though only the vertices prescribed in the list are displayed. This command enables the user to focus on a particular branch of the unwinding graph, again reducing cognitive load.

- **ToggleDisplayOfInvisibleVertices** In earlier iterations of ImpactExplorer, there were unnecessary self-loop covering arcs, since the tool was strictly following McMillan's algorithm in [52]. This algorithm, however, was simplified for presentational purposes, and although not wrong, introduces redundant self-loops which obscures the unwinding graph. It was therefore useful to hide these self-loops and covering arcs which did not convey useful information.

- **ToggleDisplayStyle** This command allows the user to switch between different display styles of the vertices. One presentation style of the vertices displays the vertex id (a unique integer identifier which increases incrementally as unwinding vertices are created) followed by the formula label, followed by the program location, each on its own line. An alternative style shows the unwinding label as the vertex id followed by the formula on one line, separated by a colon, followed by the program location on the next. The first display style is more natural but the second produces diagrams which are narrower and can better match the proportions of a page.

- **ToggleGuardLabels** This command will reduce the guard labels on the edges to unique identifiers, which can be decoded as a formula using an auxiliary table, rather than displaying them in full. For smaller unwinding

Command	Syntax
RunPlayback	`playback`
ShowHistory	`hist`

Figure 66: Commands for playback

graphs, the user would want to have the guard labels displayed in full. However, for large graphs it is more beneficial to display the identifiers instead of the formulae themselves. This keeps the node size uniform, making it easier to understand the structure of the graph. The commands **ToggleTransitionLabels** and **ToggleGuardAndTransitionLabels** work analogously, introducing identifiers to represent the formulae on transitions.

6.4.2 Commands for playback

Figure 66 shows the commands used for quickly reproducing a previous run of ImpactExplorer. Each time ImpactExplorer is used, the commands that make up a run are logged and stored in a persistent database (complete with configuration information which controls how the unwinding graph is displayed). When requested, the logged commands are displayed to the user. This is useful when the user wishes to resume a sequence of unwinding operations after trying an alternative expansion strategy.

- **RunPlayback** This command runs commands read from the playback file `./input/playback.txt`. Every command of a session is recorded in an individually time-stamped text file which is indexed and stored in a database. To recreate a session, the user would copy the text file over the playback file and then invoke this command. The playback file can be edited down to derive an earlier unwinding graph.

- **ShowHistory** This commands prints to the console the commands the user has entered during the current session.

Command	Syntax
DisplayArcs	`arcs`
DisplayCurrentUWGraphPdf	`d`
DisplayFormulaSeenWhenAndLabel	`when`
DisplayAutomatonDot	`automatondot`
DisplayAutomatonPdf	`automatonpdf`
ExtractFormulaAtVertex	`extract` v
Show	`show`

Figure 67: Commands for displaying the unwinding graph

6.4.3 Commands for displaying the unwinding graph

Figure 67 details the commands for displaying the automaton under consideration, along with its unwinding graph. The unwinding graph can itself be presented in dot format, in pdf format and in a text format, rendered by the toString method associated with the UnwindingGraph class. The dot format is a widely used textual format used to produce directed graphs using the dot renderer [2]. These display commands are orientated towards debugging and testing, enabling the developer to check the structure of the unwinding graph and validate the formulae that label the nodes.

- **DisplayArcs** This command prints out all cumulative covering arcs in dot format to the console, that is, any covering arc that has appeared in any unwinding graph up until the current graph. This is used in conjunction with the DotPadder object to produce animations of unwinding graphs whose placement of vertices does not change over time. Without this facility, dot could display two successive unwinding graphs, where one subgraph is to the left of another subgraph in the first, and to its right in the second. This obscures the development of the unwinding graph.

- **DisplayCurrentUWGraphPdf** This command displays the current unwinding graph. Note an unwinding graph is normally displayed immediately after a command is executed. However, after many commands have been executed, each generating its own pdf, it is sometimes useful to regenerate the current unwinding graph, without explicitly selecting it.

- **DisplayFormulaSeenWhenAndLabel** The command displays a table that maps an identifier to a formula, similar to figure 21. The table also identifies the unwinding graph where a formula first appears.
- **DisplayAutomatonDot** This command prints the automaton as a dot file to the console. This could be used to check the structure of a rendered pdf of an automaton.
- **DisplayAutomatonPdf** This command generates and opens a pdf of the automaton under analysis.
- **ExtractFormulaAtVertex** The command prints the formula holding at a vertex to the console. The structure of the formula could then be inspected using a text editor, or equally it could be converted to LaTeX using the Latex.toLatex method.
- **Show** This command prints the unwinding graph as a Scala object to the console, via the println method of the unwinding graph class. This is a convenient way to display the contents of a simple unwinding graph when running the program outside of an IDE environment, without a debugger.

6.5 High level usage of ImpactExplorer

ImpactExplorer employs a REPL interface for interacting with a user. It responds to commands either by presenting graphical output in pdf format, via popup screens, or by writing textual output to the console. Pdf popups are used to show visual data such as the control flow graph of the program under analysis or the current step of the unwinding graph. Console output is used to output textual data, and can be used to extract, for example, the dot file of the current unwinding graph, which is necessary for including the file in another document. The REPL commands are listed at the console after each command is run, are numbered and are grouped by function. Commands also have a short name which can be used for ease. Although the commands are grouped by function, the user can select any command in any order. Therefore guidance is given below as to how best use the tool when performing an analysis of a program:

6.5.1 Eager application of covering and error refinement

A general tactic is to apply covering as eagerly as possible to prevent the development of a vertex which otherwise could be covered. This saves the development of a subtree rooted at that vertex which has already been considered elsewhere. If covering cannot be applied then any error vertices need to be refined to rule out the possibility of reaching the error. Again, this should be performed eagerly as any propagated conditions may invalidate any existing covering relationship higher in the unwinding graph.

6.5.2 Eager expansion of loops

Repeated unwinding of a loop in an automaton can result in conjuncts being added to the formulae along the path that grow in a repetitive way. One general tactic is to prioritize the expansion of vertices within the loop and, while doing so, not develop vertices elsewhere. The rationale behind this tactic is to highlight specific patterns of formula growth without obscuring these patterns with noise (superfluous requirements) stemming from the development of vertices outside the cycle. This is useful for exposing the structure of the formulae to indicate where widening is applicable.

6.5.3 Repeated formulae analysis

Even in a large unwinding graph there tends to be only a few distinct formulae. Therefore it is helpful to factor out repeated formulae to reduce cognitive load on the user when searching the unwinding graph for formulae growth. To this end, ImpactExplorer displays a table with each unwinding graph that highlights the unwinding graph in which the formulae first appeared. This relates the first occurrence of a formula to a particular state in the automaton. Thus when the state is reached again with another formula, the different formulae can be more easily correlated to suggest widening.

6.5.4 Reversible unwinding

Even with these tactics it is possible to make a false choice when developing the unwinding graph. ImpactExplorer therefore provides the facility to restore the development of an unwinding graph from an earlier point. Although not a tactic in itself, this facility is as important as a tactic in accelerating the development of an unwinding to completion.

Although ImpactExplorer allows the user complete freedom (which is important for understanding the Impact algorithm as a whole), the above tactics reduce the degree of choice that a user is confronted with when actually applying the tool, which is key to its successful deployment.

6.6 Discussion

This chapter grew out of an attempt to implement what one might consider straight-forward domain operations. Coding the domains operations in Scala revealed ambiguities and edge cases which drove the refinement of these domains and the construction of their associated proofs. In tandem with formalizing these domains and operations, the accompanying Scala code enabled various examples to be explored without the dangers associated with hand-cranking. This accelerated the development of the mathematical concepts. The net result is a rigorously defined and proven suite of domain operations which come complete with a Scala implementation, where there is almost a one-to-one link between the definitions and the code. (The only discrepancies arise in case matching where the Scala compiler cannot always detect when the case analysis is total.) To summarize, coding the domain operations was key to their mathematical formalization and the code itself mirrors the formulation down to the very symbols used in the code.

A natural question to ask is the extent to which the application of widening in ImpactExplorer can be fully automated. It is worth remembering that ImpactExplorer arose in order to diagnose the non-termination of the Impact algorithm when it was used as a black box, that is, in a fully automatic way. The methodology presented in Chapters 4 and 5 shows how to apply widening to reduce the build up of conjunctive formulae but stops short of diagnosing when it

should be applied. One could conceive of a tool which tracked the development of conjunctive formulae looking for repeating patterns of coefficients and alerted a user to the possibility of widening. Then the user could make the choice as to whether to deploy widening at that moment or to hold off to a more propitious time (which is indeed a tactic in so-called delayed widening [9, section 7.1.3]). If widening with thresholds [9, section 7.1.2]) was used to improve interval approximation, then the derivation of the thresholds could be automated by scanning the guards and transitions in the automaton for candidate constants (thresholds). In conclusion, it seems likely that the usability of ImpactExplorer can indeed be improved through automation, though full automation is undesirable as then Impact reduces again to an inscrutable black box.

Chapter 7

Future Work and Conclusions

This thesis arose from an implementation study of IMPACT that exposed the need for the introduction of quantified formulae. The solution was inspired by techniques in abstract interpretation, namely widening, where the abstract domain does not satisfy the ascending chain condition. These domains are effectively infinite-state and therefore it is not surprising that these techniques resurface in infinite-state model checking. Future work will focus on this interplay and the mechanics of how to realize this approach in IMPACTEXPLORER.

7.1 Future work

IMPACTEXPLORER is a tool that has been specifically designed to visualize unwinding graphs and explore different tactics in their development. It was critical in the discovery of the need to introduce quantified formulae and to show that, even with widening, quantified formulae could discharge complicated verification conditions. Thus far, IMPACTEXPLORER can only be used in an interactive mode where the user provides the quantified formulae at appropriate vertices. Although these formulae are mechanically derived using the March^n domain, IMPACTEXPLORER does not currently identify those vertices where quantified formulae should be introduced: this is currently the responsibility of the user. The next step towards automation is the automatic selection of these vertices and automatic introduction of quantified formulae. Only then can tool can be easily employed across a large

class of automata. The challenge is to introduce automation and yet still retain the ability to trace the development of the unwinding graph, since the latter is necessary to diagnose the reason for non-termination, should this occur. In this case, even with widening, IMPACT would still be a semi-algorithm. Indeed, it would be interesting to discover an automaton where the numerical constants do not differ by a fixed stride which would not be amenable to the approach proposed in this thesis.

IMPACTEXPLORER has been designed as a research prototype and some consideration is needed how to minimize data and command entry, which takes focus away from understanding the evolving unwinding graph. Since in the short term the focus is on semi-automation rather than full-automation, it is also important that formulae are made comprehensible to the user. Much effort has already been expended on this but it would be useful to allow the user to define tailored rewriting tactics for deriving compact, natural formulae that are specific to the automaton under study.

Marches were devised as higher-dimension abstractions as loops are not necessarily controlled by single counters: one counter might need defined in terms of another; used, for example, to access an array with constant offset from the controlling loop counter. However, loops can also be nested which would suggest that counters take values in two distinct arithmetic progressions, possibly with the start point of the inner arithmetic progression defined as a point in the outer arithmetic progression. It is not clear how best to summarize these points and combine them with higher-dimensional bounded-box abstractions which define the range of a quantified formula.

IMPACTEXPLORER is the sum of many parts: the numeric domains that are used to summarize sets of vectors, formulae representation and manipulation machinery, interpolation and refinement code, and of course the underlying SMT solver, whose code base is not small. This begs the question of whether the unwinding graphs generated by IMPACTEXPLORER can be trusted at all? Although mechanizing a proof for the correctness of IMPACTEXPLORER would constitute a thesis in itself, mechanizing the validation of a particular unwinding graph does seem feasible. The advent of verified decision procedures for linear inequalities [10] provides a verified entailment checker, hence a covering checker, and, in general,

a way to validate an unwinding graph against a given automaton using a proof assistant. Thus the generation of the unwinding graph does not need to be validated, just the unwinding graph itself, which is a small step beyond a validated SMT solver.

7.2 Conclusions

This thesis has shown how widening can be applied to conjunctive formulae to obtain a stable covering relation. This method has been demonstrated on a case study inspired by a train controller. Without such an intervention, the method runs until host memory is depleted, and the user is none the wiser as to whether the automaton is safe or unsafe. Conjunctive formulae arise in the analysis of loops, which are ubiquitous, hence the widening technique, based on relaxing bounds of a quantified variable, is likely to find application beyond the case study. The concept of widening is well understood in abstract interpretation and, in hindsight, seems a natural candidate for improving the termination behaviour of IMPACT.

Although natural, the details of realizing this idea are not straightforward, but multi-stage. To apply widening, it is necessary to rewrite a conjunctive formula so as to expose structural similarity in the conjuncts, up to the occurrence of numeric constants. A finite set of vectors is then defined by the constants occurring in each of the conjuncts. This set-based representation is then exchanged for a symbolic one by applying the join operator of a bespoke abstract domain. This synthesizes a representation based on an integral range and a stride (vector). The domain is itself scaffolded on a hierarchy of simpler domains. The symbolic representation of the vectors then lifts to the conjuncts to derive a quantified formula that compactly summarizes all conjuncts. The formula is quantified using a variable which ranges over the interval where the interval provides a structure for widening the formula. It is curious to note that widening an interval in this setting strengthens a formula, rather than relaxing it, hence the tactic is more akin to narrowing than widening. This whole construction is justified by the staged evaluation of polynomials and formulae, the latter building on the former. When a vertex occurs in an unwinding graph whose formula is constructed from similar conjuncts, the vertex can then be updated with a stronger quantified formula. The case study shows how this can

be sufficient to stabilize the covering relation and thereby induce termination of the IMPACT algorithm.

IMPACTEXPLORER, the software artefact of this thesis, has been instrumental in discovering the locations in an unwinding graph where formulae may be widened to permit the addition of permanent covering arcs. However, this exploration still requires the interaction and insight of a user. However, augmenting the tool to discover points at which widening could be gainfully applied would assist the user by reducing cognitive load, as well as finding earlier opportunities for widening, which may otherwise be missed, thereby precluding unnecessary search. IMPACTEXPLORER has also allowed the author to better understand the IMPACT algorithm and, in particular, to expose how termination is compromised by the unbounded growth of conjunctive formulae.

To conclude, this thesis makes contributions to infinite-state model checking by providing a proof-of-concept demonstrator for extending the IMPACT algorithm with widening.

Bibliography

[1] Albarghouthi, A., Gurfinkel, A. and Chechik, M. (2012). Whale: An Interpolation-Based Algorithm for Inter-procedural Verification. In V. Kuncak and A. Rybalchenko, eds., *Verification, Model Checking, and Abstract Interpretation*, *lncs*, vol. 7148, springer, pp. 39–55.

[2] AT&T Labs (2022). Graphviz. `https://graphviz.org/doc/info/lang.html` accessed: 2022-09-04.

[3] Balakrishnan, G. et al. (2008). WYSINWYX: What You See Is Not What You eXecute. In *Verified Software: Theories, Tools, Experiments*, *Lecture Notes in Computer Science*, vol. 4171, Springer, pp. 202–213.

[4] Bardin, S. et al. (2003). *FAST – Fast Acceleration of Symbolic Transition Systems*. Laboratoire Spécification et Vérification.

[5] Behrmann, G. et al. (1999). Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In N. Halbwachs and D. Peled, eds., *Computer Aided Verification*, Springer, pp. 341–353.

[6] Beyer, D. and Wendler, P. (2007). CPAchecker: The Configurable Software-Verification Platform. `https://cpachecker.sosy-lab.org/` accessed: 2022-12-18.

[7] Biere, A. et al. (1999). Symbolic Model Checking Using SAT Procedures instead of BDDs. In M. J. Irwin, ed., *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*, ACM Press, pp. 317–320.

[8] Biere, A. et al. (2003). Bounded Model Checking. *Adv Comput*, 58, pp. 117–148.

[9] Blanchet, B. et al. (2003). A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, ACM Press, pp. 196–207.

[10] Bottesch, R. et al. (2020). Verifying a Solver for Linear Mixed Integer Arithmetic in Isabelle/HOL. In R. Lee, S. Jha and A. Mavridou, eds., *NASA Formal Methods Symposium*, *Lecture Notes in Computer Science*, vol. 12229, Springer, pp. 233–250.

[11] Brillout, A. et al. (2010). An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In *International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science*, vol. 6173, Springer, pp. 384–399.

[12] Bryant, R., German, S. and Velev, M. (2001). Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. *ACM Transactions on Computational Logic*, 2(1), pp. 93–134.

[13] Burch, J. R. et al. (1990). Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, IEEE Computer Society, pp. 428–439.

[14] Burstall, R. M. (1974). Program Proving as Hand Simulation with a Little Induction. In J. L. Rosenfeld, ed., *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, North-Holland, pp. 308–312.

[15] Cadar, C., Dunbar, D. and Engler, D. R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In R. Draves and R. van Renesse, eds., *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, USENIX Association, pp. 209–224.

[16] Cadar, C. et al. (2006). EXE: Automatically Generating Inputs of Death. In A. Juels, R. N. Wright and S. D. C. di Vimercati, eds., *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, ACM, pp. 322–335.

[17] Carré, B. and Garnsworthy, J. (1990). SPARK—An Annotated Ada Subset for Safety-Critical Programming. In *Proceedings of the Conference on TRI-ADA '90*, ACM Press, TRI-Ada '90, pp. 392—402.

[18] Cha, S. K. et al. (2012). Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, IEEE Computer Society, pp. 380–394.

[19] Christ, J., Hoenicke, J. and Nutz, A. (2013). Proof Tree Preserving Interpolationchecking. In N. Piterman and S. A. Smolka, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science*, vol. 7795, Springer, pp. 124–138.

[20] Clarke, E., Kroening, D. and Lerda, F. (2004). A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, eds., *Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Heidelberg: Springer, pp. 168–176.

[21] Clarke, E. M. and Emerson, E. A. (1981). Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, Berlin, Heidelberg: Springer, p. 52–71.

[22] Clarke, E. M. et al. (2001). Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst Des*, 19(1), pp. 7–34.

[23] Codish, M. et al. (1995). Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1).

[24] Cousot, P. (2015). Abstracting Induction by Extrapolation and Interpolation. In *Verification, Model Checking, and Abstract Interpretation*, *Lecture Notes in Computer Science*, vol. 8931, Springer, pp. 19–42.

[25] Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages*, ACM Press, pp. 238–252.

[26] Cousot, P. and Cousot, R. (1992). Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, Springer, pp. 269–295.

[27] Cousot, P. and N, H. (1978). Automatic Discovery of Linear Restraints among Variables of a Program. In *Principles of Programming Languages*, ACM Press, p. 84–96.

[28] Cousot, P. et al. (2006). Combination of Abstractions in the ASTRÉE Static Analyzer. Springer, pp. 272–300.

[29] Craig, W. (1957). Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic*, 22, pp. 269–285.

[30] Davis, M., Logemann, G. and Loveland, D. (1962). A Machine Program for Theorem-Proving. *Communications of the ACM*, 5(7), pp. 394–397.

[31] Dill, D. L. (1989). Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, ed., *Automatic Verification Methods for Finite State Systems*, *Lecture Notes in Computer Science*, vol. 407, Springer, pp. 197–212.

[32] Emerson, E. A. and Clarke, E. M. (1980). Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, Berlin, Heidelberg: Springer, pp. 169–181.

[33] Floyd, R. W. (1967). Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19, pp. 19–32.

[34] Fuchs, A. et al. (2009). Ground Interpolation for the Theory of Equality. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Lecture Notes in Computer Science, pp. 413–427.

[35] Ganesh, V. and Dill, D. (2007). A Decision Procedure for Bit-vectors and Arrays. Springer, no. 4590 in Lecture Notes in Computer Science, pp. 519–531.

[36] Granger, P. (1991). Static Analysis of Linear Congruence Equalities Among Variables of a Program. In *Theory and Practice of Software*, *LNCS*, vol. 493, Springer, pp. 169–192.

[37] Harrison, W. (1977). Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, 3(3), pp. 243–250.

[38] Henzinger, T. A. et al. (2002). Lazy Abstraction. In *Principles of Programming Languages*, pp. 58–70.

[39] Henzinger, T. A. et al. (2004). Abstractions from Proofs. In *Symposium on Principles of Programming Languages*, ACM, pp. 232–244.

[40] Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Commun ACM*, 12(10), pp. 576—580.

[41] Hsu, I.-C. and Bentley, H. (1971). Pseudo-lattices: Theory and applications. *Arkiv för Matematik*, 8(3), pp. 259–270.

[42] Jain, H., Clarke, E. M. and Grumberg, O. (2008). Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In *Computer Aided Verification*, *Lecture Notes in Computer Science*, vol. 5123, Springer, pp. 254–267.

[43] Kapur, D., Majumdar, R. and Zarba, C. G. (2006). Interpolation for Data Structures. In *Foundations of Software Engineering*, pp. 105–116.

[44] Kozen, D. (1983). Results on the propositional mu-calculus. *Theoretical Computer Science*, 27(3), pp. 333–354, special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982.

[45] Kroening, D. and Strichman, O. (2016). *Decision Procedures*. Springer.

[46] Kröger, F. (1977). LAR: A Logic of Algorithmic Reasoning. *Acta Informatica*, 8(3), pp. 243–266.

[47] Lamport, L. (1980). "Sometime" is Sometimes "Not Never" - On the Temporal Logic of Programs. In P. W. Abrahams, R. J. Lipton and S. R. Bourne, eds., *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, ACM Press, pp. 174–185.

[48] Marques-Silva, J. P., Lynce, I. and Malik, S. (2009). Conflict-Driven Clause Learning SAT Solvers. In A. Biere, M. Heule, H. Van Maaren and T. Walsh, eds., *Handbook of Satisfiability*, IOS Press, pp. 131–153.

[49] Marques-Silva, J. P. and Sakallah, K. A. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5), pp. 506–521.

[50] McMillan, K. (2005). An Interpolating Theorem Prover. *Theoretical Computer Science*, 345(1), pp. 101–121.

[51] McMillan, K. L. (2003). Interpolation and SAT-Based Model Checking. In W. A. Hunt and F. Somenzi, eds., *Computer Aided Verification*, Berlin, Heidelberg: Springer, pp. 1–13.

[52] McMillan, K. L. (2006). Lazy Abstraction with Interpolants. In *Computer Aided Verification*, *Lecture Notes in Computer Science*, vol. 4144, Springer, pp. 123–136.

[53] McMillan, K. L. (2011). Widening and interpolation. In *Static Analysis Symposium*, *Lecture Notes in Computer Science*, vol. 6887, Springer, p. 1.

[54] Miné, A. (2006). The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1), pp. 31–100.

[55] Moskewicz, M. W. et al. (2001). Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pp. 530–535.

[56] Nieuwenhuis, R. and Oliveras, A. (2005). DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. Springer, pp. 321–334.

[57] Nieuwenhuis, R., Oliveras, A. and Tinelli, C. (2006). Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J ACM*, 53(6), pp. 937–977.

[58] Oh, H. (2015). *COSE312: Compilers. Lecture 18 – Interval Analysis.*

[59] Park, D. (1974). Finiteness is mu-ineffable, unpublished D.M.R. Park, Finiteness is Mu-Ineffable, Theoretical Computer Science 3, pp. 173-181 (1976).

[60] Pnueli, A. (1977). The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57.

[61] Simon, A. and King, A. (2006). Widening Polyhedra with Landmarks. In N. Kobayashi, ed., *Asian Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science*, vol. 4279, Springer, pp. 166–182.

[62] Verdoolaege, S., Janssens, G. and Bruynooghe, M. (2009). Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. In *Computer Aided Verification*, *Lecture Notes in Computer Science*, vol. 5643, Springer, pp. 599–613.

[63] Zhang, L. and Malik, S. (2002). The Quest for Efficient Boolean Satisfiability Solvers. In *International Conference on Computer Aided Verification*, Springer, pp. 17–36.

Appendix A

Proofs

A.1 Proofs

for proposition 1. Let $x, y, z \in \mathbb{Z}$.

- Observe $x|x$, hence $x \sqsubseteq x$.
- Suppose $x \sqsubseteq y$ and $y \sqsubseteq z$. Then $x|y$ and $y|z$. Thus $\lambda x = y$ for some $\lambda \in \mathbb{Z}$ and $\mu y = z$ for some $\mu \in \mathbb{Z}$. Then $\mu\lambda x = z$. Since $\mu\lambda \in \mathbb{Z}$, hence $x|z$ and $x \sqsubseteq z$.
- Suppose $m \in \mathbb{Z}$ be a least common multiple of x and y. Observe $x|m$ and $y|m$. Thus $x \sqsubseteq m$ and $y \sqsubseteq m$. Suppose too $x \sqsubseteq z$ and $y \sqsubseteq z$. But $x|z$ and $y|z$. So by definition 4, $m|z$. Hence $m \sqsubseteq z$.

□

for lemma 1. Suppose $a|b$, $a|c$ and $k = gcd(b, c)$. Then $b = \lambda a$ and $c = \mu a$ for some $\lambda, \mu \in \mathbb{Z}$. By Bézout's identity, $k = mb + nc$ for some $m, n \in \mathbb{Z}$. By substitution, $k = m(\lambda a) + n(\mu a) = a(m\lambda + n\mu)$, hence $a|k$. □

for lemma 2.

- Suppose $\vec{d}|\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$. But $gcd(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n)|\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$. Therefore $\vec{d}|gcd(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n)$.

- Suppose $\vec{d}|gcd(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n)$. But $gcd(\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n)|\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$. Hence $\vec{d}|\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$.

□

for corollary 1. By lemma 2, $\vec{d}|gcd(\vec{a}, \vec{b}, \vec{c})$ iff $\vec{d}|\vec{a}, \vec{b}, \vec{c}$ iff $\vec{d}|gcd(\vec{a}, \vec{b}), \vec{c}$ iff $\vec{d}|gcd(gcd(\vec{a}, \vec{b}), \vec{c})$. But $gcd(\vec{a}, \vec{b}, \vec{c})|gcd(\vec{a}, \vec{b}, \vec{c})$, hence $gcd(\vec{a}, \vec{b}, \vec{c})|gcd(gcd(\vec{a}, \vec{b}), \vec{c})$. Conversely, $gcd(gcd(\vec{a}, \vec{b}), \vec{c})|gcd(gcd(\vec{a}, \vec{b}), \vec{c})$, hence $gcd(gcd(\vec{a}, \vec{b}), \vec{c})|gcd(\vec{a}, \vec{b}, \vec{c})$. The result follows. □

for proposition 2.

- Suppose $\vec{x} = \vec{v} = \vec{y} = \vec{0}$. Then $gcd(\vec{x}, \vec{v}) = \vec{0} \neq \bot$. Likewise $gcd(\vec{y}, \vec{v}) = \vec{0} \neq \bot$. Thus $gcd(\vec{x}, \vec{y}, \vec{v}) = \vec{0} \neq \bot$.

- Suppose $\vec{x} = \vec{v} = \vec{0}$ and $\vec{y} \neq \vec{0}$. Then $gcd(\vec{x}, \vec{v}) = \vec{0} \neq \bot$. But $\text{gcd}(\vec{y}, \vec{v}) = \vec{y} \neq \bot$. Thus $gcd(\vec{x}, \vec{y}, \vec{v}) = \vec{y} \neq \bot$.

- Suppose $\vec{y} = \vec{v} = \vec{0}$ and $\vec{x} \neq \vec{0}$. This case is analogous to the previous case.

- Suppose otherwise. Let $\vec{g}_1 = gcd(\vec{x}, \vec{v})$ and $\vec{g}_2 = gcd(\vec{y}, \vec{v})$. Since $\vec{g}_1|\vec{v}$, then $\vec{g}_1 = \vec{v}/\lambda$ for some $\lambda \in \mathbb{Z} \setminus \{0\}$. Similarly, $\vec{g}_2 = \vec{v}/\mu$ for some $\mu \in \mathbb{Z} \setminus \{0\}$. Observe $\lambda = \prod_{\ell=1}^{m} p_\ell^{\alpha_\ell}$, $\mu = \prod_{\ell=1}^{m} p_\ell^{\beta_\ell}$ for distinct primes $p_1, \ldots, p_m$ and $\alpha_\ell, \beta_\ell \in \mathbb{Z}$ such that $\alpha_\ell \geq 0$ and $\beta_\ell \geq 0$. Hence $lcm(\lambda, \mu) = \prod_{\ell=1}^{m} p_\ell^{\max(\alpha_\ell, \beta_\ell)}$. Thus $(\vec{v}/lcm(\lambda, \mu))|(\vec{v}/\lambda)$. But $(\vec{v}/\lambda) = \vec{g}_1$ and $\vec{g}_1|\vec{x}$, so $(\vec{v}/lcm(\lambda, \mu))|\vec{x}$. Similarly, $(\vec{v}/lcm(\lambda, \mu))|\vec{y}$. Moreover $(\vec{v}/lcm(\lambda, \mu))|\vec{v}$. Hence $gcd(\vec{x}, \vec{y}, \vec{v}) \neq \bot$.

□

for proposition 3. To show $[\ell_1, u_1] \sqcap [\ell_2, u_2]$ is a greatest lower bound.

- To show $[\ell_1, u_1] \sqcap [\ell_2, u_2] \sqsubseteq [\ell_1, u_1]$.
  - Suppose $[\ell_1, u_1] \sqcap [\ell_2, u_2] = \bot$. Then $\bot \sqsubseteq [\ell_1, u_1]$.
  - Suppose $[\ell_1, u_1] \sqcap [\ell_2, u_2] = [\ell, u]$. Observe $\ell_1 \leq max(\ell_1, \ell_2) = \ell$ and $u = min(u_1, u_2) \leq u_1$. Hence $[\ell, u] \sqsubseteq [\ell_1, u_1]$.

- To show $[\ell_1, u_1] \sqcap [\ell_2, u_2] \sqsubseteq [\ell_2, u_2]$ is analogous to above.

- Let $I \sqsubseteq [\ell_1, u_1]$ and $I \sqsubseteq [\ell_2, u_2]$. To show $I \sqsubseteq [\ell_1, u_1] \sqcap [\ell_2, u_2]$.
  - Suppose $I = \bot$. Then $I \sqsubseteq [\ell_1, u_1] \sqcap [\ell_2, u_2]$.
  - Suppose $I = [\ell, u]$. Because, $\ell_1 \leq \ell$ and $\ell_2 \leq \ell$, $max(\ell_1, \ell_2) \leq \ell$. Likewise $u \leq min(u_1, u_2)$. Also since $\ell \leq u$, $max(\ell_1, \ell_2) \leq \ell \leq u \leq min(u_1, u_2)$.

An analogous (but simpler) argument is used to show that $[\ell_1, u_1] \sqcup [\ell_2, u_2]$ is a least upper bound.

□

for lemma 3.

- To show $I_1 \sqsubseteq I_2$ implies $\gamma(I_1) \subseteq \gamma(I_2)$.
  - Suppose $I_1 = \bot, I_2 \in \mathsf{Int}$, and so $I_1 \sqsubseteq I_2$. Hence $\gamma(I_1) = \emptyset \subseteq \gamma(I_2)$.
  - Suppose $I_1 = [\ell_1, u_1], I_2 = [\ell_2, u_2]$ and $I_1 \sqsubseteq I_2$. Hence $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$. Let $y \in \gamma(I_1) = \{x \in \mathbb{Z} \mid \ell_1 \leq x \leq u_1\}$. Hence $\ell_2 \leq \ell_1 \leq y \leq u_1 \leq u_2$. So $y \in \gamma(I_2) = \{x \in \mathbb{Z} \mid \ell_2 \leq x \leq u_2\}$.
- To show $\gamma(I_1) \subseteq \gamma(I_2)$ implies $I_1 \sqsubseteq I_2$.
  - Suppose $I_1 = \bot$. Therefore $I_1 \sqsubseteq I_2$ follows immediately.
  - Suppose $I_1 = [\ell_1, u_1], I_2 = [\ell_2, u_2]$. Then $\{x \in \mathbb{Z} \mid \ell_1 \leq x \leq u_1\} = \gamma(I_1) \subseteq \gamma(I_2) = \{x \in \mathbb{Z} \mid \ell_2 \leq x \leq u_2\}$. Therefore $\ell_2 \leq \ell_1 \leq u_1 \leq u_2$. Hence $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$, which implies $I_1 \sqsubseteq I_2$.

□

for lemma 4.

- Let $I_1 = \bot$ and $I_1 \sqsubseteq I_2$. Then $a \cdot I_1 = \bot \sqsubseteq a \cdot I_2$, since $a \cdot I_2 \in \mathsf{Int}$.
- Let $I_1 = [\ell_1, u_1]$, $I_2 = [\ell_2, u_2]$ and $I_1 \sqsubseteq I_2$. Now $I_1 \sqsubseteq I_2$ implies $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$.

  - Suppose $a \leq 0$. Therefore $a \cdot I_1 = [au_1, a\ell_1]$ and $a \cdot I_2 = [au_2, a\ell_2]$. To show $a \cdot I_1 \sqsubseteq a \cdot I_2$, it suffices to show $au_2 \leq au_1$ and $a\ell_1 \leq a\ell_2$. Since $a \leq 0$, $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$, it follows $au_2 \leq au_1$ and $a\ell_1 \leq a\ell_2$. Therefore $a \cdot I_1 \sqsubseteq a \cdot I_2$.
  - Suppose $a > 0$. Therefore $a \cdot I_1 = [a\ell_1, au_1]$ and $a \cdot I_2 = [a\ell_2, au_2]$. To show $a \cdot I_1 \sqsubseteq a \cdot I_2$, it suffices to show $a\ell_2 \leq a\ell_1$ and $au_1 \leq au_2$. Since $a > 0$, $\ell_2 \leq \ell_1$ and $u_1 \leq u_2$, it follows $a\ell_2 \leq a\ell_1$ and $au_1 \leq au_2$. Therefore $a \cdot I_1 \sqsubseteq a \cdot I_2$.

□

for lemma 5.

- Let $I = \bot$. Then $a \cdot (b \cdot I) = a \cdot \bot = \bot = (ab) \cdot I = b \cdot (a \cdot I)$.
- Let $I = [\ell, u]$.
  - Suppose $a \geq 0$.
    - Suppose $b \geq 0$. Then $(ab) \geq 0$. So $(ab) \cdot I = [ab\ell, abu]$. Now $a \cdot (b \cdot I) = a \cdot [b\ell, bu] = [ab\ell, abu] = b \cdot [a\ell, au] = b \cdot (a \cdot I)$.
    - Suppose $b < 0$. Then $(ab) \leq 0$. So $(ab) \cdot I = [abu, ab\ell]$. Now $a \cdot (b \cdot I) = a \cdot [bu, b\ell] = [abu, ab\ell] = b \cdot [a\ell, au] = b \cdot (a \cdot I)$.
  - Suppose $a < 0$.
    - Suppose $b \geq 0$. Then $(ab) \leq 0$. So $(ab) \cdot I = [abu, ab\ell]$. Now $a \cdot (b \cdot I) = a \cdot [b\ell, bu] = [abu, ab\ell] = b \cdot [au, a\ell] = b \cdot (a \cdot I)$.
    - Suppose $b < 0$. Then $(ab) > 0$. So $(ab) \cdot I = [ab\ell, abu]$. Now $a \cdot (b \cdot I) = a \cdot [bu, b\ell] = [ab\ell, abu] = b \cdot [au, a\ell] = b \cdot (a \cdot I)$.

□

for corollary 2.

- To show $a \cdot I = -a \cdot (-1 \cdot I)$.
  - Let $I = \bot$. Then $a \cdot I = \bot = -a \cdot (-1 \cdot I)$.
  - Let $I = [\ell, u]$.

  - ∗ Suppose $a \geq 0$. Then $a \cdot I = a \cdot [\ell, u] = [a\ell, au] = -a \cdot [-u, -\ell] = -a \cdot (-1 \cdot [\ell, u]) = -a \cdot (-1 \cdot I)$.
  - ∗ Suppose $a < 0$. Then $a \cdot I = a \cdot [\ell, u] = [au, a\ell] = -a \cdot [-u, -\ell] = -a \cdot (-1 \cdot [\ell, u]) = -a \cdot (-1 \cdot I)$.

- To show $I = -1 \cdot (-1 \cdot I)$. Put $a = 1$, noting $1 \cdot I = I$.

□

for lemma 6.

- Suppose $I_1 = \bot$, $I_2 \in \mathsf{Int}$. Then $a \cdot (I_1 \sqcup I_2) = a \cdot I_2 = \bot \sqcup a \cdot I_2 = a \cdot I_1 \sqcup a \cdot I_2$.
- Suppose $I_2 = \bot$, $I_1 \in \mathsf{Int}$. This case is analogous to the previous case.
- Suppose $I_1, I_2 \in \mathsf{Int}$ where $I_1 \neq \bot \neq I_2$. Let $I_1 = [\ell_1, u_1]$ and $I_2 = [\ell_2, u_2]$.
  - Suppose $a \geq 0$.
    - ∗ Suppose $\ell_1 < \ell_2$ and $u_1 < u_2$. Then $a\ell_1 \leq a\ell_2$ and $au_1 \leq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_1, u_2] = [a\ell_1, au_2] = [a\ell_1, au_1] \sqcup [a\ell_2, au_2] = a \cdot I_1 \sqcup a \cdot I_2$.
    - ∗ Suppose $\ell_1 < \ell_2$ and $u_1 \geq u_2$. Then $a\ell_1 \leq a\ell_2$ and $au_1 \geq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_1, u_1] = [a\ell_1, au_1] = [a\ell_1, au_1] \sqcup [a\ell_2, au_2] = a \cdot I_1 \sqcup a \cdot I_2$.
    - ∗ Suppose $\ell_1 \geq \ell_2$ and $u_1 < u_2$. Then $a\ell_1 \geq a\ell_2$ and $au_1 \leq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_2, u_2] = [a\ell_2, au_2] = [a\ell_1, au_1] \sqcup [a\ell_2, au_2] = a \cdot I_1 \sqcup a \cdot I_2$.
    - ∗ Suppose $\ell_1 \geq \ell_2$ and $u_1 \geq u_2$. Then $a\ell_1 \geq a\ell_2$ and $au_1 \geq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_2, u_1] = [a\ell_2, au_1] = [a\ell_1, au_1] \sqcup [a\ell_2, au_2] = a \cdot I_1 \sqcup a \cdot I_2$.
  - Suppose $a < 0$.
    - ∗ Suppose $\ell_1 < \ell_2$ and $u_1 < u_2$. Then $a\ell_1 > a\ell_2$ and $au_1 > au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_1, u_2] = [au_2, a\ell_1] = [au_1, a\ell_1] \sqcup [au_2, a\ell_2] = a \cdot I_1 \sqcup a \cdot I_2$.

  - Suppose $\ell_1 < \ell_2$ and $u_1 \geq u_2$. Then $a\ell_1 > a\ell_2$ and $au_1 \leq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_1, u_1] = [au_1, a\ell_1] = [au_1, a\ell_1] \sqcup [au_2, a\ell_2] = a \cdot I_1 \sqcup a \cdot I_2$.
  - Suppose $\ell_1 \geq \ell_2$ and $u_1 < u_2$. Then $a\ell_1 \leq a\ell_2$ and $au_1 > au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_2, u_2] = [au_2, a\ell_2] = [au_1, a\ell_1] \sqcup [au_2, a\ell_2] = a \cdot I_1 \sqcup a \cdot I_2$.
  - Suppose $\ell_1 \geq \ell_2$ and $u_1 \geq u_2$. Then $a\ell_1 \leq a\ell_2$ and $au_1 \leq au_2$. Hence $a \cdot (I_1 \sqcup I_2) = a \cdot ([\ell_1, u_1] \sqcup [\ell_2, u_2]) = a \cdot [\ell_2, u_1] = [au_1, a\ell_2] = [au_1, a\ell_1] \sqcup [au_2, a\ell_2] = a \cdot I_1 \sqcup a \cdot I_2$.

□

for corollary 3.

- First, using lemma 6 and then lemma 5 twice, $-1 \cdot (a \cdot I_1 \sqcup b \cdot I_2) = -1 \cdot (a \cdot I_1) \sqcup -1 \cdot (b \cdot I_2) = a \cdot (-1 \cdot I_1) \sqcup b \cdot (-1 \cdot I_2)$.
- Second, using lemma 6 and then lemma 5 twice, $c \cdot ((a/c) \cdot I_1 \sqcup (b/c) \cdot I_2) = c \cdot ((a/c) \cdot I_1) \sqcup c \cdot ((b/c) \cdot I_2) = (ca)/c \cdot I_1 \sqcup (cb)/c \cdot I_2 = a \cdot I_1 \sqcup b \cdot I_2$.

□

for lemma 7.

- Suppose $k_1 I_1 \sqsubseteq k_2 I_2$.
  - Suppose $k_1 I_1 = \bot$. Then $\gamma(k_1 I_1) = \emptyset \subseteq \gamma(k_2 I_2)$.
  - Suppose $k_1 I_1 \neq \bot$. Then $k_2 I_2 \neq \bot$. Hence $k_2 | k_1$ and $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$. Let $I_1 = [\ell_1, u_1]$ where $\ell_1 \leq u_1$ and $I_2 = [\ell_2, u_2]$ where $\ell_2 \leq u_2$. Since $k_2 | k_1$, then $\lambda k_2 = k_1$ for some $\lambda \in \mathbb{Z}$.
    - Suppose $k_2 > 0$.
      - Suppose $k_1 > 0$. Then $\lambda > 0$. Since $[k_1\ell_1, k_1u_1] = k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2 = [k_2\ell_2, k_2u_2]$, then $k_2\ell_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2u_2$. Hence $k_2\ell_2 \leq \lambda k_2\ell_1 \leq \lambda k_2u_1 \leq k_2u_2$ and $\ell_2 \leq \lambda\ell_1 \leq \lambda u_1 \leq u_2$ since $k_2 > 0$. Let $y \in \gamma(k_1 I_1) = \gamma(k_1[\ell_1, u_1])$. Therefore $y = k_1 x$ for some $\ell_1 \leq x \leq u_1$. Therefore $\ell_2 \leq \lambda\ell_1 \leq \lambda x \leq \lambda u_1 \leq u_2$ since $\lambda > 0$. Hence $y = k_1 x = k_2 \lambda x \in \gamma(k_2[\ell_2, u_2]) = \gamma(k_2 I_2)$.

· Suppose $k_1 \leq 0$. Then by definition 9, $I_1 = [1,1]$. Since $[k_1,k_1] = k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2 = [k_2\ell_2, k_2u_2]$, then $k_2\ell_2 \leq k_1 \leq k_2u_2$. Hence $k_2\ell_2 \leq \lambda k_2 \leq k_2u_2$ and $\ell_2 \leq \lambda \leq u_2$ since $k_2 > 0$. Let $y \in \gamma(k_1I_1) = \{k_1\}$. Hence $y = k_1 = k_2\lambda \in \gamma(k_2[\ell_2,u_2]) = \gamma(k_2I_2)$.

∗ Suppose $k_2 \leq 0$. Then by definition 9, $I_2 = [1,1]$. Thus by lemma 9, $k_1 = k_2$ and $I_1 = [1,1]$. Hence $\gamma(k_1I_1) = \{k_1\} \subseteq \{k_2\} = \gamma(k_2I_2)$.

- Suppose $\gamma(k_1I_1) \subseteq \gamma(k_2I_2)$
  - Suppose $k_1I_1 = \bot$. Then $k_1I_1 \sqsubseteq k_2I_2$ follows immediately.
  - Suppose $k_1I_1 \neq \bot$. Then $k_2I_2 \neq \bot$.
    - To show $k_2|k_1$.
      · Suppose $I_1 = [1,1]$. Hence $\gamma(k_1I_1) = \{k_1\} \subseteq \gamma(k_2I_2)$. Therefore $k_1 = k_2x$ for some $x \in \gamma(I_2)$. Since $x \in \mathbb{Z}$, $k_2|k_1$.
      · Suppose $I_1 = [\ell_1,u_1]$ where $\ell_1 < u_1$. Since $\ell_1 < u_1$, it follows $\ell_1 < \ell_1 + 1 \leq u_1$. Observe $\{k_1\ell_1, k_1(\ell_1+1)\} \subseteq \gamma(k_1I_1) \subseteq \gamma(k_2I_2)$. Thus $k_1\ell_1 = k_2x$ and $k_1(\ell_1+1) = k_2y$ for some $x,y \in \gamma(I_2)$. Hence $k_1 = k_1((\ell_1+1)-\ell_1) = k_2(y-x)$. Therefore $k_2|k_1$.
    - To show $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$
      · Suppose $k_1 > 0$. Let $y \in \gamma(k_1 \cdot I_1) = \gamma([k_1\ell_1, k_1u_1])$. Observe $\{k_1\ell_1, k_1u_1\} \subseteq \gamma(k_1I_1) \subseteq \gamma(k_2I_2) \subseteq \gamma(k_2 \cdot I_2) = \gamma([\ell',u'])$ for some $\ell' \leq u'$. Therefore $\ell' \leq k_1\ell_1 \leq y \leq k_1u_1 \leq u'$. Hence $y \in \gamma(k_2 \cdot I_2)$.
      · Suppose $k_1 \leq 0$. This case follows analogously.

□

for proposition 4.

- Let $d \in \mathbb{Z}$. To show $(dk_1) \cdot I_1 \sqsubseteq (dk_2) \cdot I_2$. Since $k_1I_1 \sqsubseteq k_2I_2$, then $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$. Let $I_1 = [\ell_1,u_1]$ and $I_2 = [\ell_2,u_2]$.
  - Suppose $k_1 \geq 0$.

* Suppose $k_2 \geq 0$.
  - Suppose $d \geq 0$. Since $k_1 \cdot I_1 = [k_1\ell_1, k_1u_1] \sqsubseteq [k_2\ell_2, k_2u_2] = k_2 \cdot I_2$, then $k_2\ell_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2u_2$. Since $d \geq 0$, then $dk_2\ell_2 \leq dk_1\ell_1 \leq dk_1u_1 \leq dk_2u_2$. Also $dk_1 \geq 0$ and $dk_2 \geq 0$. Hence $(dk_1) \cdot I_1 = [dk_1\ell_1, dk_1u_1] \sqsubseteq [dk_2\ell_2, dk_2u_2] = (dk_2) \cdot I_2$.
  - Suppose $d < 0$. As in the previous case, $k_2\ell_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2u_2$. Since $d < 0$, then $dk_2u_2 \leq dk_1u_1 \leq dk_1\ell_1 \leq dk_2\ell_2$. Also $dk_1 \leq 0$ and $dk_2 \leq 0$. Hence $(dk_1) \cdot I_1 = [dk_1u_1, dk_1\ell_1] \sqsubseteq [dk_2u_2, dk_2\ell_2] = (dk_2) \cdot I_2$.
* Suppose $k_2 < 0$.
  - Suppose $d \geq 0$. Since $k_1 \cdot I_1 = [k_1\ell_1, k_1u_1] \sqsubseteq [k_2u_2, k_2\ell_2] = k_2 \cdot I_2$, then $k_2u_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2\ell_2$. Since $d \geq 0$, then $dk_2u_2 \leq dk_1\ell_1 \leq dk_1u_1 \leq dk_2\ell_2$. Also $dk_1 \geq 0$ and $dk_2 \leq 0$. Hence $(dk_1) \cdot I_1 = [dk_1\ell_1, dk_1u_1] \sqsubseteq [dk_2u_2, dk_2\ell_2] = (dk_2) \cdot I_2$.
  - Suppose $d < 0$. As in the previous case, $k_2u_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2\ell_2$. Since $d < 0$, then $dk_2\ell_2 \leq dk_1u_1 \leq dk_1\ell_1 \leq dk_2u_2$. Also $dk_1 \leq 0$ and $dk_2 > 0$. Hence $(dk_1) \cdot I_1 = [dk_1u_1, dk_1\ell_1] \sqsubseteq [dk_2\ell_2, dk_2u_2] = (dk_2) \cdot I_2$.

– Suppose $k_1 < 0$.

* Suppose $k_2 \geq 0$.
  - Suppose $d \geq 0$. Since $k_1 \cdot I_1 = [k_1u_1, k_1\ell_1] \sqsubseteq [k_2\ell_2, k_2u_2] = k_2 \cdot I_2$, then $k_2\ell_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2u_2$. Since $d \geq 0$, then $dk_2\ell_2 \leq dk_1u_1 \leq dk_1\ell_1 \leq dk_2u_2$. Also $dk_1 \leq 0$ and $dk_2 \geq 0$. Hence $dk_1 \cdot I_1 = [dk_1u_1, dk_1\ell_1] \sqsubseteq [dk_2\ell_2, dk_2u_2] = (dk_2) \cdot I_2$.
  - Suppose $d < 0$. As in the previous case, $k_2\ell_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2u_2$. Since $d < 0$, then $dk_2u_2 \leq dk_1\ell_1 \leq dk_1u_1 \leq dk_2\ell_2$. Also $dk_1 > 0$ and $dk_2 \leq 0$. Hence $dk_1 \cdot I_1 = [dk_1\ell_1, dk_1u_1] \sqsubseteq [dk_2u_2, dk_2\ell_2] = (dk_2) \cdot I_2$.
* Suppose $k_2 < 0$.
  - Suppose $d \geq 0$. Since $k_1 \cdot I_1 = [k_1u_1, k_1\ell_1] \sqsubseteq [k_2u_2, k_2\ell_2] = k_2 \cdot I_2$, then $k_2u_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2\ell_2$. Since $d \geq 0$, then

$dk_2u_2 \leq dk_1u_1 \leq dk_1\ell_1 \leq dk_2\ell_2$. Also $dk_1 \leq 0$ and $dk_2 \leq 0$. Hence $(dk_1) \cdot I_1 = [dk_1u_1, dk_1\ell_1] \sqsubseteq [dk_2u_2, dk_2\ell_2] = (dk_2) \cdot I_2$.

· Suppose $d < 0$. As in the previous case, $k_2u_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2\ell_2$. Since $d < 0$, then $dk_2\ell_2 \leq dk_1\ell_1 \leq dk_1u_1 \leq dk_2u_2$. Also $dk_1 > 0$ and $dk_2 > 0$. Hence $(dk_1) \cdot I_1 = [dk_1\ell_1, dk_1u_1] \sqsubseteq [dk_2\ell_2, dk_2u_2] = (dk_2) \cdot I_2$.

- Let $d \in \mathbb{Z} \setminus \{0\}$, $d|k_1$ and $d|k_2$. To show $(k_1/d) \cdot I_1 \sqsubseteq (k_2/d) \cdot I_2$. Since $k_1I_1 \sqsubseteq k_2I_2$, then $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$. Let $I_1 = [\ell_1, u_1]$ and $I_2 = [\ell_2, u_2]$.
  - Suppose $k_1 \geq 0$.
    - Suppose $k_2 \geq 0$.
      · Suppose $d > 0$. Since $k_1 \cdot I_1 = [k_1\ell_1, k_1u_1] \sqsubseteq [k_2\ell_2, k_2u_2] = k_2 \cdot I_2$, then $k_2\ell_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2u_2$. Since $d > 0$, $(k_2/d)\ell_2 \leq (k_1/d)\ell_1 \leq (k_1/d)u_1 \leq (k_2/d)u_2$. Also $(k_1/d) \geq 0$ and $(k_2/d) \geq 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)\ell_1, (k_1/d)u_1] \sqsubseteq [(k_2/d)\ell_2, (k_2/d)u_2] = (k_2/d) \cdot I_2$.
      · Suppose $d < 0$. As in the previous case, $k_2\ell_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2u_2$. Since $d < 0$, $(k_2/d)u_2 \leq (k_1/d)u_1 \leq (k_1/d)\ell_1 \leq (k_2/d)\ell_2$. Also $(k_1/d) \leq 0$ and $(k_2/d) \leq 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)u_1, (k_1/d)\ell_1] \sqsubseteq [(k_2/d)u_2, (k_2/d)\ell_2] = (k_2/d) \cdot I_2$.
    - Suppose $k_2 < 0$.
      · Suppose $d > 0$. Since $k_1 \cdot I_1 = [k_1\ell_1, k_1u_1] \sqsubseteq [k_2u_2, k_2\ell_2] = k_2 \cdot I_2$, then $k_2u_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2\ell_2$. Since $d > 0$, $(k_2/d)u_2 \leq (k_1/d)\ell_1 \leq (k_1/d)u_1 \leq (k_2/d)\ell_2$. Also $(k_1/d) \geq 0$ and $(k_2/d) < 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)\ell_1, (k_1/d)u_1] \sqsubseteq [(k_2/d)u_2, (k_2/d)\ell_2] = (k_2/d) \cdot I_2$.
      · Suppose $d < 0$. As in the previous case, $k_2u_2 \leq k_1\ell_1 \leq k_1u_1 \leq k_2\ell_2$. Since $d < 0$, $(k_2/d)\ell_2 \leq (k_1/d)u_1 \leq (k_1/d)\ell_1 \leq (k_2/d)u_2$. Also $(k_1/d) \leq 0$ and $(k_2/d) > 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)u_1, (k_1/d)\ell_1] \sqsubseteq [(k_2/d)\ell_2, (k_2/d)u_2] = (k_2/d) \cdot I_2$.
  - Suppose $k_1 < 0$.
    - Suppose $k_2 \geq 0$.

- Suppose $d > 0$. Since $k_1 \cdot I_1 = [k_1u_1, k_1\ell_1] \sqsubseteq [k_2\ell_2, k_2u_2] = k_2 \cdot I_2$, then $k_2\ell_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2u_2$. Since $d > 0$, $(k_2/d)\ell_2 \leq (k_1/d)u_1 \leq (k_1/d)\ell_1 \leq (k_2/d)u_2$. Also $(k_1/d) < 0$ and $(k_2/d) \geq 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)u_1, (k_1/d)\ell_1] \sqsubseteq [(k_2/d)\ell_2, (k_2/d)u_2] = (k_2/d) \cdot I_2$.
- Suppose $d < 0$. As in the previous case, $k_2\ell_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2u_2$. Since $d < 0$, $(k_2/d)u_2 \leq (k_1/d)\ell_1 \leq (k_1/d)u_1 \leq (k_2/d)\ell_2$. Also $(k_1/d) > 0$ and $(k_2/d) \leq 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)\ell_1, (k_1/d)u_1] \sqsubseteq [(k_2/d)u_2, (k_2/d)\ell_2] = (k_2/d) \cdot I_2$.

* Suppose $k_2 < 0$.
  - Suppose $d > 0$. Since $k_1 \cdot I_1 = [k_1u_1, k_1\ell_1] \sqsubseteq [k_2u_2, k_2\ell_2] = k_2 \cdot I_2$, then $k_2u_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2\ell_2$. Since $d > 0$, $(k_2/d)u_2 \leq (k_1/d)u_1 \leq (k_1/d)\ell_1 \leq (k_2/d)\ell_2$. Also $(k_1/d) < 0$ and $(k_2/d) < 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)u_1, (k_1/d)\ell_1] \sqsubseteq [(k_2/d)u_2, (k_2/d)\ell_2] = (k_2/d) \cdot I_2$.
  - Suppose $d < 0$. As in the previous case, $k_2u_2 \leq k_1u_1 \leq k_1\ell_1 \leq k_2\ell_2$. Since $d < 0$, $(k_2/d)\ell_2 \leq (k_1/d)\ell_1 \leq (k_1/d)u_1 \leq (k_2/d)u_2$. Also $(k_1/d) > 0$ and $(k_2/d) > 0$. Hence $(k_1/d) \cdot I_1 = [(k_1/d)\ell_1, (k_1/d)u_1] \sqsubseteq [(k_2/d)\ell_2, (k_2/d)u_2] = (k_2/d) \cdot I_2$.

□

for lemma 8. Suppose $I = [\ell, u]$. Let $y \in \gamma(kI)$.

- Suppose $k \geq 0$. Then $y = kx$ for some $\ell \leq x \leq u$. Hence $k\ell \leq kx \leq ku$ and $y = kx \in \gamma([k\ell, ku]) = \gamma(k \cdot [\ell, u])$.
- Suppose $k < 0$. Then $y = kx$ for some $\ell \leq x \leq u$. Hence $ku \leq kx \leq k\ell$ and $y = kx \in \gamma([ku, k\ell]) = \gamma(k \cdot [\ell, u])$.

□

for corollary 4. Observe $1|k$ and, by lemma 5, $k \cdot I \sqsubseteq k \cdot I = (1k) \cdot I = 1 \cdot (k \cdot I)$. □

for lemma 9.

- Let $k_1I_1 \sqsubseteq k_2I_2$ where $I_2 = [1, 1]$.

  - Suppose $k_1 I_1 = \bot$. Then, by definition 10, $k_1 I_1 \sqsubseteq k_2 I_2$.
  - Suppose $k_1 I_1 \neq \bot$. By definition 10, $k_1 \cdot I_1 \sqsubseteq k_2 \cdot I_2$. Hence by lemma 3, $\gamma(k_1 \cdot I_1) \subseteq \gamma(k_2 \cdot I_2) = \gamma(k_2 \cdot [1,1]) = \{k_2\}$. Therefore it must hold that $|\gamma(k_1 \cdot I_1)| = 1$ and thus $I_1 = [1,1]$. So $\{k_1\} = \gamma(k_1 \cdot [1,1]) = \gamma(k_1 \cdot I_1) \subseteq \{k_2\}$. Hence $k_1 = k_2$.

- Let $|\gamma(kI)| = 1$. Then $|\{kx \mid x \in \gamma(I)\}| = 1$. Thus $|\gamma(I)| = 1$ and so $I = [1,1]$.

□

for proposition 5. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$. To show $k_1 I_1 \sqcup k_2 I_2$ is a least upper bound.

- To show $k_1 I_1 \sqsubseteq k_1 I_1 \sqcup k_2 I_2$.
  - Suppose $k_1 I_1 = \bot$. Then $k_1 I_1 = \bot \sqsubseteq k_2 I_2 = k_1 I_1 \sqcup k_2 I_2$.
  - Suppose $k_1 = k_2$ and $I_1 = I_2 = [1,1]$. Then $k_1 I_1 \sqsubseteq k_1 I_1 = k_1 I_1 \sqcup k_2 I_2$.
  - Suppose otherwise. By the reasoning immediately following proposition 5, $k > 0$, and by the definition of *gcd*, then $k | k_1$ and $k | k_2$. So by proposition 3, $(k_1/k) \cdot I_1 \sqsubseteq (k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2$ and thus by lemma 5 and then by lemma 4, $k_1 \cdot I_1 = k \cdot \big((k_1/k) \cdot I_1\big) \sqsubseteq k \cdot \big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big)$. Hence by definition 10, $k_1 I_1 \sqsubseteq k\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) = k_1 \sqcup k_2$.
- To show $k_2 I_2 \sqsubseteq k_1 I_1 \sqcup k_2 I_2$ is analogous to above.
- Let $k_1 I_1 \sqsubseteq k' I$ and $k_2 I_2 \sqsubseteq k' I$. To show $k_1 I_1 \sqcup k_2 I_2 \sqsubseteq k' I$.
  - Suppose $k' I = \bot$. Then $k_1 I_1 = k_2 I_2 = \bot$. Thus $k_1 I_1 \sqcup k_2 I_2 = \bot \sqsubseteq k' I$.
  - Suppose $k' I = \top$. Then $k_1 I_1 \sqcup k_2 I_2 \sqsubseteq k' I$.
  - Suppose $k' I = k'[1,1]$. Since $k_1 I_1 \sqsubseteq k' I$, then by lemma 9, $k_1 = k'$ and $I_1 = [1,1]$. Likewise, $k_2 = k'$ and $I_2 = [1,1]$. Thus $k_1 I_1 \sqcup k_2 I_2 = k_1 I_1 \sqsubseteq k'[1,1]$.
  - Suppose otherwise. To show $k\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) \sqsubseteq k' I$.
    - To show $k' | k$. Observe since $k_1 I_1 \sqsubseteq k' I$, then $k' | k_1$. Likewise, since $k_2 I_2 \sqsubseteq k' I$, then $k' | k_2$. Hence, by lemma 1, $k' | k$.

* To show $k \cdot \big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) \sqsubseteq k' \cdot I$. Since $k_1 I_1 \sqsubseteq k' I$, then $k_1 \cdot I_1 \sqsubseteq k' \cdot I$. Likewise, since $k_2 I_2 \sqsubseteq k' I$, then $k_2 \cdot I_2 \sqsubseteq k' \cdot I$. Also $k \neq 0$ by the reasoning immediately following proposition 5. Hence by corollary 3 and then by proposition 3, $k \cdot \big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) = k_1 \cdot I_1 \sqcup k_2 \cdot I_2 \sqsubseteq k' \cdot I$.

□

for lemma 10. Let $kI \in \mathsf{kInt} \setminus \{\top\}$.

- Suppose $kI = \bot$. Then $\gamma(\lambda \otimes kI) = \gamma(\bot) = \emptyset = \{\lambda x \mid x \in \gamma(\bot)\} = \{\lambda x \mid x \in \gamma(kI)\}$.
- Suppose $kI \neq \bot$.
  - Suppose $I = [1, 1]$. Then $\lambda \otimes kI = (\lambda k)I$. Hence $\gamma(\lambda \otimes kI) = \gamma((\lambda k)I) = \{(\lambda k)x \mid x \in \gamma(I)\} = \{\lambda k\} = \{\lambda x \mid x \in \gamma(kI)\}$.
  - Suppose $I \neq [1, 1]$ and $\lambda > 0$. Then $\gamma(\lambda \otimes kI) = \gamma((\lambda k)I) = \{\lambda x \mid x \in \gamma(kI)\}$.
  - Suppose $I \neq [1, 1]$ and $\lambda = 0$. Then $\gamma(\lambda \otimes kI) = \gamma(0[1, 1]) = \{\lambda x \mid x \in \gamma(kI)\}$.
  - Suppose $I \neq [1, 1]$ and $\lambda < 0$. Then $\gamma(\lambda \otimes kI) = \gamma((-\lambda k)(-1 \cdot I))$. Since $kI \in \mathsf{kInt}$ and $I \neq [1, 1]$, then $k > 0$. Thus $(-\lambda k) > 0$. Let $I = [\ell, u]$. Thus $(-1 \cdot I) = [-u, -\ell]$. Therefore $\gamma(\lambda \otimes kI) = \gamma((-\lambda k)(-1 \cdot I)) = \gamma((-\lambda k)[-u, -\ell])) = \{-\lambda k x \mid -u \leq x \leq -\ell\} = \{\lambda k x \mid \ell \leq x \leq u\} = \{\lambda k x \mid x \in \gamma(I)\} = \{\lambda x \mid x \in \gamma(kI)\}$.

□

for lemma 11. Let $kI \in \mathsf{kInt} \setminus \{\bot, \top\}$.

- Suppose $k > 0$. Hence $k \otimes 1I = (k \times 1)I = kI$.
- Suppose $k = 0$. Then $I = [1, 1]$. Hence $k \otimes 1I = (0 \times 1)I = 0[1, 1] = kI$.
- Suppose $k < 0$. Then $I = [1, 1]$. Hence $k \otimes 1I = (k \times 1)I = kI$.

□

for lemma 12.

- Let $\lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2$. To show $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$.
  - Suppose $I_1 = [1,1]$ or $\lambda > 0$.
    - Suppose $I_2 = [1,1]$ or $\mu > 0$. Then $(\lambda k_1) I_1 = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (\mu k_2) I_2$. By definition 10, $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$.
    - Suppose $\mu = 0$. Then $(\lambda k_1) I_1 = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = 0[1,1] = (\mu k_2)[1,1]$. By definition 10, $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot [1,1] = [0,0] = \mu k_2 \cdot I_2$.
    - Suppose otherwise. Then $(\lambda k_1) I_1 = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (-\mu k_2)(-1 \cdot I_2)$. By definition 10, $-\mu k_2 | \lambda k_1$ and so $\mu k_2 | \lambda k_1$, and by corollary 2, $\lambda k_1 \cdot I_1 \sqsubseteq -\mu k_2 \cdot (-1 \cdot I_2) = \mu k_2 \cdot I_2$.
  - Suppose $\lambda = 0$.
    - Suppose $I_2 = [1,1]$ or $\mu > 0$. Then $(\lambda k_1)[1,1] = 0[1,1] = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (\mu k_2) I_2$. By definition 10, $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 = [0,0] = \lambda k_1 \cdot [1,1] \sqsubseteq \mu k_2 \cdot I_2$.
    - Suppose $\mu = 0$. Then $(\lambda k_1)[1,1] = 0[1,1] = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = 0[1,1] = (\mu k_2)[1,1]$. By definition 10, $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 = [0,0] = \lambda k_1 \cdot [1,1] \sqsubseteq \mu k_2 \cdot [1,1] = [0,0] = \mu k_2 \cdot I_2$.
    - Suppose otherwise. Then $(\lambda k_1)[1,1] = 0[1,1] = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (-\mu k_2)(-1 \cdot I_2)$. By definition 10, $-\mu k_2 | \lambda k_1$ and so $\mu k_2 | \lambda k_1$, and by corollary 2, $\lambda k_1 \cdot I_1 = [0,0] = \lambda k_1 \cdot [1,1] \sqsubseteq -\mu k_2 \cdot (-1 \cdot I_2) = \mu k_2 \cdot I_2$.
  - Suppose otherwise.
    - Suppose $I_2 = [1,1]$ or $\mu > 0$. Then $(-\lambda k_1)(-1 \cdot I_1) = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (\mu k_2) I_2$. By definition 10, $\mu k_2 | -\lambda k_1$ and so $\mu k_2 | \lambda k_1$, and by corollary 2, $\lambda k_1 \cdot I_1 = -\lambda k_1 \cdot (-1 \cdot I_1) \sqsubseteq \mu k_2 \cdot I_2$.
    - Suppose $\mu = 0$. Then $(-\lambda k_1)(-1 \cdot I_1) = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = 0[1,1] = (\mu k_2)[1,1]$. By definition 10, $\mu k_2 | -\lambda k_1$ and so $\mu k_2 | \lambda k_1$, and by corollary 2, $\lambda k_1 \cdot I_1 = -\lambda k_1 \cdot (-1 \cdot I_1) \sqsubseteq \mu k_2 \cdot [1,1] = [0,0] = \mu k_2 \cdot I_2$.

    * Suppose otherwise. Then $(-\lambda k_1)(-1 \cdot I_1) = \lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2 = (-\mu k_2)(-1 \cdot I_2)$. By definition 10, $-\mu k_2 | -\lambda k_1$ and so $\mu k_2 | \lambda k_1$, and by two applications of corollary 2, $\lambda k_1 \cdot I_1 = -\lambda k_1 \cdot (-1 \cdot I_1) \sqsubseteq -\mu k_2 \cdot (-1 \cdot I_2) = \mu k_2 \cdot I_2$.

- Let $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$. To show $\lambda \otimes k_1 I_1 \sqsubseteq \mu \otimes k_2 I_2$.
  - Suppose $I_1 = [1, 1]$ or $\lambda > 0$.
    * Suppose $I_2 = [1, 1]$ or $\mu > 0$. Since $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$, then by definition 10, $\lambda \otimes k_1 I_1 = (\lambda k_1) I_1 \sqsubseteq (\mu k_2) I_2 = \mu \otimes k_2 I_2$.
    * Suppose $\mu = 0$. Since $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = [0, 0] = (\mu k_2) \cdot [1, 1]$, then by definition 10, $\lambda \otimes k_1 I_1 = (\lambda k_1) I_1 \sqsubseteq (\mu k_2)[1, 1] = 0[1, 1] = \mu \otimes k_2 I_2$.
    * Suppose otherwise. Since $\mu k_2 | \lambda k_1$, then $-\mu k_2 | \lambda k_1$, and using corollary 2, since $\lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = -\mu k_2 \cdot (-1 \cdot I_2)$, then by definition 10, $\lambda \otimes k_1 I_1 = (\lambda k_1) I_1 \sqsubseteq (-\mu k_2)(-1 \cdot I_2) = \mu \otimes k_2 I_2$.
  - Suppose $\lambda = 0$.
    * Suppose $I_2 = [1, 1]$ or $\mu > 0$. Since $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot [1, 1] = [0, 0] = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$, then by definition 10, $\lambda \otimes k_1 I_1 = 0[1, 1] = (\lambda k_1)[1, 1] \sqsubseteq (\mu k_2) I_2 = \mu \otimes k_2 I_2$.
    * Suppose $\mu = 0$. Since $\mu k_2 | \lambda k_1$ and $\lambda k_1 \cdot [1, 1] = [0, 0] = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = [0, 0] = \mu k_2 \cdot [1, 1]$, then by definition 10, $\lambda \otimes k_1 I_1 = 0[1, 1] = (\lambda k_1)[1, 1] \sqsubseteq (\mu k_2)[1, 1] = 0[1, 1] = \mu \otimes k_2 I_2$.
    * Suppose otherwise. Since $\mu k_2 | \lambda k_1$, then $-\mu k_2 | \lambda k_1$, and using corollary 2, since $\lambda k_1 \cdot [1, 1] = [0, 0] = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = -\mu k_2 \cdot (-1 \cdot I_2)$, then by definition 10, $\lambda \otimes k_1 I_1 = 0[1, 1] = (\lambda k_1)[1, 1] \sqsubseteq (-\mu k_2)(-1 \cdot I_2) = \mu \otimes k_2 I_2$.
  - Suppose otherwise.
    * Suppose $I_2 = [1, 1]$ or $\mu > 0$. Since $\mu k_2 | \lambda k_1$, then $\mu k_2 | -\lambda k_1$, and using corollary 2, since $-\lambda k_1 \cdot (-1 \cdot I_1) = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2$, then by definition 10, $\lambda \otimes k_1 I_1 = (-\lambda k_1)(-1 \cdot I_1) \sqsubseteq (\mu k_2) I_2 = \mu \otimes k_2 I_2$.

- Suppose $\mu = 0$. Since $\mu k_2 | \lambda k_1$, then $\mu k_2 | -\lambda k_1$, and using corollary 2, since $-\lambda k_1 \cdot (-1 \cdot I_1) = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = [0,0] = \mu k_2 \cdot [1,1]$, then by definition 10, $\lambda \otimes k_1 I_1 = (-\lambda k_1)(-1 \cdot I_1) \sqsubseteq (\mu k_2)[1,1] = 0[1,1] = \mu \otimes k_2 I_2$.
- Suppose otherwise. Since $\mu k_2 | \lambda k_1$, then $-\mu k_2 | -\lambda k_1$, and by two applications of corollary 2, since $-\lambda k_1 \cdot (-1 \cdot I_1) = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = -\mu k_2 \cdot (-1 \cdot I_2)$, then by definition 10, $\lambda \otimes k_1 I_1 = (-\lambda k_1)(-1 \cdot I_1) \sqsubseteq (-\mu k_2)(-1 \cdot I_2) = \mu \otimes k_2 I_2$.

□

for lemma 13. Let $I = [\ell, u]$ where $\ell < u$. Observe $\gamma(k(\lambda + I)) = \{kx \mid x \in (\lambda + I)\} = \{kx \mid x \in (\lambda + [\ell, u])\} = \{kx \mid x \in [\lambda + \ell, \lambda + u]\}$. Also observe $\gamma(-k(-\lambda + (-1 \cdot I))) = \{-kx \mid x \in (-\lambda + (-1 \cdot I))\} = \{-kx \mid x \in (-\lambda + (-1 \cdot [\ell, u]))\} = \{-kx \mid x \in (-\lambda + [-u, -\ell])\} = \{-kx \mid x \in [-\lambda - u, -\lambda - \ell]\} = \{kx \mid x \in [\lambda + \ell, \lambda + u]\} = \gamma(k(\lambda + I))$. Hence, the result holds by lemma 7. □

for corollary 5. Let $k_1 = k_2 = \mu = 1$. Then $\mu k_2 = 1 | \lambda = \lambda k_1$. Thus by lemma 12, $\lambda \otimes 1 I_1 \sqsubseteq \mu \otimes 1 I_2 = 1 \otimes 1 I_2 = 1 I_2$ iff $\lambda \cdot I_1 = \lambda k_1 \cdot I_1 \sqsubseteq \mu k_2 \cdot I_2 = 1 \cdot I_2$. □

for corollary 6. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$.

- Suppose $k_1 I_1 = \bot$.
  - Let $\lambda \in \mathbb{Z}$. Then $\lambda \otimes k_1 I_1 = \bot \sqsubseteq \lambda \otimes k_2 I_2$, since $\lambda \otimes k_2 I_2 \in \mathsf{kInt}$ and $\bot \sqsubseteq kI$ for all $kI \in \mathsf{kInt}$.
  - Put $\lambda = 1$.
  - Put $\lambda = -1$.
- Suppose $k_1 I_1, k_2 I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$.
  - Since $k_1 I_1 \sqsubseteq k_2 I_2$ then $k_2 | k_1$, thus $\lambda k_2 | \lambda k_1$ and by proposition 4, $\lambda k_1 \cdot I_1 \sqsubseteq \lambda k_2 \cdot I_2$. The result then follows from lemma 12.
  - Put $\lambda = 1$.
  - Put $\lambda = -1$

- Suppose $k_2 I_2 = \top$. This is analogous to the first case.

□

for proposition 6. Let $k_1 I_1, k_2 I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$. Furthermore, let $d \in \mathbb{Z}$ such that $d > 0$.

- To show $(dk_1)I_1 \sqsubseteq (dk_2)I_2$. Observe $(dk_1)I_1, (dk_2)I_2 \in \mathsf{kInt}$.
  - To show $(dk_2)|(dk_1)$. Since $k_1 I_1 \sqsubseteq k_2 I_2$ then $k_2|k_1$, hence $\lambda k_2 = k_1$ for some $\lambda \in \mathbb{Z}$. So $\lambda(dk_2) = (dk_1)$. Thus $(dk_2)|(dk_1)$.
  - To show $(dk_1) \cdot I_1 \sqsubseteq (dk_2) \cdot I_2$. This follows from proposition 4.
- To show $(k_1/d)I_1 \sqsubseteq (k_2/d)I_2$. Observe $(k_1/d)I_1, (k_2/d)I_2 \in \mathsf{kInt}$.
  - To show $(k_2/d)|(k_1/d)$. Since $k_1 I_1 \sqsubseteq k_2 I_2$ then $k_2|k_1$, hence $\lambda k_2 = k_1$ for some $\lambda \in \mathbb{Z}$. Also since $d \neq 0$, $d|k_1$ and $d|k_2$, then $\lambda(k_2/d) = (k_1/d)$. Thus $(k_2/d)|(k_1/d)$.
  - To show $(k_1/d) \cdot I_1 \sqsubseteq (k_2/d) \cdot I_2$. Since $d \neq 0$, $d|k_1$, $d|k_2$ and $k_1 I_1 \sqsubseteq k_2 I_2$, then by proposition 4, $(k_1/d) \cdot I_1 \sqsubseteq (k_2/d) \cdot I_2$.

□

for corollary 7.

- Let $d \in \mathbb{Z}$. To show $(dk_1) \otimes 1I_1 \sqsubseteq (dk_2) \otimes 1I_2$.
  - Suppose $d > 0$. Since $k_1 I_1 \sqsubseteq k_2 I_2$, then by proposition 6, $(dk_1)I_1 \sqsubseteq (dk_2)I_2$. Thus by lemma 11, $(dk_1) \otimes 1I_1 \sqsubseteq (dk_2) \otimes 1I_2$.
  - Suppose $d = 0$. Then $(dk_1) \otimes 1I_1 = 0[1,1] \sqsubseteq 0[1,1] = (dk_2) \otimes 1I_2$.
  - Suppose $d < 0$. Observe $(-dk_1)I_1, (-dk_2)I_2 \in \mathsf{kInt}$. By proposition 6, $(-dk_1)I_1 \sqsubseteq (-dk_2)I_2$. Thus by lemma 11, $(-dk_1) \otimes 1I_1 \sqsubseteq (-dk_2) \otimes 1I_2$. Thus by corollary 6, $-1 \otimes \big(-dk_1 \otimes 1I_1\big) \sqsubseteq -1 \otimes \big(-dk_2 \otimes 1I_2\big)$. So by lemma 14, $(-1 \times -dk_1) \otimes 1I_1 \sqsubseteq (-1 \times -dk_2) \otimes 1I_2$. Thus $(dk_1) \otimes 1I_1 \sqsubseteq (dk_2) \otimes 1I_2$.
- Let $d \in \mathbb{Z} \setminus \{0\}$ such that $d|k_1$ and $d|k_2$. To show $(k_1/d) \otimes 1I_1 \sqsubseteq (k_2/d) \otimes 1I_2$.

- Suppose $d > 0$. Observe $(k_1/d)I_1, (k_2/d)I_2 \in \mathsf{kInt}$. Again by proposition 6 and lemma 11, $(k_1/d) \otimes 1I_1 = (k_1/d)I_1 \sqsubseteq (k_2/d)I_2 = (k_2/d) \otimes 1I_2$.
- Suppose $d < 0$. Since $-d > 0$, observe $(k_1/(-d))I_1, (k_2/(-d))I_2 \in \mathsf{kInt}$. Thus by lemma 11 and by proposition 6, $(k_1/(-d)) \otimes 1I_1 = (k_1/(-d))I_1 \sqsubseteq (k_2/(-d))I_2 = (k_2/(-d)) \otimes 1I_2$. Hence by corollary 6, $-1 \otimes \big(k_1/(-d) \otimes 1I_1\big) \sqsubseteq -1 \otimes \big(k_2/(-d) \otimes 1I_2\big)$. So by lemma 14, $(-1 \times k_1/(-d)) \otimes 1I_1 \sqsubseteq (-1 \times k_2/(-d)) \otimes 1I_2$. Thus $(k_1/d) \otimes 1I_1 \sqsubseteq (k_2/d) \otimes 1I_2$.

□

for lemma 14.

- Suppose $kI = \bot$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes \bot = \bot = (\lambda\mu) \otimes \bot = (\lambda\mu) \otimes kI$. Observe $\lambda \otimes (\mu \otimes kI) = \bot = \mu \otimes \bot = \mu \otimes (\lambda \otimes \bot) = \mu \otimes (\lambda \otimes kI)$.
- Suppose $kI = \top$. This case is analogous to the previous case.
- Suppose $kI \in \mathsf{kInt} \setminus \{\bot, \top\}$.
  - Suppose $\lambda > 0$.
    - Suppose $\mu > 0$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes (\mu k)I = (\lambda\mu k)I = (\lambda\mu) \otimes kI$ since $\lambda\mu > 0$. Observe $\mu \otimes (\lambda \otimes kI) = \mu \otimes (\lambda k)I = (\lambda\mu k)I = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.
    - Suppose $\mu = 0$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes (0 \otimes kI) = \lambda \otimes 0[1,1] = 0[1,1] = (\lambda\mu) \otimes kI$ since $\lambda\mu = 0$. Observe $\mu \otimes (\lambda \otimes kI) = 0[1,1] = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.
    - Suppose $\mu < 0$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes (-\mu k)(-1 \cdot I) = (-\lambda\mu k)(-1 \cdot I) = (\lambda\mu) \otimes kI$ since $\lambda\mu < 0$. Observe $\mu \otimes (\lambda \otimes kI) = \mu \otimes (\lambda k)I = (-\lambda\mu k)(-1 \cdot I) = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.
  - Suppose $\lambda = 0$. Then $\lambda \otimes (\mu \otimes kI) = 0 \otimes (\mu \otimes kI) = 0[1,1] = (\lambda\mu) \otimes kI$ since $\lambda\mu = 0$. Observe $\mu \otimes (\lambda \otimes kI) = \mu \otimes 0[1,1] = (\mu \times 0)[1,1] = 0[1,1] = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.
  - Suppose $\lambda < 0$.

  - Suppose $\mu > 0$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes (\mu k)I = (-\lambda\mu k)(-1 \cdot I) = (\lambda\mu) \otimes kI$ since $\lambda\mu < 0$. Observe $\mu \otimes (\lambda \otimes kI) = \mu \otimes (-\lambda k)(-1 \cdot I) = (-\lambda\mu k)(-1 \cdot I) = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.
  - Suppose $\mu = 0$. Analogous for the $\lambda > 0$ case.
  - Suppose $\mu < 0$. Then $\lambda \otimes (\mu \otimes kI) = \lambda \otimes (-\mu k)(-1 \cdot I) = (-\lambda(-\mu k))(-1 \cdot (-1 \cdot I)) = (\lambda\mu k)I = (\lambda\mu) \otimes kI$ because $\lambda\mu > 0$ and by corollary 2, $-1 \cdot (-1 \cdot I) = I$. Observe $\mu \otimes (\lambda \otimes kI) = \mu \otimes (-\lambda k)(-1 \cdot I) = (-\mu(-\lambda k))(-1 \cdot (-1 \cdot I)) = (\lambda\mu k)I = (\lambda\mu) \otimes kI = \lambda \otimes (\mu \otimes kI)$.

□

for corollary 8. By lemma 14, $\vec{x}/\vec{y} \otimes (\vec{w}/\vec{x} \otimes kI) = ((\vec{x}/\vec{y})(\vec{w}/\vec{x})) \otimes kI = (\vec{x}\vec{w})/(\vec{y}\vec{x}) \otimes kI = \vec{w}/\vec{y} \otimes kI$. □

for lemma 15.

- Suppose $k_1 I_1 = \bot$. Then $\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) = \lambda \otimes (\bot \sqcup k_2 I_2) = \lambda \otimes k_2 I_2 = \bot \sqcup \lambda \otimes k_2 I_2 = \lambda \otimes \bot \sqcup \lambda \otimes k_2 I_2 = \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2$.
- Suppose $k_1 I_1 = \top$. Then $\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) = \lambda \otimes (\top \sqcup k_2 I_2) = \lambda \otimes \top = \top = \top \sqcup \lambda \otimes k_2 I_2 = \lambda \otimes \top \sqcup \lambda \otimes k_2 I_2 = \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2$.
- Suppose $k_2 I_2 = \bot$. This is analogous to the first case but with $k_1 I_1$ and $k_2 I_2$ swapped.
- Suppose $k_2 I_2 = \top$. This is analogous to the second case but with $k_1 I_1$ and $k_2 I_2$ swapped.
- Suppose $k_1 I_1, k_2 I_2 \in \mathsf{kInt} \setminus \{\bot, \top\}$.
  - Suppose $k_1 = k_2$ and $I_1 = I_2 = [1, 1]$. Hence $k_1 I_1 = k_2 I_2$ and thus $\lambda \otimes k_1 I_1 = \lambda \otimes k_2 I_2$. Then $\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) = \lambda \otimes k_1 I_1 = \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2$.
  - Suppose otherwise. Then $k_1 \neq 0$ or $k_2 \neq 0$.

* Suppose $\lambda > 0$. Let $k = gcd(k_1, k_2)$ and $k' = gcd(\lambda k_1, \lambda k_2) = \lambda k$. Since $\lambda > 0$ and either $k_1 \neq 0$ or $k_2 \neq 0$, then $k \neq 0$ and $k' \neq 0$. Thus observe $k_1/k = (\lambda k_1)/k'$ and $k_2/k = (\lambda k_2)/k'$. Then

$$\begin{aligned}
\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) &= \lambda \otimes k\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) \\
&= (\lambda k)\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) \\
&= k'\big((\lambda k_1)/k' \cdot I_1 \sqcup (\lambda k_2)/k' \cdot I_2\big) \\
&= (\lambda k_1) I_1 \sqcup (\lambda k_2) I_2 \\
&= \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2
\end{aligned}$$

* Suppose $\lambda = 0$. Then $\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) = 0[1,1] = 0[1,1] \sqcup 0[1,1] = \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2$.
* Suppose $\lambda < 0$. Let $k = gcd(k_1, k_2)$ and $k' = gcd(-\lambda k_1, -\lambda k_2) = -\lambda k$. Observe $k_1/k = (-\lambda k_1)/k'$ and $k_2/k = (-\lambda k_2)/k'$. Using corollary 3:

$$\begin{aligned}
\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) &= \lambda \otimes k\big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big) \\
&= (-\lambda k)\big(-1 \cdot \big((k_1/k) \cdot I_1 \sqcup (k_2/k) \cdot I_2\big)\big) \\
&= (-\lambda k)\big((k_1/k) \cdot (-1 \cdot I_1) \sqcup (k_2/k) \cdot (-1 \cdot I_2)\big) \\
&= k'\big((-\lambda k_1)/k' \cdot (-1 \cdot I_1) \sqcup (-\lambda k_2)/k' \cdot (-1 \cdot I_2)\big) \\
&= (-\lambda k_1)(-1 \cdot I_1) \sqcup (-\lambda k_2)(-1 \cdot I_2) \\
&= \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2
\end{aligned}$$

□

for corollary 9. By lemma 15 and then corollary 8, $\vec{x}/\vec{y} \otimes \big(\vec{v}/\vec{x} \otimes k_1 I_1 \sqcup \vec{w}/\vec{x} \otimes k_2 I_2\big) = \vec{x}/\vec{y} \otimes (\vec{v}/\vec{x} \otimes k_1 I_1) \sqcup \vec{x}/\vec{y} \otimes (\vec{w}/\vec{x} \otimes k_2 I_2) = \vec{v}/\vec{y} \otimes k_1 I_1 \sqcup \vec{w}/\vec{y} \otimes k_2 I_2$. □

for lemma 16. Since $\emptyset = \gamma(\bot) \subseteq \gamma(r) \subseteq \gamma(\top) = \mathbb{Z}^n$ for all $r \in \mathsf{Range}^n$, it suffices to consider $r_1 = k_1 I_1 \vec{x}$ and $r_2 = k_2 I_2 \vec{y}$.

- Let $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$.
  - Suppose $I_1 = I_2 = [1,1]$ and $k_1\vec{x} = k_2\vec{y}$. Hence $\gamma(k_1 I_1 \vec{x}) = \{k_1\vec{x}\} \subseteq \{k_2\vec{y}\} = \gamma(k_2 I_2 \vec{y})$.
  - Suppose $I_2 \neq [1,1]$, $k_2\vec{y} | k_1\vec{x}$ and $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$. Now since $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \in \mathsf{kInt}$, then by lemma 10 and lemma 7, $\{x(k_1\vec{x})/(k_2\vec{y}) \mid x \in \gamma(I_1)\} = \{x(k_1\vec{x})/(k_2\vec{y}) \mid x \in \gamma(1I_1)\} = \gamma((k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1) \subseteq \gamma(1I_2) = \gamma(I_2)$. Since $k_1 I_1 \vec{x}, k_2 I_2 \vec{y} \in \mathsf{Range}^n$, then $k_1 I_1, k_2 I_2 \in \mathsf{kInt}$, and therefore $\gamma(k_1 I_1 \vec{x}) = \{x\vec{x} \mid x \in \gamma(k_1 I_1)\} = \{xk_1\vec{x} \mid x \in \gamma(I_1)\} \subseteq \{yk_2\vec{y} \mid y \in \gamma(I_2)\} = \{y\vec{y} \mid y \in \gamma(k_2 I_2)\} = \gamma(k_2 I_2 \vec{y})$.
- Let $\gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y})$.
  - Suppose $\gamma(k_2 I_2 \vec{y}) = \{k_2\vec{y}\}$. Thus $I_2 = [1,1]$. Also since $\gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y}) = \{k_2\vec{y}\}$, then $\gamma(k_1 I_1 \vec{x}) = \{k_2\vec{y}\}$. So it must follow that $I_1 = [1,1]$ and so $\gamma(k_1 I_1 \vec{x}) = \{k_1\vec{x}\} = \{k_2\vec{y}\}$. Hence $I_1 = I_2 = [1,1]$ and $k_1\vec{x} = k_2\vec{y}$. Therefore $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$.
  - Suppose $\gamma(k_2 I_2 \vec{y}) \neq \{k_2\vec{y}\}$.
    - To show $I_2 \neq [1,1]$. Since $\gamma(k_2 I_2 \vec{y}) \neq \{k_2\vec{y}\}$, then $|\gamma(k_2 I_2 \vec{y})| > 1$ and thus $I_2 \neq [1,1]$.
    - To show $k_2\vec{y} | k_1\vec{x}$.
      - Suppose $|\gamma(k_1 I_1 \vec{x})| = 1$. Then $I_1 = [1,1]$. Observe $\{k_1\vec{x}\} = \gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y})$. Thus $k_1\vec{x} = yk_2\vec{y}$ for some $y \in \gamma(I_2)$. Thus $k_2\vec{y} | k_1\vec{x}$.
      - Suppose $|\gamma(k_1 I_1 \vec{x})| > 1$. Then $|I_1| > 1$. Let $I_1 = [\ell_1, u_1]$ where $\ell_1 < u_1$. Thus $\ell_1 < \ell_1 + 1 \leq u_1$ follows. Observe $\{k_1\ell_1\vec{x}, k_1(\ell_1+1)\vec{x}\} \subseteq \gamma(k_1 I_1 \vec{x})$. Since $\gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y})$, it follows $k_1\ell_1\vec{x} = k_2 v\vec{y}$ and likewise $k_1(\ell_1+1)\vec{x} = k_2 w\vec{y}$ for some $v, w \in \gamma(I_2)$. Therefore $k_1\vec{x} = k_1((\ell_1+1) - \ell_1)\vec{x} = k_2(w-v)\vec{y}$. Thus $k_2\vec{y} | k_1\vec{x}$.
    - To show $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$. Since $I_2 \neq [1,1]$, then $k_2 \neq 0$ so $k_2\vec{y} \neq \vec{0}$. Observe $\{xk_1\vec{x} \mid x \in \gamma(I_1)\} = \gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y}) = \{yk_2\vec{y} \mid y \in \gamma(I_2)\}$. Since $k_2\vec{y} | k_1\vec{x}$ and $k_2\vec{y} \neq \vec{0}$, then $(k_1\vec{x})/(k_2\vec{y}) \in$

$\mathbb{Z}$. Hence $\gamma((k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1) = \{x(k_1\vec{x})/(k_2\vec{y}) \mid x \in \gamma(1I_1)\} = \{x(k_1\vec{x})/(k_2\vec{y}) \mid x \in \gamma(I_1)\} \subseteq \{y(k_2\vec{y})/(k_2\vec{y}) \mid y \in \gamma(I_2)\} = \{y \mid y \in \gamma(1I_2)\} = \gamma(1I_2)$

□

for lemma 17. If $|\gamma(kI\vec{v})| = 1$, then $|\{x\vec{v} \mid x \in \gamma(kI)\}| = 1$, thus $|\gamma(kI)| = 1$, hence by lemma 9, $I = [1, 1]$. □

for lemma 18. Let $k_1I_1, k_2I_2 \in \mathsf{kInt}$ and $\vec{x} \in \mathbb{Z}^n \setminus \{\vec{0}\}$.

- Let $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{x}$. To show $k_1I_1 \sqsubseteq k_2I_2$.
  - Suppose $I_1 = I_2 = [1, 1]$ and $k_1\vec{x} = k_2\vec{x}$. Then $k_1 = k_2$, thus $k_2|k_1$ and $k_1 \cdot I_1 \sqsubseteq k_1 \cdot I_1 = k_2 \cdot I_2$. Therefore $k_1I_1 \sqsubseteq k_2I_2$.
  - Suppose $I_2 \neq [1, 1]$, $k_2\vec{x}|k_1\vec{x}$ and $(k_1\vec{x})/(k_2\vec{x}) \otimes 1I_2 \sqsubseteq 1I_2$. Since $k_2\vec{x}|k_1\vec{x}$, then $\lambda k_2\vec{x} = k_1\vec{x}$ for some $\lambda \in \mathbb{Z}$. Thus $\lambda k_2 = k_1$, so $k_2|k_1$. Since $(k_1\vec{x})/(k_2\vec{x}) \otimes 1I_2 \sqsubseteq 1I_2$, then $k_1/k_2 \otimes 1I_1 \sqsubseteq 1I_2$ and thus by lemma 12, $k_1/k_2 \cdot I_1 \sqsubseteq 1 \cdot I_2$.
- Let $k_1I_1 \sqsubseteq k_2I_2$. To show $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{x}$. By lemma 7, $\gamma(k_1I_1) \subseteq \gamma(k_2I_2)$. Then $\gamma(k_1I_1\vec{x}) \subseteq \gamma(k_2I_2\vec{x})$. By lemma 16, $k_1I_1\vec{x} \sqsubseteq k_2I_2\vec{x}$ holds.

□

for lemma 19. Suppose $I = [\ell, u]$ where $\ell \leq u$. Observe $\gamma(k(\lambda + I)\vec{v}) = \{x\vec{v} \mid x \in \gamma(k(\lambda + I))\} = \{kx\vec{v} \mid x \in \gamma(\lambda + I)\} = \{kx\vec{v} \mid \lambda + \ell \leq x \leq \lambda + u\} = \{k(\lambda + x)\vec{v} \mid \ell \leq x \leq u\} = \{\lambda k\vec{v} + kx\vec{v} \mid \ell \leq x \leq u\} = \{\lambda k\vec{v} + kx\vec{v} \mid x \in \gamma(I)\} = \{\lambda k\vec{v} + x\vec{v} \mid x \in \gamma(kI)\} = \{\lambda k\vec{v} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$. □

for corollary 10. Since $k\vec{v}|\vec{t}$ and $k\vec{v} \neq \vec{0}$, then $\lambda = \vec{t}/(k\vec{v})$ for some $\lambda \in \mathbb{Z}$. Thus by lemma 19, $\gamma(k(\vec{t}/(k\vec{v}) + I)\vec{v}) = \{(\vec{t}/(k\vec{v}))k\vec{v} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\} = \{\vec{t} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$. □

for lemma 20. Since $\vec{z}|k\vec{v}$ and $\vec{z} \neq \vec{0}$ then $\lambda = (k\vec{v})/\vec{z}$ for some $\lambda \in \mathbb{Z}$. Since both $\lambda \in \mathbb{Z}$ and $1I \in \mathsf{kInt}$, then by lemma 10, $\gamma(kI\vec{v}) = \{v\vec{v} \mid v \in \gamma(kI)\} = \{v(k\vec{v}) \mid v \in \gamma(1I)\} = \{v((k\vec{v})/\vec{z})\vec{z} \mid v \in \gamma(1I)\} = \{v\vec{z} \mid v \in \gamma((k\vec{v})/\vec{z} \otimes 1I)\} = \gamma(((k\vec{v})/\vec{z} \otimes 1I)\vec{z})$. Hence by lemma 16, $kI\vec{v} \equiv ((k\vec{v})/\vec{z} \otimes 1I)\vec{z}$. □

for corollary 11.

- Since $kI(\lambda\vec{v}) \in \mathsf{Range}^n$ then $\lambda\vec{v} \neq \vec{0}$ and so both $\lambda \neq 0$ and $\vec{v} \neq \vec{0}$. Observe $\vec{v}|k(\lambda\vec{v})$. By lemma 20 and then by lemma 11, $kI(\lambda\vec{v}) \equiv ((k(\lambda\vec{v}))/\vec{v} \otimes 1I)\vec{v} = (k\lambda \otimes 1I)\vec{v} = (\lambda \otimes kI)\vec{v}$ since $kI \in \mathsf{kInt}$.
- Let $\vec{z} = k\vec{v}$. Since $k \neq 0$, then $\vec{z} \neq \vec{0}$. By lemma 20, $kI\vec{v} \equiv ((k\vec{v})/(k\vec{v}) \otimes 1I)(k\vec{v}) = (1 \otimes 1I)(k\vec{v}) = 1I(k\vec{v})$.

□

for lemma 21. Let $kI(\lambda\vec{v}) \in \mathsf{Range}^n$ where $\lambda \in \mathbb{Z}$.

- To show $kI(\lambda\vec{v}) \sqsubseteq k(\lambda \cdot I)\vec{v}$.
  - To show $(\lambda \otimes kI) \sqsubseteq k(\lambda \cdot I)$.
    - Suppose $I = [1, 1]$ or $\lambda > 0$. By lemma 5, $(\lambda k)\cdot I \sqsubseteq (\lambda k)\cdot I = k\cdot(\lambda\cdot I)$ and because $k|(\lambda k)$, then by definition 10, $(\lambda k)I \sqsubseteq k(\lambda \cdot I)$. Hence by definition 11, $(\lambda \otimes kI) = (\lambda k)I \sqsubseteq k(\lambda \cdot I)$.
    - Suppose $\lambda = 0$. Then $\lambda\vec{v} = \vec{0}$, so $kI(\lambda\vec{v}) \notin \mathsf{Range}^n$, contradicting the supposition.
    - Suppose otherwise. By two applications of lemma 5, $(-\lambda k) \cdot (-1 \cdot I) \sqsubseteq (-\lambda k) \cdot (-1 \cdot I) = k \cdot (-\lambda \cdot (-1 \cdot I)) = k \cdot (\lambda \cdot I)$, and since $k|(-\lambda k)$, then by definition 10, $(-\lambda k)(-1 \cdot I) \sqsubseteq k(\lambda \cdot I)$. Hence by definition 11, $(\lambda \otimes kI) = (-\lambda k)(-1 \cdot I) \sqsubseteq k(\lambda \cdot I)$.

    Thus $(\lambda \otimes kI) \sqsubseteq k(\lambda \cdot I)$. Hence by corollary 11 and lemma 18, $kI(\lambda\vec{v}) \equiv (\lambda \otimes kI)\vec{v} \sqsubseteq k(\lambda \cdot I)\vec{v}$.
- To show $k(\lambda \cdot I)\vec{v} \sqsubseteq 1(k\lambda \cdot I)\vec{v}$.
  - By corollary 4 and then 5, $k(\lambda \cdot I) \sqsubseteq 1(k \cdot (\lambda \cdot I)) = 1(k\lambda \cdot I)$. Since $\lambda\vec{v} \neq \vec{0}$ and so $\vec{v} \neq \vec{0}$, thus by lemma 18, $k(\lambda \cdot I)\vec{v} \sqsubseteq 1(k\lambda \cdot I)\vec{v}$.

By transitivity, $kI(\lambda\vec{v}) \sqsubseteq k(\lambda \cdot I)\vec{v} \sqsubseteq 1(k\lambda \cdot I)\vec{v}$. □

for corollary 12.

- $1I(\lambda\vec{v}) \sqsubseteq 1(\lambda \cdot I)\vec{v}$. The proof is immediate with $k = 1$.

- $kI\vec{v} \sqsubseteq 1(k \cdot I)\vec{v}$. The proof is immediate with $\lambda = 1$.

□

for lemma 22. By supposition $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$ where $I_1 \neq [1,1]$. Thus by definition 13, $I_2 \neq [1,1]$, $k_2\vec{y} | k_1\vec{x}$ and $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$. To show $k_1 I_1 \vec{x} \sqsubseteq 1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z} \sqsubseteq k_2 I_2 \vec{y}$, it is necessary to show both $k_1 I_1 \vec{x} \sqsubseteq 1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z}$ and $1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z} \sqsubseteq k_2 I_2 \vec{y}$.

- First, to show $k_1 I_1 \vec{x} \sqsubseteq 1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z}$.
  - To show $(k_1\vec{x})/\vec{z} \cdot I_1 \neq [1,1]$. Since $I_1 \neq [1,1]$, this follows immediately.
  - To show $1\vec{z} | k_1\vec{x}$. This holds by supposition, since $1\vec{z} = \vec{z}$.
  - To show $(k_1\vec{x})/(1\vec{z}) \otimes 1I_1 \sqsubseteq 1((k_1\vec{x})/\vec{z} \cdot I_1)$. Since $1\vec{z} = \vec{z}$, then this follows immediately from corollary 5, where $\lambda = (k_1\vec{x})/\vec{z}$.
- Second, to show $1((k_1\vec{x})/\vec{z} \cdot I_1)\vec{z} \sqsubseteq k_2 I_2 \vec{y}$.
  - To show $I_2 \neq [1,1]$. Since $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$ and $I_1 \neq [1,1]$, it must follow by lemma 16 that $I_2 \neq [1,1]$.
  - To show $k_2\vec{y} | 1\vec{z}$. This holds from supposition, since $1\vec{z} = \vec{z}$.
  - To show $(1\vec{z})/(k_2\vec{y}) \otimes 1((k_1\vec{x})/\vec{z} \cdot I_1) \sqsubseteq 1I_2$. By corollary 5, this reduces to showing $(1\vec{z})/(k_2\vec{y}) \cdot ((k_1\vec{x})/\vec{z} \cdot I_1) \sqsubseteq 1 \cdot I_2$. Let $a = \vec{z}/(k_2\vec{y})$ and $b = (k_1\vec{x})/\vec{z}$. Then, by lemma 5, this further reduces to showing $(k_1\vec{x})/(k_2\vec{y}) \cdot I_1 \sqsubseteq 1 \cdot I_2$. But it has been shown that $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$. Hence, by corollary 5, $(k_1\vec{x})/(k_2\vec{y}) \cdot I_1 \sqsubseteq 1 \cdot I_2$ and the result follows.

□

for proposition 7. Let $k_1 I_1 \vec{x}, k_2 I_2 \vec{y} \in \mathsf{Range}^n$. To show $k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$ is a least upper bound.

- To show $k_1 I_1 \vec{x} \sqsubseteq k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$.
  - Suppose $I_1 = I_2 = [1,1]$ and $k_1\vec{x} = k_2\vec{y}$. Then $k_1 I_1 \vec{x} \sqsubseteq k_1 I_1 \vec{x} = k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$.

- Suppose $\vec{z} = gcd(k_1\vec{x}, k_2\vec{y}) \neq \bot$. Then $\vec{z}|k_1\vec{x}$ and by the reasoning immediately following proposition 7, $\vec{z} \neq \vec{0}$. Thus by lemma 20, $k_1 I_1 \vec{x} \equiv ((k_1\vec{x})/\vec{z} \otimes 1I_1)\vec{z}$. Likewise $\vec{z}|k_2\vec{y}$ and so $k_2 I_2 \vec{y} \equiv ((k_2\vec{y})/\vec{z} \otimes 1I_2)\vec{z}$. By proposition 5, $(k_1\vec{x})/\vec{z} \otimes 1I_1 \sqsubseteq (k_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1I_2$. Hence by lemma 18, $k_1 I_1 \vec{x} \equiv ((k_1\vec{x})/\vec{z} \otimes 1I_1)\vec{z} \sqsubseteq \big((k_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1I_2\big)\vec{z} = k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$.
- Suppose otherwise. Then $k_1 I_1 \vec{x} \sqsubseteq \top = k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$.

- To show $k_2 I_2 \vec{y} \sqsubseteq k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y}$ is analogous to above.

- Let $k_1 I_1 \vec{x} \sqsubseteq kI\vec{v}$ and $k_2 I_2 \vec{y} \sqsubseteq kI\vec{v}$. To show $k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y} \sqsubseteq kI\vec{v}$.
  - Suppose $kI\vec{v} = \bot$. Then $k_1 I_1 \vec{x} = k_2 I_2 \vec{y} = \bot$. Thus $k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y} = \bot \sqsubseteq kI\vec{v}$.
  - Suppose $kI\vec{v} = \top$. Thus $k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y} \sqsubseteq kI\vec{v}$.
  - Suppose $kI\vec{v} \in \mathsf{Range}^n \setminus \{\bot, \top\}$.
    - Suppose both $I_1 = I = [1, 1]$ and $k_1\vec{x} = k\vec{v}$, and $I_2 = I = [1, 1]$ and $k_2\vec{y} = k\vec{v}$. Then $k_1 I_1 \vec{x} \sqcup k_2 I_2 \vec{y} = k_1 I_1 \vec{x} \sqsubseteq kI\vec{v}$.
    - Suppose $I \neq [1, 1]$ and both $k\vec{v}|k_1\vec{x}$ and $(k_1\vec{x})/(k\vec{v}) \otimes 1I_1 \sqsubseteq 1I$, and $k\vec{v}|k_2\vec{y}$ and $(k_2\vec{y})/(k\vec{v}) \otimes 1I_2 \sqsubseteq 1I$. Let $\ell J\vec{w} = \big((k_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1I_2\big)\vec{z}$ where $\vec{z} = gcd(k_1\vec{x}, k_2\vec{y})$, thus $\ell J = \big((k_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1I_2\big)$ and $\vec{w} = \vec{z}$.
      - To show $k\vec{v}|\ell\vec{w}$. Since $k\vec{v}|k_1\vec{x}$ and $k\vec{v}|k_2\vec{y}$, then by lemma 2, $k\vec{v}|gcd(k_1\vec{x}, k_2\vec{y}) = \vec{z}$. Since $\ell \in \mathbb{Z}$, then $k\vec{v}|\ell\vec{z}$. But $\vec{z} = \vec{w}$, so $\ell\vec{z} = \ell\vec{w}$, hence $k\vec{v}|\ell\vec{w}$.
      - To show $(\ell\vec{w})/(k\vec{v}) \otimes 1J \sqsubseteq 1I$. Since $k\vec{v}|\vec{z}$, then $a(k\vec{v}) = \vec{z}$ for some $a \in \mathbb{Z}$ and by the reasoning immediately following proposition 7, $\vec{z} \neq 0$, thus $a \neq 0$ and $k\vec{v} \neq \vec{0}$, so $k\vec{v} = \vec{z}/a$ and $a = \vec{z}/(k\vec{v})$. Since $(k_1\vec{x})/(k\vec{v}) \otimes 1I_1 \sqsubseteq 1I$ and $(k_2\vec{y})/(k\vec{v}) \otimes 1I_2 \sqsubseteq 1I$, then by proposition 5, $(k_1\vec{x})/(k\vec{v}) \otimes 1I_1 \sqcup (k_2\vec{y})/(k\vec{v}) \otimes 1I_2 \sqsubseteq 1I$. Substituting in $k\vec{v} = \vec{z}/a$ gives $(ak_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (ak_2\vec{y})/\vec{z} \otimes 1I_2 \sqsubseteq 1I$. Then by definition of ℓJ, by lemma 15 and lemma 14, $a \otimes \ell J = a \otimes \big((k_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (k_2\vec{y})/\vec{z} \otimes 1I_2\big) = \big(a \otimes ((k_1\vec{x})/\vec{z} \otimes$

$1I_1)) \sqcup \big(a \otimes ((k_2\vec{y})/\vec{z} \otimes 1I_2)\big) = (ak_1\vec{x})/\vec{z} \otimes 1I_1 \sqcup (ak_2\vec{y})/\vec{z} \otimes 1I_2 \sqsubseteq 1I$. Substituting back in $a = \vec{z}/(k\vec{v})$ and by lemma 11, $(\ell\vec{z})/(k\vec{v}) \otimes 1J = a\ell \otimes 1J = a \otimes \ell J \sqsubseteq 1I$. But $\vec{z} = \vec{w}$, so the result follows.

□

for lemma 23. Since $\emptyset = \gamma(\bot) \subseteq \gamma(m) \subseteq \gamma(\top) = \mathbb{Z}^n$ for all $m \in \mathsf{March}^n$, it suffices to consider $m_1 = \vec{s}_1 + k_1 I_1 \vec{x}$ and $m_2 = \vec{s}_2 + k_2 I_2 \vec{y}$.

- Let $\vec{s}_1 + k_1 I_1 \vec{x} \sqsubseteq \vec{s}_2 + k_2 I_2 \vec{y}$
  - Suppose $I_1 = I_2 = [1,1]$ and $\vec{s}_1 + k_1\vec{x} = \vec{s}_2 + k_2\vec{y}$. Since $I_1 = I_2 = [1,1]$, then $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) = \{\vec{s}_1 + k_1\vec{x}\} \subseteq \{\vec{s}_2 + k_2\vec{y}\} = \gamma(\vec{s}_2 + k_2 I_2 \vec{y})$.
  - Suppose $I_1 = [1,1]$, $I_2 \neq [1,1]$, $k_2\vec{y} | (\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}))$ and $[0,0] \sqsubseteq (\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}))/(k_2\vec{y}) + I_2$. By lemma 3, $\{0\} \subseteq \{y \mid y \in \gamma((\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}))/(k_2\vec{y}) + I_2)\} = \{(\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}))/(k_2\vec{y}) + y \mid y \in \gamma(I_2)\}$ and thus $\{\vec{0}\} \subseteq \{\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}) + yk_2\vec{y} \mid y \in \gamma(I_2)\}$. Bringing $\vec{s}_1 + k_1\vec{x}$ across and noting $I_1 = [1,1]$ gives $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) = \{\vec{s}_1 + k_1\vec{x}\} \subseteq \{\vec{s}_2 + \vec{v} \mid \vec{v} \in \gamma(k_2 I_2 \vec{y})\} = \gamma(\vec{s}_2 + k_2 I_2 \vec{y})$.
  - Suppose $I_2 \neq [1,1]$, $k_2\vec{y} | (\vec{s}_2 - \vec{s}_1)$ and $k_1 I_1 \vec{x} \sqsubseteq k_2((\vec{s}_2 - \vec{s}_1)/(k_2\vec{y}) + I_2)\vec{y}$. Thus by lemma 16 and then by corollary 10, $\{\vec{x}' \mid \vec{x}' \in \gamma(k_1 I_1 \vec{x})\} = \gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2((\vec{s}_2 - \vec{s}_1)/(k_2\vec{y}) + I_2)\vec{y}) = \{(\vec{s}_2 - \vec{s}_1) + \vec{y}' \mid \vec{y}' \in \gamma(k_2 I_2 \vec{y})\}$. Hence $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) = \{\vec{s}_1 + \vec{x}' \mid \vec{x}' \in \gamma(k_1 I_1 \vec{x})\} \subseteq \{\vec{s}_2 + \vec{y}' \mid \vec{y}' \in \gamma(k_2 I_2 \vec{y})\} = \gamma(\vec{s}_2 + k_2 I_2 \vec{y})$.
- Let $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2 \vec{y})$.
  - Suppose $|\gamma(\vec{s}_2 + k_2 I_2 \vec{y})| = 1$. Then $I_2 = [1,1]$ and thus $\gamma(\vec{s}_2 + k_2 I_2 \vec{y}) = \{\vec{s}_2 + k_2\vec{y}\}$. Since $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2 \vec{y}) = \{\vec{s}_2 + k_2\vec{y}\}$ then it must hold that $I_1 = [1,1]$ and so $\{\vec{s}_1 + k_1\vec{x}\} = \gamma(\vec{s}_1 + k_1 I_1 \vec{x}) \subseteq \{\vec{s}_2 + k_2\vec{y}\}$. Thus $I_1 = I_2 = [1,1]$ and $\vec{s}_1 + k_1\vec{x} = \vec{s}_2 + k_2\vec{y}$.
  - Suppose $|\gamma(\vec{s}_2 + k_2 I_2 \vec{y})| > 1$
    - To show $I_2 \neq [1,1]$. This holds since $|\gamma(\vec{s}_2 + k_2 I_2 \vec{y})| > 1$.

* To show $k_2\vec{y}|(\vec{s}_2 - \vec{s}_1)$. Let $\vec{s}_1 + k_1 x\vec{x} = \vec{p}_1 \in \gamma(\vec{s}_1 + k_1 I_1\vec{x})$ for some $x \in \gamma(I_1)$. Since $\vec{p}_1 \in \gamma(\vec{s}_1 + k_1 I_1\vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2\vec{y})$, then $\vec{s}_1 + k_1 x\vec{x} = \vec{p}_1 = \vec{s}_2 + k_2 y\vec{y}$ for some $y \in \gamma(I_2)$.
  - Suppose $|I_1| = 1$. Since $\{\vec{s}_1 + k_1\vec{x}\} = \gamma(\vec{s}_1 + k_1 I_1\vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2\vec{y}) = \{\vec{s}_2 + yk_2\vec{y} \mid y \in \gamma(I_2)\}$, then $\{\vec{s}_1 + k_1\vec{x} - \vec{s}_2\} = \{yk_2\vec{y} \mid y \in \gamma(I_2)\}$. Thus $\vec{s}_1 + k_1\vec{x} - \vec{s}_2 = yk_2\vec{y}$ for some $y \in I_2$ and $\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}) = -(\vec{s}_1 + k_1\vec{x} - \vec{s}_2) = (-y)k_2\vec{y}$ where $-y \in \mathbb{Z}$. Thus $k_2\vec{y}|(\vec{s}_2 - (\vec{s}_1 + k_1\vec{x}))$.
  - Suppose $|I_1| > 1$. Then $x+\delta \in \gamma(I_1)$ for some fixed $\delta \in \{-1, 1\}$ and thus $\vec{s}_1 + k_1(x+\delta)\vec{x} \in \gamma(\vec{s}_1 + k_1 I_1\vec{x})$. Let $\vec{p}_2 = \vec{s}_1 + k_1(x+\delta)\vec{x}$. Since $\vec{p}_2 \in \gamma(\vec{s}_1 + k_1 I_1\vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2\vec{y})$, then $\vec{p}_2 = \vec{s}_2 + k_2(y + \lambda)\vec{y}$ for some $\lambda \in \mathbb{Z}$. Observe $\delta k_1\vec{x} = k_1(x + \delta)\vec{x} - k_1 x\vec{x} = \vec{p}_2 - \vec{p}_1 = k_2(y + \lambda)\vec{y} - k_2 y\vec{y} = \lambda k_2\vec{y}$. Hence $\delta k_1\vec{x} = \lambda k_2\vec{y}$. Since $\delta \in \{-1, 1\}$, it follows $k_1\vec{x} = \delta^2 k_1\vec{x} = \delta\lambda k_2\vec{y}$ and because $\vec{s}_1 + k_1 x\vec{x} = \vec{p}_1 = \vec{s}_2 + k_2 y\vec{y}$, then $\vec{s}_2 - \vec{s}_1 = (\delta\lambda k_2\vec{y})x - k_2 y\vec{y} = k_2(\delta\lambda x - y)\vec{y} = \mu k_2\vec{y}$ where $\mu = (\delta\lambda x - y) \in \mathbb{Z}$. Hence $k_2\vec{y}|(\vec{s}_2 - \vec{s}_1)$.
* To show $k_1 I_1\vec{x} \sqsubseteq k_2((\vec{s}_2 - \vec{s}_1)/(k_2\vec{y}) + I_2)\vec{y}$. Since $\{\vec{s}_1 + \vec{x}' \mid \vec{x}' \in \gamma(k_1 I_1\vec{x})\} = \gamma(\vec{s}_1 + k_1 I_1\vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2\vec{y}) = \{\vec{s}_2 + \vec{y}' \mid \vec{y}' \in \gamma(k_2 I_2\vec{y})\}$, then $\gamma(k_1 I_1\vec{x}) = \{\vec{x}' \mid \vec{x}' \in \gamma(k_1 I_1\vec{x})\} \subseteq \{(\vec{s}_2 - \vec{s}_1) + \vec{y}' \mid \vec{y}' \in \gamma(k_2 I_2\vec{y})\}$. Since $I_2 \neq [1, 1]$, then $k_2\vec{y} \neq \vec{0}$ and since $k_2\vec{y}|(\vec{s}_2 - \vec{s}_1)$, then by corollary 10, $\gamma(k_1 I_1\vec{x}) \subseteq \{(\vec{s}_2 - \vec{s}_1) + \vec{y}' \mid \vec{y}' \in \gamma(k_2 I_2\vec{y})\} = \gamma(k_2((\vec{s}_2 - \vec{s}_1)/(k_2\vec{y}) + I_2)\vec{y})$. Thus, by lemma 16, $k_1 I_1\vec{x} \sqsubseteq k_2((\vec{s}_2 - \vec{s}_1)/(k_2\vec{y}) + I_2)\vec{y}$.

□

for corollary 13. If $\vec{s}_1 + k_1 I_1\vec{x} \sqsubseteq \vec{s}_2 + k_2 I_2\vec{y}$, then $\{\vec{s}_1 + x\vec{x} \mid x \in \gamma(k_1 I_1)\} = \gamma(\vec{s}_1 + k_1 I_1\vec{x}) \subseteq \gamma(\vec{s}_2 + k_2 I_2\vec{y}) = \{\vec{s}_2 + y\vec{y} \mid y \in \gamma(k_2 I_2)\}$. Hence $\gamma((\vec{s}_1 + \vec{s}) + k_1 I_1\vec{x}) = \{\vec{s}_1 + \vec{s} + x\vec{x} \mid x \in \gamma(k_1 I_1)\} \subseteq \{\vec{s}_2 + \vec{s} + y\vec{y} \mid y \in \gamma(k_2 I_2)\} = \gamma((\vec{s}_2 + \vec{s}) + k_2 I_2\vec{y})$. Hence by lemma 23, $(\vec{s}_1 + \vec{s}) + k_1 I_1\vec{x} \sqsubseteq (\vec{s}_2 + \vec{s}) + k_2 I_2\vec{y}$. □

for lemma 24. If $|\gamma(\vec{s} + kI\vec{v})| = 1$, then $|\{\vec{s} + x\vec{v} \mid x \in \gamma(kI\vec{v})\}| = 1$, thus $|\gamma(kI\vec{v})| = 1$ which, by lemma 17, also implies $I = [1, 1]$. □

for lemma 25. Let $\lambda \in \mathbb{Z}$. Since $I \neq [1,1]$, then $k(\lambda + I)\vec{v} \in \mathsf{Range}^n$ and therefore $(\vec{s} - \lambda k\vec{v}) + k(\lambda + I)\vec{v} \in \mathsf{March}^n$. Observe by lemma 23 and by lemma 19, $\gamma((\vec{s} - \lambda k\vec{v}) + k(\lambda + I)\vec{v}) = \{(\vec{s} - \lambda k\vec{v}) + \vec{x} \mid \vec{x} \in \gamma(k(\lambda + I)\vec{v})\} = \{(\vec{s} - \lambda k\vec{v}) + \lambda k\vec{v} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\} = \{\vec{s} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\} = \gamma(\vec{s} + kI\vec{v})$. Hence by lemma 23, $(\vec{s} - \lambda k\vec{v}) + k(\lambda + I)\vec{v} \equiv \vec{s} + kI\vec{v}$. □

for corollary 14.

1. Since $k\vec{v}|\vec{u}$ then $\lambda k\vec{v} = \vec{u}$ for some $\lambda \in \mathbb{Z}$. Also since $I \neq [1,1]$, then $k \neq 0$ and so $k\vec{v} \neq \vec{0}$. Thus $\lambda = \vec{u}/(k\vec{v})$. From lemma 25, $\vec{s} + kI\vec{v} \equiv (\vec{s} - \lambda k\vec{v}) + k(\lambda + I)\vec{v} = (\vec{s} - (\vec{u}/(k\vec{v}))k\vec{v}) + k(\vec{u}/(k\vec{v}) + I)\vec{v} = (\vec{s} - \vec{u}) + k(\vec{u}/(k\vec{v}) + I)\vec{v}$.
2. Put $\vec{u} = k\vec{v}$ into the first case.
3. Put $\vec{u} = -k\vec{v}$ into the first case.
4. Put $\vec{u} = \vec{s} - \vec{t}$ into the first case.

Special cases:

1. Put $\vec{s} = \vec{0}$ into the first case.
   - Put $k = 1$ into the preceeding special case.
2. Put $\vec{u} = k\vec{v}$ into the first special case.
   - Put $k = 1$ into the preceeding special case.
3. Put $\vec{u} = -k\vec{v}$ into the first special case.
   - Put $k = 1$ into the preceeding special case.

□

for lemma 26.

- Let $\vec{s}_1 + k_1 I_1 \vec{x} \sqsubseteq \vec{s}_1 + k_2 I_2 \vec{y}$. Observe $\{\vec{s}_1 + x\vec{x} \mid x \in \gamma(k_1 I_1)\} = \gamma(\vec{s}_1 + k_1 I_1 \vec{x}) \subseteq \gamma(\vec{s}_1 + k_2 I_2 \vec{y}) = \{\vec{s}_1 + y\vec{y} \mid y \in \gamma(k_2 I_2)\}$. Thus $\gamma(k_1 I_1 \vec{x}) = \{x\vec{x} \mid x \in \gamma(k_1 I_1)\} \subseteq \{y\vec{y} \mid y \in \gamma(k_2 I_2)\} = \gamma(k_2 I_2 \vec{y})$. Hence, by lemma 16, $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$.

- Let $k_1 I_1 \vec{x} \sqsubseteq k_2 I_2 \vec{y}$. Observe $\{x\vec{x} \mid x \in \gamma(k_1 I_1)\} = \gamma(k_1 I_1 \vec{x}) \subseteq \gamma(k_2 I_2 \vec{y}) = \{y\vec{y} \mid y \in \gamma(k_2 I_2)\}$. Thus $\gamma(\vec{s}_1 + k_1 I_1 \vec{x}) = \{\vec{s}_1 + x\vec{x} \mid x \in \gamma(k_1 I_1)\} \subseteq \{\vec{s}_1 + y\vec{y} \mid y \in \gamma(k_2 I_2)\} = \gamma(\vec{s}_2 + k_2 I_2 \vec{y})$. Hence, by lemma 16, $\vec{s}_1 + k_1 I_1 \vec{x} \sqsubseteq \vec{s}_1 + k_2 I_2 \vec{y}$.

□

for lemma 27. Observe $\gamma(\vec{s} + k[1,1]\vec{v}) = \{\vec{s} + k\vec{v}\} = \gamma((\vec{s} + k\vec{v}) + 0[1,1]\vec{1})$. □

for lemma 28. To show $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma((\vec{s} + kI\vec{v})^{-1})$.

- Suppose $I = [1,1]$ and $\vec{s} + k\vec{v} = \vec{0}$. Then $\gamma(\vec{s} + kI\vec{v}) = \{\vec{0}\} = \gamma(0[1,1]\vec{1}) = \gamma((\vec{s} + kI\vec{v})^{-1})$.
- Suppose $I = [1,1]$ and $\vec{s} + k\vec{v} \neq \vec{0}$. Then $\gamma(\vec{s} + kI\vec{v}) = \{\vec{s} + k\vec{v}\} = \gamma(1[1,1](\vec{s} + k\vec{v})) = \gamma((\vec{s} + kI\vec{v})^{-1})$.
- Suppose $\vec{z} = gcd(\vec{s}, k\vec{v}) \neq \bot$. Then $\lambda\vec{z} = \vec{s}$ and $\mu\vec{z} = k\vec{v}$ for some $\lambda, \mu \in \mathbb{Z}$. Since $I \neq [1,1]$, then $k \neq 0$ and $k\vec{v} \neq \vec{0}$ and thus $\vec{z} \neq \vec{0}$, and so by corollary 11 and then by corollary 12, $kI\vec{v} \equiv 1I(k\vec{v}) = 1I(\mu\vec{z}) \sqsubseteq 1(\mu \cdot I)\vec{z} = 1((k\vec{v})/\vec{z} \cdot I)\vec{z}$. Thus by lemma 26, and since $\vec{z} | \vec{s}$, then by special case 1 of corollary 14, it follows $\vec{0} + kI\vec{v} \sqsubseteq \vec{0} + 1((k\vec{v})/\vec{z} \cdot I)\vec{z} \equiv -\vec{s} + 1(\vec{s}/\vec{z} + (k\vec{v})/\vec{z} \cdot I)\vec{z}$. By corollary 13, it follows $\vec{s} + kI\vec{v} \sqsubseteq \vec{0} + 1(\vec{s}/\vec{z} + (k\vec{v}/\vec{z}) \cdot I)\vec{z}$. Hence by lemma 23, $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(\vec{0} + 1(\vec{s}/\vec{z} + (k\vec{v}/\vec{z}) \cdot I)\vec{z}) = \gamma((\vec{s} + kI\vec{v})^{-1})$.
- Suppose otherwise. Then $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(\top) = \gamma((\vec{s} + kI\vec{v})^{-1})$.

Let $r \in \mathsf{Range}^n$ such that $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(r)$. To show $\gamma((\vec{s} + kI\vec{v})^{-1}) \subseteq \gamma(r)$.

- Suppose $I = [1,1]$ and $\vec{s} + k\vec{v} = \vec{0}$. Then $\gamma((\vec{s} + kI\vec{v})^{-1}) = \gamma(0[1,1]\vec{1}) = \{\vec{0}\} = \gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(r)$.
- Suppose $I = [1,1]$ and $\vec{s} + k\vec{v} \neq \vec{0}$. Then $\gamma((s + kI\vec{v})^{-1}) = \gamma(1[1,1](\vec{s} + k\vec{v})) = \{\vec{s} + k\vec{v}\} = \gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(r)$.
- Suppose $\vec{z} = gcd(\vec{s}, k\vec{v}) \neq \bot$. Thus $\lambda\vec{z} = \vec{s}$ and $\mu\vec{z} = k\vec{v}$ for some $\lambda, \mu \in \mathbb{Z}$. Suppose $r = k'I'\vec{v}'$. Since $\gamma(\vec{s} + kI\vec{v}) \subseteq \gamma(r) = \gamma(r^{+1})$, then by lemma 23, $\vec{s} + kI\vec{v} \sqsubseteq \vec{0} + k'I'\vec{v}'$. Observe since $I \neq [1,1]$, then $I' \neq [1,1]$. Now by definition 15, $k'\vec{v}' | -\vec{s}$, so by special case 1 of corollary 14 (with $-\vec{s}$ in place

of $\vec{u}$), $\vec{s}+kI\vec{v} \sqsubseteq \vec{0}+k'I'\vec{v}' \equiv \vec{s}+k'(-\vec{s}/(k'\vec{v}')+I')\vec{v}'$. Thus by lemma 26, $kI\vec{v} \sqsubseteq k'(-\vec{s}/(k'\vec{v}')+I')\vec{v}'$ and so by definition 13, $k'\vec{v}'|k\vec{v}$. Furthermore, since $k'\vec{v}'|-\vec{s}$, then $k'\vec{v}'|\vec{z}$. Additionally, by definition of gcd, $\vec{z}|k\vec{v}$ and since $I \neq [1,1]$, then by lemma 22, $1((k\vec{v})/\vec{z}\cdot I)\vec{z} \sqsubseteq k'(-\vec{s}/(k'\vec{v}')+I')\vec{v}'$. By lemma 26 and then by special case 1 of corollary 14 (with $\vec{u}$ brought across and $-\vec{s}$ replacing $\vec{u}$), $\vec{0}+1((k\vec{v})/\vec{z}\cdot I)\vec{z} \sqsubseteq \vec{0}+k'(-\vec{s}/(k'\vec{v}')+I')\vec{v}' \equiv -\vec{s}+k'I'\vec{v}'$. Thus by corollary 13, $\vec{s}+1((k\vec{v})/\vec{z}\cdot I) \sqsubseteq \vec{0}+k'I'\vec{v}'$. Hence by lemma 23, $\gamma(\vec{s}+1((k\vec{v})/\vec{z}\cdot I)) \subseteq \gamma(\vec{0}+k'I'\vec{v}') = \gamma(k'I'\vec{v}') = \gamma(r)$.

- Suppose otherwise. Suppose $r = k'I'\vec{v}'$. Since $\gamma(\vec{s}+kI\vec{v}) \subseteq \gamma(k'I'\vec{v}') = \gamma(k'I'\vec{v}'^{+1})$, then $\vec{s}+kI\vec{v} \sqsubseteq \vec{0}+k'I'\vec{v}'$. Therefore, by definition 15, $kI\vec{v} \sqsubseteq k'((\vec{0}-\vec{s})/(k'\vec{v}')+I')\vec{v}'$. So $k'\vec{v}'|k\vec{v}$ and $k'\vec{v}'|\vec{s}$. By lemma 2, $k'\vec{v}'|gcd(\vec{s},k\vec{v}) \neq \perp$, which is a contradiction.

□

for proposition 8. To show $m \sqsubseteq \vec{u}+(m \ominus \vec{u})$.

- Suppose $I = [1,1]$ and $(\vec{s}-\vec{u})+k\vec{v} = \vec{0}$. Then, by lemma 27, $m \equiv (\vec{s}+k\vec{v})+0[1,1]\vec{1} = \vec{u}+0[1,1]\vec{1} = \vec{u}+((\vec{s}-\vec{u})+kI\vec{v})^{-1} = \vec{u}+(m \ominus \vec{u})$.

- Suppose $I = [1,1]$ and $(\vec{s}-\vec{u})+k\vec{v} \neq \vec{0}$. Then, by two applications of lemma 27, $m \equiv (\vec{s}+k\vec{v})+0[1,1]\vec{1} = (\vec{u}+((\vec{s}-\vec{u})+k\vec{v}))+0[1,1]\vec{1} \equiv \vec{u}+1[1,1]((\vec{s}-\vec{u})+k\vec{v}) = \vec{u}+((\vec{s}-\vec{u})+kI\vec{v})^{-1} = \vec{u}+(m \ominus \vec{u})$.

- Suppose otherwise. By lemma 28, $\gamma((\vec{s}-\vec{u})+kI\vec{v}) \subseteq \gamma(((\vec{s}-\vec{u})+kI\vec{v})^{-1}) = \gamma(\vec{0}+((\vec{s}-\vec{u})+kI\vec{v})^{-1})$. Thus by lemma 23, $(\vec{s}-\vec{u})+kI\vec{v} \sqsubseteq \vec{0}+((\vec{s}-\vec{u})+kI\vec{v})^{-1}$. Thus by corollary 13, $m = \vec{s}+kI\vec{v} \sqsubseteq \vec{u}+((\vec{s}-\vec{u})+kI\vec{v})^{-1} = \vec{u}+(m \ominus \vec{u})$.

Now suppose $I = [1,1]$ or $k\vec{v}|(\vec{s}-\vec{u})$. To show $m \equiv \vec{u}+(m \ominus \vec{u})$.

- Suppose $I = [1,1]$ and $\vec{s}-\vec{u}+k\vec{v} = \vec{0}$. Then $m \ominus \vec{u} = ((\vec{s}-\vec{u})+kI\vec{v})^{-1} = 0[1,1]\vec{1}$. Thus $\vec{u}+(m \ominus \vec{u}) = \vec{u}+0[1,1]\vec{1} = (\vec{s}+k\vec{v})+0[1,1]\vec{1} \equiv \vec{s}+kI\vec{v} = m$, by lemma 27.

- Suppose $I = [1,1]$ and $\vec{s}-\vec{u}+k\vec{v} \neq \vec{0}$. Then $m \ominus \vec{u} = ((\vec{s}-\vec{u})+kI\vec{v})^{-1} = 1[1,1](\vec{s}-\vec{u}+k\vec{v})$. Thus by two applications of lemma 27, $\vec{u}+(m \ominus \vec{u}) = \vec{u}+1[1,1](\vec{s}-\vec{u}+k\vec{v}) \equiv (\vec{u}+\vec{s}-\vec{u}+k\vec{v})+0[1,1]\vec{1} = (\vec{s}+k\vec{v})+0[1,1]\vec{1} \equiv \vec{s}+kI\vec{v} = m$.

- Suppose $k\vec{v}|(\vec{s}-\vec{u})$. Then $\vec{z} = gcd(\vec{s}-\vec{u}, k\vec{v}) = \pm k\vec{v} \neq \perp$.
  - Suppose $\vec{z} = +k\vec{v}$. Thus $m \ominus \vec{u} = ((\vec{s}-\vec{u})+kI\vec{v})^{-1} = 1((\vec{s}-\vec{u})/\vec{z}+(k\vec{v})/\vec{z}\cdot I)\vec{z} = 1((\vec{s}-\vec{u})/(k\vec{v})+(k\vec{v})/(k\vec{v})\cdot I)(k\vec{v}) = 1((\vec{s}-\vec{u})/(k\vec{v})+1\cdot I)(k\vec{v}) = 1((\vec{s}-\vec{u})/(k\vec{v})+I)(k\vec{v}) \equiv k((\vec{s}-\vec{u})/(k\vec{v})+I)\vec{v}$, by the second part of corollary 11 since $I \neq [1,1]$ so $k \neq 0$. Then $\vec{u} + (m \ominus \vec{u}) \equiv \vec{u} + k((\vec{s}-\vec{u})/(k\vec{v})+I)\vec{v}$. Let $\vec{t} = \vec{s}-\vec{u} \in \mathbb{Z}^n$, thus $k\vec{v}|\vec{t}$. By part 4 of corollary 14 with $\vec{u}$ and $\vec{t}$ transposed, $m = \vec{s} + kI\vec{v} \equiv \vec{u} + k((\vec{s}-\vec{u})/(k\vec{v})+I)\vec{v} \equiv \vec{u} + (m \ominus \vec{u})$.
  - Suppose $\vec{z} = -k\vec{v}$. Thus $m \ominus \vec{u} = ((\vec{s}-\vec{u})+kI\vec{v})^{-1} = 1((\vec{s}-\vec{u})/\vec{z}+(k\vec{v})/\vec{z}\cdot I)\vec{z} = 1((\vec{s}-\vec{u})/(-k\vec{v})+(k\vec{v})/(-k\vec{v})\cdot I)(-k\vec{v}) = 1((\vec{s}-\vec{u})/(-k\vec{v})+(-1)\cdot I)(-k\vec{v}) \equiv -1((\vec{s}-\vec{u})/k\vec{v}+I)(-k\vec{v})$, by lemma 13. Then since $I \neq [1,1]$ it follows $k \neq 0$, hence by the second part of corollary 11, $-1((\vec{s}-\vec{u})/k\vec{v}+I)(-k\vec{v}) \equiv (-1\times -k)((\vec{s}-\vec{u})/(k\vec{v})+I)\vec{v} \equiv k((\vec{s}-\vec{u})/(k\vec{v})+I)\vec{v}$. The proof now proceeds as per above.

□

for proposition 9. Let $m_1 = \vec{s}_1 + k_1 I_1 \vec{x}$ and $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} \in \mathsf{March}^n$. To show $m_1 \sqcup m_2$ is a least upper bound.

- To show $m_1 \sqsubseteq m_1 \sqcup m_2$.
  - Suppose $I_1 = [1,1]$. By lemma 28, and then by proposition 7, $\gamma((\vec{s}_1-\vec{u})+k_1I_1\vec{x}) \subseteq \gamma(((\vec{s}_1-\vec{u})+k_1I_1\vec{x})^{-1}) = \gamma(m_1 \ominus \vec{u}) \subseteq \gamma(m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) = \gamma(\vec{0} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}))$. Thus by lemma 23, $(\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x} \sqsubseteq \vec{0} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u})$. Hence, by corollary 13, $m_1 = \vec{s}_1 + k_1 I_1 \vec{x} \sqsubseteq \vec{u} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) = m_1 \sqcup m_2$.
  - Suppose otherwise. The proof is analogous to the previous case.
- To show $m_2 \sqsubseteq m_1 \sqcup m_2$. The proof is symmetric to the previous case.

Let $m = \vec{s} + kI\vec{w}$ and let $m_1 \sqsubseteq m$ and $m_2 \sqsubseteq m$. To show $m_1 \sqcup m_2 \sqsubseteq m$.

- Suppose $I_1 = [1,1]$.

– Suppose $I = [1,1]$. Since $m_1 \sqsubseteq m$, then by definition 15, $\vec{s}_1 + k_1\vec{x} = \vec{s} + k\vec{w}$. Since $I = [1,1]$, it must follow that $I_2 = [1,1]$, and thus $\vec{s}_2 + k_2\vec{y} = \vec{s} + k\vec{w}$ similarly holds. Thus $\vec{u} = \vec{s}_1 + k_1\vec{x} = \vec{s} + k\vec{w} = \vec{s}_2 + k_2\vec{y}$. Observe $\gamma((\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x}) = \{\vec{s}_1 - \vec{u} + k_1\vec{x}\} = \{\vec{0}\} \subseteq \{\vec{0}\} = \gamma(0[1,1]\vec{1})$. Then by lemma 28, $\gamma(m_1 \ominus \vec{u}) = \gamma((\vec{s}_1 + k_1 I_1 \vec{x}) \ominus \vec{u}) = \gamma(((\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x})^{-1}) \subseteq \gamma(0[1,1]\vec{1})$. Since $\gamma((\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x}) = \{(\vec{s}_1 + k_1\vec{x}) - \vec{u}\} = \{\vec{0}\} = \{(\vec{s}_2 + k_2\vec{y}) - \vec{u}\} = \gamma((\vec{s}_2 - \vec{u}) + k_2 I_2 \vec{y})$, then by the same reasoning, $\gamma(m_2 \ominus \vec{u}) \subseteq \gamma(0[1,1]\vec{1})$. Thus by proposition 7, $\gamma(m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) \subseteq \gamma(0[1,1]\vec{1})$ and so by lemma 16, $m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u} \sqsubseteq 0[1,1]\vec{1}$. Observing by lemma 27, $m_1 \equiv (\vec{s}_1 + k_1\vec{x}) + 0[1,1]\vec{1} = \vec{u} + 0[1,1]\vec{1}$, hence by lemma 26, $m_1 \sqcup m_2 = \vec{u} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) \sqsubseteq \vec{u} + 0[1,1]\vec{1} \equiv m_1 \sqsubseteq m$.

– Suppose otherwise.

  * To show $m_1 \ominus \vec{u} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. By lemma 27, $\vec{u} + 0[1,1]\vec{1} \equiv m_1 \sqsubseteq m = \vec{s} + kI\vec{w}$. Thus by definition 15, $0[1,1]\vec{1} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. Then by lemma 16, $\gamma((\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x}) = \gamma((\vec{s}_1 - (\vec{s}_1 + k_1\vec{x})) + k_1 I_1 \vec{x}) = \gamma(-k_1\vec{x} + k_1 I_1 \vec{x}) = \gamma(0[1,1]\vec{1}) \subseteq \gamma(k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w})$. So by lemma 28, $\gamma(m_1 \ominus \vec{u}) = \gamma((\vec{s}_1 + k_1 I_1 \vec{x}) \ominus \vec{u}) = \gamma(((\vec{s}_1 - \vec{u}) + k_1 I_1 \vec{x})^{-1}) \subseteq \gamma(k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w})$. Hence by lemma 16, $m_1 \ominus \vec{u} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$.
  * To show $m_2 \ominus \vec{u} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. Repeating the argument from the previous case, by definition 15, $k\vec{w} | \vec{s} - \vec{u}$, then by case 4 of corollary 14, $\vec{s}_2 + k_2 I_2 \vec{y} = m_2 \sqsubseteq m = \vec{s} + kI\vec{w} \equiv \vec{u} + k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. By corollary 13, $(\vec{s}_2 - \vec{u}) + k_2 I_2 \vec{y} \sqsubseteq \vec{0} + k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. Thus, by lemma 23, $\gamma((\vec{s}_2 - \vec{u}) + k_2 I_2 \vec{y}) \subseteq \gamma(\vec{0} + k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}) = \gamma(k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w})$. So by lemma 28, $\gamma(m_2 \ominus \vec{u}) = \gamma((\vec{s}_2 + k_2 I_2 \vec{y}) \ominus \vec{u}) = \gamma(((\vec{s}_2 - \vec{u}) + k_2 I_2 \vec{y})^{-1}) \subseteq \gamma(k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w})$. Hence by lemma 16, $m_2 \ominus \vec{u} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$.

  Thus by proposition 7, $m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u} \sqsubseteq k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w}$. Thus by lemma 26 and then by case 4 of corollary 14 (with $\vec{t}$ brought across and $\vec{u}$ replacing $\vec{t}$), $\vec{0} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) \sqsubseteq \vec{0} + k((\vec{s} - \vec{u})/(k\vec{w}) + I)\vec{w} \equiv (\vec{s} - \vec{u}) + kI\vec{w}$. Hence by corollary 13, $m_1 \sqcup m_2 = \vec{u} + (m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) \sqsubseteq \vec{s} + kI\vec{w} = m$.

- Suppose otherwise.
  - Suppose $I_2 = [1,1]$.
    - To show $m_1 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. This is analogous to the case above showing $m_2 \ominus \vec{u} \sqsubseteq k((\vec{s}-\vec{u})/(k\vec{w})+I)\vec{w}$ when $I_1 = [1,1]$.
    - To show $m_2 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. This is analogous to the case above showing $m_1 \ominus \vec{u} \sqsubseteq k((\vec{s}-\vec{u})/(k\vec{w})+I)\vec{w}$ when $I_1 = [1,1]$.
  - Suppose $I_2 \neq [1,1]$.
    - To show $m_1 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. This is analogous to the case showing $m_2 \ominus \vec{u} \sqsubseteq k((\vec{s}-\vec{u})/(k\vec{w})+I)\vec{w}$ when $I_1 = [1,1]$.
    - To show $m_2 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. Observe by case 3 of corollary 14, $\vec{v}+k_2(-1+I_2)\vec{y} = (\vec{s}_2+k_2\vec{y})+k_2(-1+I_2)\vec{y} \equiv \vec{s}_2 + k_2I_2\vec{y} = m_2 \sqsubseteq m = \vec{s}+kI\vec{w}$. Thus by definition 15, $k_2(-1+I_2)\vec{y} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. So by special case 3 of corollary 14 (with $k_2\vec{y}$ brought across) and then by lemma 26, $(\vec{s}_2-\vec{v})+k_2I_2\vec{y} = (\vec{s}_2-(\vec{s}_2+k_2\vec{y}))+k_2I_2\vec{y} = -k_2\vec{y}+k_2I_2\vec{y} \equiv \vec{0}+k_2(-1+I_2)\vec{y} \sqsubseteq \vec{0}+k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. Thus by lemma 23, $\gamma((\vec{s}_2-\vec{v})+k_2I_2\vec{y}) \subseteq \gamma(\vec{0}+k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}) = \gamma(k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w})$. So by lemma 28, $\gamma(m_2 \ominus \vec{v}) = \gamma((\vec{s}_2+k_2I_2\vec{y}) \ominus \vec{v}) = \gamma(((\vec{s}_2-\vec{v})+k_2I_2\vec{y})^{-1}) \subseteq \gamma(k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w})$. Hence by lemma 16, $m_2 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$.

  Thus by proposition 7, $m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v} \sqsubseteq k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w}$. Since $|I_1| > 1$, then $I \neq [1,1]$. Thus by lemma 26 and then case 4 of corollary 14, $m_1 \sqcup m_2 = \vec{v}+(m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v}) \sqsubseteq \vec{v}+k((\vec{s}-\vec{v})/(k\vec{w})+I)\vec{w} \equiv \vec{s}+kI\vec{w} = m$.

□

for corollary 15. Let $\vec{u} = \vec{s}_1 + k_1\vec{x} = \vec{s}_1 + \vec{0} = \vec{s}_1$. By definition 18, $m_1 \ominus \vec{u} = ((\vec{s}_1-\vec{s}_1)+0[1,1]\vec{1})^{-1} = (0[1,1]\vec{1}+0[1,1]\vec{1})^{-1} = (0[1,1]\vec{1})^{-1} = 0[1,1]\vec{1}$. Since $\vec{s}_1 \neq \vec{s}_2$, then $m_2 \ominus \vec{u} = ((\vec{s}_2-\vec{s}_1)+0[1,1]\vec{1})^{-1} = 1[1,1](\vec{s}_2-\vec{s}_1)$ since $(\vec{s}_2-\vec{s}_1)+\vec{0} \neq \vec{0}$. So $m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u} = 0[1,1]\vec{1} \sqcup 1[1,1](\vec{s}_2-\vec{s}_1) = 1[0,1](\vec{s}_2-\vec{s}_1)$. Thus by proposition 9, $m_1 \sqcup m_2 = \vec{u}+(m_1 \ominus \vec{u} \sqcup m_2 \ominus \vec{u}) = \vec{s}_1 + 1[0,1](\vec{s}_2-\vec{s}_1)$. □

for proposition 10. Let $\vec{b} \in \mathbb{Z}^{|\vec{y}|}$.

- Let $j \in [1, \ell]$. To show $[\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) \models [\![f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b})$. The result is immediate if $[\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) = \bot$. Hence suppose $[\![\forall i \in I.\, f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : \vec{0}) = \top$. It suffices to show $[\![f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = \top$. There exists $a' \in \gamma(I)$ such that $[\![\vec{p}]\!]_{\vec{i}}(a') = \vec{a}_j$. By definition 24, it follows $[\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : a') = \top$. By definition 28, $[\![f]\!]_{\vec{x}:\vec{y}}([\![\vec{p}]\!]_{\vec{i}}(a') : \vec{b}) = [\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : a') = \top$. But $[\![\vec{p}]\!]_{\vec{i}}(a') = \vec{a}_j$, hence, by definition 27, $[\![f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = \top$, as required.

- Let $m \in \gamma(I)$. To show $[\![\wedge_{j=1}^{\ell} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) \models [\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : m)$. The result is immediate if $[\![\wedge_{j=1}^{\ell} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = \bot$. Hence, as before, suppose $[\![\wedge_{j=1}^{\ell} f[\vec{x} \to \vec{a}_j]]\!]_{\vec{y}}(\vec{b}) = \top$. It suffices to show $[\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : m) = \top$. By definition 24, it follows $[\![f[\vec{x} \to \vec{a}_j]\!]_{\vec{y}}(\vec{b}) = \top$ for all $j \in [1, \ell]$. Since $A = [\![\vec{p}]\!]_{\vec{i}}(\gamma(I))$ then $\vec{a}_j = [\![\vec{p}]\!]_{\vec{i}}(m)$ for some $j \in [1, \ell]$. By definitions 27 and 28, $\top = [\![f]\!]_{\vec{x}:\vec{y}}([\![\vec{p}]\!]_{\vec{i}}(m) : \vec{b}) = [\![f[\vec{x} \to \vec{p}]]\!]_{\vec{y}:\vec{i}}(\vec{b} : m)$, as required.

□

Appendix B

Implementation of abstract domains

This appendix continues the presentation of the abstract domain operations that commenced in section 3.7 of Chapter 3.

B.1 The k-interval abstract domain: kInt

Now follows the definition of the k-interval domain:

```
sealed trait kIntDomain extends AbstractDomain[kIntDomain] {

  def ⊑(that: kIntDomain): Boolean = kIntDomain.⊑(this, that)
  def ⊔(that: kIntDomain): kIntDomain = kIntDomain.⊔(this, that)
  def ⊓(that: kIntDomain): kIntDomain = ???

}
```

The definitions of $\sqsubseteq$ and $\sqcup$ are delegated to the companion object `kIntDomain`. The $\sqcap$ operator is defined `???`. A commentary is now provided on $\sqsubseteq$ and $\sqcup$:

The $\sqsubseteq$ operator is defined thus:

```
def ⊑(k1: kIntDomain, k2: kIntDomain): Boolean =
  (k1, k2) match {
    case (⊥, _)                          ⇒ true
    case (_, ⊥)                          ⇒ false
    case (KI(k1, i1), KI(k2, i2))        ⇒ (k2 | k1) && ((k1 `·` i1) ⊑ (k2 `·` i2))
```

```
    case (_, ⊤)                          ⇒ true
    case (⊤, _)                          ⇒ false
  }
```

The third case of $\sqsubseteq$ makes use of $|$ and $\cdot$ which are not native to the Scala Integer class but were added to it using enrichment (which augments an existing class with additional methods without introducing subtypes).

The join operator $\sqcup$ is defined as below, with the fifth case defined as an auxiliary method which follows:

```
def ⊔(ki1: kIntDomain, ki2: kIntDomain): kIntDomain =
  (ki1, ki2) match {
    case (ki1, ⊥)              ⇒ ki1
    case (⊥, ki2)              ⇒ ki2
    case (_, ⊤)                ⇒ ⊤
    case (⊤, _)                ⇒ ⊤
    case (ki1: KI, ki2: KI)    ⇒ ki1 ⊔ ki2
  }
```

Notice that the auxiliary method returns returns a kI rather than a $\top$ or $\bot$. Observe the side condition on the first case, the second case only matching if the side condition fails.

```
def ⊔(ki1: KI, ki2: KI): KI =
  (ki1, ki2) match {
    case (KI(k1, i1), KI(k2, i2))
      if k1 == k2 && i1 == i2 && i2 == Interval(1, 1) ⇒ ki1

    case (KI(k1, i1), KI(k2, i2)) ⇒
      val k = gcd(k1, k2)
      val i = ((k1 / k) `·` i1) ⊔ ((k2 / k) `·` i2)
      KI(k, i)
  }
```

B.2 The range abstract domain: Range^n

The code implementation for the Range^n domain follows:

```
trait RangeDomain extends AbstractDomain[RangeDomain] {

  def ⊑(that: RangeDomain): Boolean = RangeDomain.⊑(this, that)
  def ⊔(that: RangeDomain): RangeDomain = RangeDomain.⊔(this, that)
```

```
  def ⊓(that: RangeDomain): RangeDomain = ???

}
```

Similar to the k-interval domain, the definitions of $\sqsubseteq$ and $\sqcup$ for the Range^n are delegated to the companion object `RangeNDomain`, and the $\sqcap$ operator is defined `???`. A commentary is now provided on $\sqsubseteq$ and $\sqcup$:

The $\sqsubseteq$ operator is defined thus:

```
def ⊑(r1: RangeDomain, r2: RangeDomain): Boolean =
  (r1, r2) match {
    case (⊥, _) ⇒ true
    case (_, ⊤) ⇒ true
    case (KIV(k1, i1, x), KIV(k2, i2, y))
      if i1 == i2 && i1 == Interval(1, 1) && (k1 * x) == (k2 * y) ⇒ true
    case (KIV(k1, i1, x), KIV(k2, i2, y))
      if i2 != Interval(1, 1) &&
      ((k2 * y) | (k1 * x)) &&
      ((((k1 * x) / (k2 * y)) ⊗ KI(1, i1)) ⊑ KI(1, i2)) ⇒ true
    case _ ⇒ false
  }
```

The $\sqsubseteq$ operator is expressed over five cases; two special cases, two main cases and one catch-all case. The first two special cases express the condition $\bot \sqsubseteq r \sqsubseteq \top$ for all $r \in \mathsf{Range}^n$. The third case, the first main case of the specification, expresses the case $I_1 = I_2 = [1, 1]$ and $k_1\vec{x} = k_2\vec{y}$. The fourth case, the second main case, expresses the case $I_2 \neq [1, 1]$, $k_2\vec{y} | k_1\vec{x}$ and $(k_1\vec{x})/(k_2\vec{y}) \otimes 1I_1 \sqsubseteq 1I_2$. The fifth case is a catch-all that returns `false` for any pair of ranges that do not satisfy any of the previous cases, the intuition being that the previous four cases give complete coverage, specifying every way in which one range can be included within another.

The $\sqcup$ operator is defined thus:

```
def ⊔(kIOneX: RangeDomain, kITwoY: RangeDomain): RangeDomain =
  (kIOneX, kITwoY) match {
    case (KIV(k1, i1, x), KIV(k2, i2, y))
      if i1 == i2 && i2 == Interval(1, 1) && (k1 * x == k2 * y) ⇒ kIOneX
    case (KIV(k1, i1, x), KIV(k2, i2, y))
      if gcdVectors(k1 * x, k2 * y).isDefined ⇒
        val z = gcdVectors(k1 * x, k2 * y).get
        KIV((((k1 * x) / z) ⊗ KI(1, i1)) ⊔ (((k2 * y) / z) ⊗ KI(1, i2)), z)
```

```
    case (KIV(_, _, _), KIV(_, _, _)) ⇒ ⊤
    case (r, ⊥) ⇒ r
    case (⊥, r) ⇒ r
    case (r, ⊤) ⇒ ⊤
    case (⊤, r) ⇒ ⊤
}
```

The $\sqcup$ operator is defined over seven cases; the first three match the first three main cases of the specification, and the final four handle the cases $r \sqcup \bot = \bot \sqcup r = r$ and $r \sqcup \top = \top \sqcup r = \top$ for all $r \in \mathsf{Range}^n$. The first main case handles when two ranges describe the same point. The second case checks to see if $\vec{z} = gcd(k_1\vec{x}, k_2\vec{y})$ is defined. If so, both input ranges describe points on a common line. Join then reduces to the join of two k-intervals, followed by a scaling step with $\vec{z}$. The third and final main case returns $\top$ as a fallback when joining two ranges (not $\top$ or $\bot$) that do not satisfy either the first or second case.

B.3 The march abstract domain: March^n

The code for the March^n domain follows, where the trait augments the main methods $\sqsubseteq$, $\sqcup$ and $\sqcap$, analogous to those given previously, with special auxiliaries for dimension reduction ($^{-1}$) and march translation ($\ominus$):

```
trait MarchDomain extends AbstractDomain[MarchDomain] {

  def ⊑(that: MarchDomain): Boolean = MarchDomain.⊑(this, that)
  def ⊔(that: MarchDomain): MarchDomain = MarchDomain.⊔(this, that)
  def ⊓(that: MarchDomain): MarchDomain = ???

  def dimensionReduction: RangeDomain =
    this match {
      case March(s, KIV(k, i, v))
        if i == Interval(1,1) && (s + k * v) == Vector.fill(s.length)(0)
                              ⇒ KIV(KI(0, Interval(1,1)), Vector.fill(s.length)(1))
      case March(s, KIV(k, i, v))
        if i == Interval(1,1) && (s + k * v) != Vector.fill(s.length)(0)
                              ⇒ KIV(KI(1, Interval(1,1)), s + k * v)
      case March(s, KIV(k, i, v))
        if gcdVectors(s, k*v).isDefined
                              ⇒ val z = gcdVectors(s, k*v).get
                                KIV(KI(1, (s/z) + ((k*v)/z dot i)), z)
      case March(s, KIV(k, i, v)) ⇒ RangeDomain.⊤
      case MarchDomain.⊤          ⇒ RangeDomain.⊤
```

```
      case MarchDomain.⊥          ⇒ RangeDomain.⊥
    }

  def ^(i: Int): RangeDomain = if (i == -1) dimensionReduction else ???

  def ⊖(u: Vector[Int]): RangeDomain = this match {
    case March(s, kiv) ⇒ March(s - u, kiv)^(-1)
    case MarchDomain.⊥ ⇒ RangeDomain.⊥
    case MarchDomain.⊤ ⇒ RangeDomain.⊤
  }
  def ominus: Vector[Int] ⇒ RangeDomain = ⊖

}
```

The definitions of $\sqsubseteq$ and $\sqcup$ are defined in the companion object `MarchDomain` and are presented below, starting with the $\sqsubseteq$ operator, then the join $\sqcup$ operator:

The $\sqsubseteq$ operator is defined thus:

```
def ⊑(m1: MarchDomain, m2: MarchDomain): Boolean =
  (m1, m2) match {
    case (⊥, m)                                                              ⇒ true
    case (m, ⊤)                                                              ⇒ true
    case (March(s1, KIV(k1, i1, x)), March(s2, KIV(k2, i2, y)))
      if i1 == i2 && i2 == Interval(1,1) && (s1 + k1 * x == s2 + k2 * y)     ⇒ true
    case (March(s1, KIV(k1, i1, x)), March(s2, KIV(k2, i2, y)))
      if i1 == Interval(1,1) &&
         i2 != Interval(1,1) &&
         ((k2 * y) | (s2 - (s1 + k1 * x))) &&
         (Interval(0,0) ⊑ ((s2 - (s1 + k1 * x))/(k2 * y) + i2))              ⇒ true
    case (March(s1, kIOneX @ KIV(k1, i1, x)), March(s2, KIV(k2, i2, y)))
      if i1 != Interval(1,1) &&
         i2 != Interval(1,1) &&
         (kIOneX ⊑ KIV(KI(k2, (s2 - s1) / (k2 * y) + i2), y))                ⇒ true
    case _ ⇒ false
  }
```

The method follows the specification of $\sqsubseteq$ closely where the first two case statements encode $\bot \sqsubseteq m \sqsubseteq \top$ for all $m \in \mathsf{March}^n$. The third case deals with two marches which describe the same point. The fourth case handles the case where a point described by the first march is enclosed by the points of the second march. The fifth case covers the case where both marches describe multiple points. This reduces to a range inclusion problem.

The join operator $\sqcup$ is defined thus:

```
def ⊔(m1: MarchDomain, m2: MarchDomain): MarchDomain =
  (m1, m2) match {
    case (March(s1, KIV(k1, i1, x)), m2: March) if i1 == Interval(1,1) ⇒
      val u = s1 + k1 * x
      March(u, (m1 ⊖ u) ⊔ (m2 ⊖ u))
    case (m1: March, March(s2, KIV(k2, i2, y))) ⇒
      val v = s2 + k2 * y
      March(v, (m1 ⊖ v) ⊔ (m2 ⊖ v))
    case (_, ⊤) ⇒ ⊤
    case (⊤, _) ⇒ ⊤
  }
```

The implementation of $\sqcup$ amounts to two main cases and two special cases. In the main cases, an offset vector $\vec{u}$ or $\vec{v}$ is defined by which both marches are translated, resulting in two ranges which are then combined by join. This resultant k-interval is then offset by $\vec{u}$ or $\vec{v}$ to give the resultant march.

Special auxiliaries $^{-1}$ and $\ominus$ The methods $^{-1}$ and $\ominus$ realize dimension reduction and march translation respectively.

Concluding statement The fact that the domain operators have been proven correct provides assurance that the algorithms are conceptually sound whereas their direct encoding into Scala provides little opportunity for error in the implementation. Further discussion on the rationale for using Scala is provided in section 6.2 of Chapter 6.

Appendix C

Examples validation

The examples presented in Chapter 3 have been validated by passing them through ImpactExplorer.

C.1 Examples from Chapter 3

Example 2. Let $I_1 = [0, 4]$ and $I_2 = [-1, 5]$. Observe $I_1, I_2 \in \mathsf{Int}$ and $I_1 \sqsubseteq I_2$. Then $-5 \cdot I_1 = -5 \cdot [0, 4] = [-20, 0] \sqsubseteq [-25, 5] = -5 \cdot [-1, 5] = -5 \cdot I_2$.

```
def example2: Unit = {
  println("EXAMPLE 2")

  val I1 = IOne(0,4)
  val I2 = ITwo(-1,5)

  println(s"Let $I1 = ${I1.i} and $I2 = ${I2.i}. Observe $I1, $I2 ∈ $IntDomain.")
  assert(Seq(I1,I2).forall(_.isInstanceOf[IntDomain]))

  println(s"Observe $I1 ⊑ $I2.")
  assert(I1 ⊑ I2)

  println(s"Then -5 · $I1 = -5 · ${I1.i} = ${-5 `·` I1} ⊑ ${-5 `·` I2} = -5 · ${I2.i} =
↪  -5 · $I2.")
  assert((-5 `·` I1) ⊑ (-5 `·` I2))
}
```

```
EXAMPLE 2
Let I₁ = [0,4] and I₂ = [-1,5]. Observe I₁, I₂ ∈ Int.
Observe I₁ ⊑ I₂.
Then -5 · I₁ = -5 · [0,4] = [-20,0] ⊑ [-25,5] = -5 · [-1,5] = -5 · I₂.
```

Example 3. Let $a = 2, b = -3$ and $I = [1, 3]$.

- Then $a \cdot (b \cdot I) = 2 \cdot (-3 \cdot [1, 3]) = 2 \cdot [-9, -3] = [-18, -6]$
- Then $(ab) \cdot I = (2 \times -3) \cdot [1, 3] = -6 \cdot [1, 3] = [-18, -6]$
- Then $b \cdot (a \cdot I) = -3 \cdot (2 \cdot [1, 3]) = -3 \cdot [2, 6] = [-18, -6]$

```
def example3: Unit = {
  println("EXAMPLE 3")

  val a = 2 ; val b = -3 ; val I = Interval(1,3)

  println(s"Let a = $a, b = $b and I = $I.")
  println(s"Then a · (b · I) = $a · ($b · $I) = $a · ${b `·` I} = " + (a `·` (b `·`
↪  I)))
  println(s"Then (ab) · I = ($a * $b) · $I = ${a*b} · $I = " + ((a*b) `·` I))
  println(s"Then b · (a · I) = $b · ($a · $I) = $b · ${a `·` I} = " + (b `·` (a `·`
↪  I)))
}
```

```
EXAMPLE 3
Let a = 2, b = -3 and I = [1,3].
Then a · (b · I) = 2 · (-3 · [1,3]) = 2 · [-9,-3] = [-18,-6]
Then (ab) · I = (2 * -3) · [1,3] = -6 · [1,3] = [-18,-6]
Then b · (a · I) = -3 · (2 · [1,3]) = -3 · [2,6] = [-18,-6]
```

Example 4. Let $a = -3$, $I_1 = [-4, -2]$ and $I_2 = [1, 3]$. Observe $I_1, I_2 \in \mathsf{Int}$. Then $a \cdot (I_1 \sqcup I_2) = -3 \cdot ([-4, -2] \sqcup [1, 3]) = -3 \cdot [-4, 3] = [-9, 12] = [6, 12] \sqcup [-9, -3] = -3 \cdot [-4, -2] \sqcup -3 \cdot [1, 3] = a \cdot I_1 \sqcup a \cdot I_2$.

```
def example4: Unit = {
  println("EXAMPLE 4")

   val a = -3 ; val I1 = IOne(-4,-2) ; val I2 = ITwo(1,3)

  println(s"Let a = $a, $I1 = ${I1.i} and $I2 = ${I2.i}. Observe $I1, $I2 ∈
↪  $IntDomain.")
  assert(Seq(I1,I2).forall(_.isInstanceOf[IntDomain]))

  println(s"Then a · ($I1 ⊔ $I2) = $a · (${I1.i} ⊔ ${I2.i}) = $a · ${I1 ⊔ I2} = " + (a
↪  `·` (I1 ⊔ I2)))
  assert((a `·` (I1 ⊔ I2)) == ((a `·` I1) ⊔ (a `·` I2)))
  print(s"= ${a `·` I1} ⊔ ${a `·` I2} = $a · ${I1.i} ⊔ $a · ${I2.i} = ")
  println(s"a · $I1 ⊔ a · $I2")
}
```

```
EXAMPLE 4
Let a = -3, I₁ = [-4,-2] and I₂ = [1,3]. Observe I₁, I₂ ∈ Int.
Then a · (I₁ ⊔ I₂) = -3 · ([-4,-2] ⊔ [1,3]) = -3 · [-4,3] = [-9,12]
= [6,12] ⊔ [-9,-3] = -3 · [-4,-2] ⊔ -3 · [1,3] = a · I₁ ⊔ a · I₂
```

Example 5. Let $k_1I_1 = 1[-3,-1]$ and $k_2I_2 = 3[-1,0]$. The two multipliers k_1 and k_2 are weakened to a common $k = gcd(k_1, k_2)$. Here $k = gcd(1,3) = 1$. Then the intervals I_1 and I_2 are scaled to compensate for the smaller multiplier k. Since $k_2 = 3$, the interval $I_2 = [-1,0]$ is scaled to give $3/1 \cdot [-1,0] = [-3,0]$. Since $k_1 = k$, I_1 is not scaled. Hence $k_1I_1 \sqcup k_2I_2 = 1\big(1/1 \cdot [-3,-1] \sqcup 3/1 \cdot [-1,0]\big) = 1([-3,-1] \sqcup [-3,0]) = 1[-3,0]$. Figure 10(a) illustrates this join where the concretizations of k_1I_1 and k_2I_2 are shown in green and blue respectively; the concretization of kI is given in red. Note each point in the concretization of kI corresponds to a point in the concretization of k_1I_1 or k_2I_2. Hence no extraneous points are introduced.

```
def example5: Unit = {
  println("EXAMPLE 5")

  val kI1 = KIOne(1, Interval(-3,-1))
  val kI2 = KITwo(3, Interval(-1,0))
  println(s"Let $kI1 = ${kI1.ki} and $kI2 = ${kI2.ki}.")
  println(s"Here k = gcd(${kI1.k}, ${kI2.k}) = " + gcd(kI1.k, kI2.k) + ".")

  val kI = kI1 ⊔ kI2

  println(s"Hence kI = $kI1 ⊔ $kI2 = ${kI1.ki ⊔ (kI2.ki, "step1")} = ${kI1.ki ⊔
↪  (kI2.ki, "step3")} = ${kI1.ki ⊔ kI2.ki}")
  println(s"γ($kI1) = γ(${kI1.ki}) = " + `γ`(kI1))
  println(s"γ($kI2) = γ(${kI2.ki}) = " + `γ`(kI2))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
}
```

```
EXAMPLE 5
Let k₁I₁ = 1[-3,-1] and k₂I₂ = 3[-1,0].
Here k = gcd(1, 3) = 1.
Hence kI = k₁I₁ ⊔ k₂I₂ = 1(1/1 · [-3,-1] ⊔ 3/1 · [-1,0]) = 1([-3,-1] ⊔ [-3,0]) =
↪  1[-3,0]
γ(k₁I₁) = γ(1[-3,-1]) = TreeSet(-3, -2, -1)
γ(k₂I₂) = γ(3[-1,0]) = TreeSet(-3, 0)
γ(kI) = γ(1[-3,0]) = TreeSet(-3, -2, -1, 0)
```

Example 6. Let $k_1I_1 = 1[-3,-1]$ and $k_3I_3 = 2[-1,0]$ so that the first points of the concretizations, as well as their last, are out of alignment. Since $gcd(1,2) = 1$, then

$kI = k_1I_1 \sqcup k_3I_3 = 1\big(1/1 \cdot [-3,-1] \sqcup 2/1 \cdot [-1,0]\big) = 1([-3,-1] \sqcup [-2,0]) = 1[-3,0]$. Figure 10(b) illustrates that, again, no extraneous points are introduced.

```
def example6: Unit = {
  println("EXAMPLE 6")

  val kI1 = KIOne(1, Interval(-3,-1))
  val kI3 = KIThree(2, Interval(-1,0))
  println(s"Let $kI1 = ${kI1.ki} and $kI3 = ${kI3.ki}.")
  println(s"Here k = gcd(${kI1.k}, ${kI3.k}) = " + gcd(kI1.k, kI3.k) + ".")

  val kI = kI1 ⊔ kI3

  println(s"Hence kI = $kI1 ⊔ $kI3 = ${kI1.ki ⊔ (kI3.ki, "step1")} = ${kI1.ki ⊔
↪  (kI3.ki, "step3")} = ${kI1.ki ⊔ kI3.ki}")
  println(s"γ($kI1) = γ(${kI1.ki}) = " + `γ`(kI1))
  println(s"γ($kI3) = γ(${kI3.ki}) = " + `γ`(kI3))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
}
```

```
EXAMPLE 6
Let k₁I₁ = 1[-3,-1] and k₃I₃ = 2[-1,0].
Here k = gcd(1, 2) = 1.
Hence kI = k₁I₁ ⊔ k₃I₃ = 1(1/1 · [-3,-1] ⊔ 2/1 · [-1,0]) = 1([-3,-1] ⊔ [-2,0]) =
↪  1[-3,0]
γ(k₁I₁) = γ(1[-3,-1]) = TreeSet(-3, -2, -1)
γ(k₃I₃) = γ(2[-1,0]) = TreeSet(-2, 0)
γ(kI) = γ(1[-3,0]) = TreeSet(-3, -2, -1, 0)
```

Example 7. Let $k_1I_1 = 1[-3,-1]$ and $k_4I_4 = 0[1,1]$ such that k_4I_4 is degenerate in that it describes a single point. See figure 10(c). Since $gcd(1,0) = 1$, then $k_1I_1 \sqcup k_4I_4 = 1\big(1/1 \cdot [-3,-1] \sqcup 0/1 \cdot [1,1]\big) = 1([-3,-1] \sqcup [0,0]) = 1[-3,0]$. This illustrates that, in general, the magnitude of k can exceed the magnitude of k_4, and likewise k_1.

```
def example7: Unit = {
  println("EXAMPLE 7")

  val kI1 = KIOne(1, Interval(-3,-1))
  val kI4 = KIFour(0, Interval(1,1))
  println(s"Let $kI1 = ${kI1.ki} and $kI4 = ${kI4.ki}.")
  println(s"Here k = gcd(${kI1.k}, ${kI4.k}) = " + gcd(kI1.k, kI4.k) + ".")

  val kI = kI1 ⊔ kI4
```

```
  println(s"Hence kI = $kI1 ⊔ $kI4 = ${kI1.ki ⊔ (kI4.ki, "step1")} = ${kI1.ki ⊔
↪  (kI4.ki, "step3")} = ${kI1.ki ⊔ kI4.ki}")
  println(s"γ($kI1) = γ(${kI1.ki}) = " + `γ`(kI1))
  println(s"γ($kI4) = γ(${kI4.ki}) = " + `γ`(kI4))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
}
```

```
EXAMPLE 7
Let k₁I₁ = 1[-3,-1] and k₄I₄ = 0[1,1].
Here k = gcd(1, 0) = 1.
Hence kI = k₁I₁ ⊔ k₄I₄ = 1(1/1 · [-3,-1] ⊔ 0/1 · [1,1]) = 1([-3,-1] ⊔ [0,0]) =
↪  1[-3,0]
γ(k₁I₁) = γ(1[-3,-1]) = TreeSet(-3, -2, -1)
γ(k₄I₄) = γ(0[1,1]) = TreeSet(0)
γ(kI) = γ(1[-3,0]) = TreeSet(-3, -2, -1, 0)
```

Example 8. Let $k_5 I_5 = -3[1,1]$ and $k_4 I_4 = 0[1,1]$ so that both k-intervals are degenerate. See figure 10(d). Since $gcd(-3,0) = 3$, then $k_5 I_5 \sqcup k_4 I_4 = 3\big(-3/3 \cdot [1,1] \sqcup 0/3 \cdot [1,1]\big) = 3([-1,-1] \sqcup [0,0]) = 3[-1,0]$. More generally, whenever there are two degenerate k-intervals, no information is lost through join.

```
def example8: Unit = {
  println("EXAMPLE 8")

  val kI5 = KIFive(-3, Interval(1,1))
  val kI4 = KIFour(0, Interval(1,1))
  println(s"Let $kI5 = ${kI5.ki} and $kI4 = ${kI4.ki}.")
  println(s"Here k = gcd(${kI5.k}, ${kI4.k}) = " + gcd(kI5.k, kI4.k) + ".")

  val kI = kI5 ⊔ kI4

  println(s"Hence kI = $kI5 ⊔ $kI4 = ${kI5.ki ⊔ (kI4.ki, "step1")} = ${kI5.ki ⊔
↪  (kI4.ki, "step3")} = ${kI5.ki ⊔ kI4.ki}")
  println(s"γ($kI5) = γ(${kI5.ki}) = " + `γ`(kI5))
  println(s"γ($kI4) = γ(${kI4.ki}) = " + `γ`(kI4))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
}
```

```
EXAMPLE 8
Let k₅I₅ = -3[1,1] and k₄I₄ = 0[1,1].
Here k = gcd(-3, 0) = 3.
Hence kI = k₅I₅ ⊔ k₄I₄ = 3(-3/3 · [1,1] ⊔ 0/3 · [1,1]) = 3([-1,-1] ⊔ [0,0]) = 3[-1,0]
γ(k₅I₅) = γ(-3[1,1]) = TreeSet(-3)
γ(k₄I₄) = γ(0[1,1]) = TreeSet(0)
γ(kI) = γ(3[-1,0]) = TreeSet(-3, 0)
```

Example 9. To illustrate where extraneous points are introduced, consider the series of examples illustrated in figure 11. First, let $k_6 I_6 = 4[0,1]$ and $k_7 I_7 = 8[1,1]$. Since $gcd(4,8) = 4$, then $k_6 I_6 \sqcup k_7 I_7 = 4[0,1] \sqcup 8[1,1] = 4\big(4/4 \cdot [0,1] \sqcup 8/4 \cdot [1,1]\big) = 4([0,1] \sqcup [2,2]) = 4[0,2]$. Note in this case the point 8 can be reached by extending the interval of $k_6 I_6$ from $[0,1]$ to $[0,2]$, thus no information is lost. However, if the point had been 12, an extra point would have been be introduced at 8. Second, let $k_8 I_8 = 6[1,1]$. Since $gcd(4,6) = 2$, then $k_6 I_6 \sqcup k_8 I_8 = 4[0,1] \sqcup 6[1,1] = 2\big(4/2 \cdot [0,1] \sqcup 6/2 \cdot [1,1]\big) = 2([0,2] \sqcup [3,3]) = 2[0,3]$. Since the common multiplier k is strictly less than both k_1 and k_2, an extra point at 2 is introduced (indicated by a ring). Third, let $k_9 I_9 = 5[1,1]$. Since $gcd(4,5) = 1$, then $k_6 I_6 \sqcup k_9 I_9 = 4[0,1] \sqcup 5[1,1] = 1\big(4/1 \cdot [0,1] \sqcup 5/1 \cdot [1,1]\big) = 1([0,4] \sqcup [5,5]) = 1[0,5]$. The common multiplier of $k = 1$ triggers loss of precision.

```
def example9: Unit = {
  println("EXAMPLE 9")

  val kI6 = KISix(4, Interval(0,1))
  val kI7 = KISeven(8, Interval(1,1))
  println(s"Let $kI6 = ${kI6.ki} and $kI7 = ${kI7.ki}. Since gcd(${kI6.k}, ${kI7.k}) =
↪  ${gcd(kI6.k, kI7.k)}, then")
  var kI = kI6 ⊔ kI7
  println(s"kI = $kI6 ⊔ $kI7 = ${kI6.ki} ⊔ ${kI7.ki} = ${kI6.ki ⊔ (kI7.ki, "step1")} =
↪  ${kI6.ki ⊔ (kI7.ki, "step3")} = ${kI6.ki ⊔ kI7.ki}")

  println(s"γ($kI6) = γ(${kI6.ki}) = " + `γ`(kI6))
  println(s"γ($kI7) = γ(${kI7.ki}) = " + `γ`(kI7))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
  println("Extra points: " + (`γ`(kI) diff (`γ`(kI6) union `γ`(kI7))))

  println()
  val kI8 = KIEight(6, Interval(1,1))
  kI = kI6 ⊔ kI8
  println(s"Now let $kI8 = ${kI8.ki}. Since gcd(${kI6.k}, ${kI8.k}) = ${gcd(kI6.k,
↪  kI8.k)}, then" )
  println(s"kI = $kI6 ⊔ $kI8 = ${kI6.ki} ⊔ ${kI8.ki} = ${kI6.ki ⊔ (kI8.ki, "step1")} =
↪  ${kI6.ki ⊔ (kI8.ki, "step3")} = ${kI6.ki ⊔ kI8.ki}")
  println(s"γ($kI6) = γ(${kI6.ki}) = " + `γ`(kI6))
  println(s"γ($kI8) = γ(${kI8.ki}) = " + `γ`(kI8))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
  println("Extra points: " + (`γ`(kI) diff (`γ`(kI6) union `γ`(kI8))))

  println()
  val kI9 = KINine(5, Interval(1,1))
  kI = kI6 ⊔ kI9
```

```
  println(s"Now let $kI9 = ${kI9.ki}. Since gcd(${kI6.k}, ${kI9.k}) = ${gcd(kI6.k,
↪  kI9.k)}, then" )
  println(s"kI = $kI6 ⊔ $kI9 = ${kI6.ki} ⊔ ${kI9.ki} = ${kI6.ki ⊔ (kI9.ki, "step1")} =
↪  ${kI6.ki ⊔ (kI9.ki, "step3")} = ${kI6.ki ⊔ kI9.ki}")
  println(s"γ($kI6) = γ(${kI6.ki}) = " + `γ`(kI6))
  println(s"γ($kI9) = γ(${kI9.ki}) = " + `γ`(kI9))
  println(s"γ(kI) = γ($kI) = " + `γ`(kI))
  println("Extra points: " + (`γ`(kI) diff (`γ`(kI6) union `γ`(kI9))))
}
```

```
EXAMPLE 9
Let k₆I₆ = 4[0,1] and k₇I₇ = 8[1,1]. Since gcd(4, 8) = 4, then
kI = k₆I₆ ⊔ k₇I₇ = 4[0,1] ⊔ 8[1,1] = 4(4/4 · [0,1] ⊔ 8/4 · [1,1]) = 4([0,1] ⊔ [2,2])
↪  = 4[0,2]
γ(k₆I₆) = γ(4[0,1]) = TreeSet(0, 4)
γ(k₇I₇) = γ(8[1,1]) = TreeSet(8)
γ(kI) = γ(4[0,2]) = TreeSet(0, 4, 8)
Extra points: TreeSet()

Now let k₈I₈ = 6[1,1]. Since gcd(4, 6) = 2, then
kI = k₆I₆ ⊔ k₈I₈ = 4[0,1] ⊔ 6[1,1] = 2(4/2 · [0,1] ⊔ 6/2 · [1,1]) = 2([0,2] ⊔ [3,3])
↪  = 2[0,3]
γ(k₆I₆) = γ(4[0,1]) = TreeSet(0, 4)
γ(k₈I₈) = γ(6[1,1]) = TreeSet(6)
γ(kI) = γ(2[0,3]) = TreeSet(0, 2, 4, 6)
Extra points: TreeSet(2)

Now let k₉I₉ = 5[1,1]. Since gcd(4, 5) = 1, then
kI = k₆I₆ ⊔ k₉I₉ = 4[0,1] ⊔ 5[1,1] = 1(4/1 · [0,1] ⊔ 5/1 · [1,1]) = 1([0,4] ⊔ [5,5])
↪  = 1[0,5]
γ(k₆I₆) = γ(4[0,1]) = TreeSet(0, 4)
γ(k₉I₉) = γ(5[1,1]) = TreeSet(5)
γ(kI) = γ(1[0,5]) = TreeSet(0, 1, 2, 3, 4, 5)
Extra points: TreeSet(1, 2, 3)
```

Example 10. Let $I \in \mathsf{Int}$ where $I = [-1, 1]$. Then $2 \cdot I = 2 \cdot [-1, 1] = [-2, 2]$. Note $[-1, 1] \sqsubseteq [-2, 2]$. Now $2 \otimes 1I = 2 \otimes 1[-1, 1] = 2[-1, 1]$. But $\gamma(2[-1, 1]) = \{-2, 0, 2\}$ and $\gamma(1[-1, 1]) = \{-1, 0, 1\}$, hence by lemma 7, $2[-1, 1]$ and $1[-1, 1]$ are incomparable.

```
def example10: Unit = {
  println("EXAMPLE 10")

  val I = Interval(-1,1); assert(I.isInstanceOf[IntDomain])
  println(s"Let I = $I. Then 2 · I = 2 · $I = ${2 `·` I}.")
  println(s"Note $I ⊑ ${2 `·` I}"); assert(I ⊑ (2 `·` I))
  val oneI = KI(1, I)
```

```
  println(s"Then 2 ⊗ 1I = 2 ⊗ $oneI = " + (2 ⊗ oneI))
  println(s"But γ(${2 ⊗ oneI}) = ${`γ`(2 ⊗ oneI)} and γ($oneI) = ${`γ`(oneI)})" +
      s" hence ${2 ⊗ oneI} and $oneI are incomparable.")
  assert(2 ⊗ oneI ⋢ oneI)
}
```

```
EXAMPLE 10
Let I = [-1,1]. Then 2 · I = 2 · [-1,1] = [-2,2].
Note [-1,1] ⊑ [-2,2]
Then 2 ⊗ 1I = 2 ⊗ 1[-1,1] = 2[-1,1]
But γ(2[-1,1]) = TreeSet(-2, 0, 2) and γ(1[-1,1]) = TreeSet(-1, 0, 1)) hence 2[-1,1]
↪  and 1[-1,1] are incomparable.
```

Example 11. Consider $\lambda = -3$ and $4[-2,1] \in \mathsf{kInt}$. Then $\gamma(\lambda \otimes 4[-2,1]) = \gamma((-(-3) \times 4)(-1 \cdot [-2,1])) = \gamma(12[-1,2]) = \{-12, 0, 12, 24\} = \{-3x \mid x \in \{-8, -4, 0, 4\}\} = \{\lambda x \mid x \in \gamma(4[-2,1])\}$

```
def example11: Unit = {
  println("EXAMPLE 11")

  val λ = -3
  val kI = KI(4, Interval(-2, 1)); assert(kI.isInstanceOf[kIntDomain])
  print(s"Consider λ = $λ and $kI ∈ $kIntDomain. Then γ(λ ⊗ $kI}) = γ(${λ ⊗ (kI,
↪  "step1")}) = γ(${λ ⊗ kI}) = ${`γ`(λ ⊗ kI)}")
  println(s" = {$λ x | x ∈ ${`γ`(kI)} = {λ x | x ∈ γ($kI)}")
  assert(`γ`(λ ⊗ kI) == (for (x <- `γ`(kI)) yield λ * x))
}
```

```
EXAMPLE 11
Consider λ = -3 and 4[-2,1] ∈ kInt. Then γ(λ ⊗ 4[-2,1]}) = γ((-(-3) * 4)(-1 ·
↪  [-2,1])) = γ(12[-1,2]) = TreeSet(-12, 0, 12, 24) = {-3 x | x ∈ TreeSet(-8, -4, 0,
↪  4) = {λ x | x ∈ γ(4[-2,1])}
```

Example 12. To illustrate that the lemma holds even when λ is strictly negative, consider $\lambda = -2$, $k_1 I_1 = 5[1,2]$ and $k_2 I_2 = 0[1,1]$. First, observe $k_1 I_1 \sqcup k_2 I_2 = 5[1,2] \sqcup 0[1,1] = 5\big(5/5 \cdot [1,2] \sqcup 0/5 \cdot [1,1]\big) = 5\big(1 \cdot [1,2] \sqcup 0 \cdot [1,1]\big) = 5\big([1,2] \sqcup [0,0]\big) = 5[0,2]$. Then $\lambda \otimes (k_1 I_1 \sqcup k_2 I_2) = -2 \otimes (5[1,2] \sqcup 0[1,1]) = -2 \otimes 5[0,2] = 10[-2,0] = 10[-2,-1] \sqcup 0[1,1] = -2 \otimes 5[1,2] \sqcup -2 \otimes 0[1,1] = \lambda \otimes k_1 I_1 \sqcup \lambda \otimes k_2 I_2$.

```
def example12: Unit = {
  println("EXAMPLE 12")

  val λ = -2
  val kI1 = KIOne(5, Interval(1,2))
```

```
  val kI2 = KITwo(0, Interval(1,1))
  println(s"Consider λ = $λ, $kI1 = ${kI1.ki} and $kI2 = ${kI2.ki}.")
  println(s"First observe")
  println(s"$kI1 ⊔ $kI2 = ${kI1.ki} ⊔ ${kI2.ki} = ${kI1 ⊔ (kI2,"step1")})")
  println(s"            = ${kI1 ⊔ (kI2,"step2")} = ${kI1 ⊔ (kI2,"step3")} = ${kI1 ⊔
↪  kI2}")
  println(s"Then")
  println(s"λ ⊗ ($kI1 ⊔ $kI2) = $λ ⊗ (${kI1.ki} ⊔ ${kI2.ki}) = $λ ⊗ ${kI1 ⊔ kI2} = ${λ
↪  ⊗ (kI1 ⊔ kI2)}")
  assert(λ ⊗ (kI1 ⊔ kI2) == ((λ ⊗ kI1) ⊔ (λ ⊗ kI2)))
  println(s"                  = ${λ ⊗ kI1} ⊔ ${λ ⊗ kI2} = $λ ⊗ ${kI1.ki} ⊔ $λ ⊗
↪  ${kI2.ki})")
  println(s"                  = λ ⊗ $kI1 ⊔ λ ⊗ $kI2")
}
```

```
EXAMPLE 12
Consider λ = -2, k₁I₁ = 5[1,2] and k₂I₂ = 0[1,1].
First observe
k₁I₁ ⊔ k₂I₂ = 5[1,2] ⊔ 0[1,1] = 5(5/5 · [1,2] ⊔ 0/5 · [1,1]))
            = 5(1 · [1,2] ⊔ 0 · [1,1]) = 5([1,2] ⊔ [0,0]) = 5[0,2]
Then
λ ⊗ (k₁I₁ ⊔ k₂I₂) = -2 ⊗ (5[1,2] ⊔ 0[1,1]) = -2 ⊗ 5[0,2] = 10[-2,0]
                  = 10[-2,-1] ⊔ 0[1,1] = -2 ⊗ 5[1,2] ⊔ -2 ⊗ 0[1,1])
                  = λ ⊗ k₁I₁ ⊔ λ ⊗ k₂I₂
```

Example 13. To show how distributivity is applicable even when the scalar multipliers on the k-intervals differ, consider $\vec{v} = (16, 8)$, $\vec{w} = (12, 6)$, $\vec{x} = (4, 2)$, $\vec{y} = (2, 1)$. Observe $\vec{v}/\vec{x} = 4$ and $\vec{w}/\vec{x} = 3$ hence the scalar multipliers on k_1I_1 and k_2I_2 differ. For concreteness, let $k_1I_1 = -3[1, 1]$ and $k_2I_2 = 10[3, 4]$. Then $\vec{x}/\vec{y} = 2$, $\vec{v}/\vec{y} = 8$ and $\vec{w}/\vec{y} = 6$. Then $\vec{x}/\vec{y} \otimes \left(\vec{v}/\vec{x} \otimes k_1I_1 \sqcup \vec{w}/\vec{x} \otimes k_2I_2\right) = 2 \otimes (4 \otimes -3[1, 1] \sqcup 3 \otimes 10[3, 4]) = 2 \otimes (-12[1, 1] \sqcup 30[3, 4]) = 2 \otimes 6[-2, 20] = 12[-2, 20] = -24[1, 1] \sqcup 60[3, 4] = 8 \otimes -3[1, 1] \sqcup 6 \otimes 10[3, 4] = \vec{v}/\vec{y} \otimes k_1I_1 \sqcup \vec{w}/\vec{y} \otimes k_2I_2$.

```
def example13: Unit = {
  println("EXAMPLE 13")

  val (v, vecv) = (Vector(16,8), vec("v"))
  val (w, vecw) = (Vector(12,6), vec("w"))
  val (x, vecx) = (Vector(4,2), vec("x"))
  val (y, vecy) = (Vector(2,1), vec("y"))

  println(s"Observe $vecv/$vecx = ${v/x} and $vecw/$vecx = ${w/x}")
  val kI1 = KIOne(-3, Interval(1,1))
  val kI2 = KITwo(10, Interval(3,4))
  println(s"Let $kI1 = ${kI1.ki} and $kI2 = ${kI2.ki}.")
```

```
  println(s"Then $vecx/$vecy = ${x/y}, $vecv/$vecy = ${v/y} and $vecw/$vecy = ${w/y}.")
  print(s"Then $vecx/$vecy ⊗ ($vecv/$vecx ⊗ $kI1 ⊔ $vecw/$vecx ⊗ $kI2) = ")
  print(s"${x/y} ⊗ (${v/x} ⊗ ${kI1.ki} ⊔ ${w/x} ⊗ ${kI2.ki}) = ")
  print(s"${x/y} ⊗ (${(v/x) ⊗ kI1} ⊔ ${(w/x) ⊗ kI2}) = ")
  print(s"${x/y} ⊗ ${((v/x) ⊗ kI1) ⊔ ((w/x) ⊗ kI2)} = ")

  print((x/y) ⊗ (((v/x) ⊗ kI1) ⊔ ((w/x) ⊗ kI2)))
  assert((x/y) ⊗ (((v/x) ⊗ kI1) ⊔ ((w/x) ⊗ kI2)) == (((v/y) ⊗ kI1) ⊔ ((w/y) ⊗ kI2)))
  print(s" = ${(v/y) ⊗ kI1} ⊔ ${(w/y) ⊗ kI2}")
  println(s" = ${v/y} ⊗ ${kI1.ki} ⊔ ${w/y} ⊗ ${kI2.ki} = $vecv/$vecy ⊗ $kI1 ⊔
↪  $vecw/$vecy ⊗ $kI2")
}
```

```
EXAMPLE 13
Observe v/x = 4 and w/x = 3
Let k₁I₁ = -3[1,1] and k₂I₂ = 10[3,4].
Then x/y = 2, v/y = 8 and w/y = 6.
Then x/y ⊗ (v/x ⊗ k₁I₁ ⊔ w/x ⊗ k₂I₂) = 2 ⊗ (4 ⊗ -3[1,1] ⊔ 3 ⊗ 10[3,4]) = 2 ⊗
↪  (-12[1,1] ⊔ 30[3,4]) = 2 ⊗ 6[-2,20] = 12[-2,20] = -24[1,1] ⊔ 60[3,4] = 8 ⊗
↪  -3[1,1] ⊔ 6 ⊗ 10[3,4] = v/y ⊗ k₁I₁ ⊔ w/y ⊗ k₂I₂
```

Example 14. To illustrate the cardinality property described above, consider $kI\vec{v} \in \mathsf{Range}^n$ where $kI = 2[1,3]$, $n = 2$ and $\vec{v} = (1,5)$. Observe $\vec{v}$ is irreducible since if $\vec{v} = \lambda\vec{w}$ where $\vec{w} \in \mathbb{Z}^n$ then $\lambda = \pm 1$. Then $|\gamma(I)| = |\{1,2,3\}| = 3$. Observe $|\gamma(kI\vec{v})| = |\{2\vec{v}, 4\vec{v}, 6\vec{v}\}| = 3$.

```
def example14: Unit = {
  println("EXAMPLE 14")

  val kI = KI(2, Interval(1,3))
  val (v, vecv) = (Vector(1,5), vec("v"))
  val kIV = KIV(kI, v)
  println(s"Consider kI = $kI, n = 2 and v = $v.")
  println(s"Observe $vecv is irreducible since its divisors are ${v.divisors}")
  assert(v.divisors == Set(-1,1))
  println(s"Then |γ(I)| = |${`γ`(kIV.i)}| = " + cardinality(`γ`(kIV.i)))
  println(s"Observe |γ(kIV)| = |{2*$v, 4*$v, 6*$v}| = " + cardinality(`γ`(kIV)))
}
```

```
EXAMPLE 14
Consider kI = 2[1,3], n = 2 and v = Vector(1, 5).
Observe v is irreducible since its divisors are TreeSet(-1, 1)
Then |γ(I)| = |TreeSet(1, 2, 3)| = 3
Observe |γ(kIV)| = |{2*Vector(1, 5), 4*Vector(1, 5), 6*Vector(1, 5)}| = 3
```

Example 15. To illustrate the lemma, suppose $k(\lambda + I)\vec{v} \in \mathsf{Range}^n$ where $k = 2$, $\lambda = 5$, $I = [1, 2]$ and $\vec{v} \in \mathbb{Z}^n \setminus \{\vec{0}\}$. Observe $k(\lambda + I)\vec{v} = 2(5 + [1, 2])\vec{v} = 2[6, 7]\vec{v} \in \mathsf{Range}^n$. Then $\gamma(k(\lambda + I)\vec{v}) = \gamma(2[6, 7]\vec{v}) = \{12\vec{v}, 14\vec{v}\} = \{10\vec{v} + 2\vec{v}, 10\vec{v} + 4\vec{v}\} = \{10\vec{v} + \vec{x} \mid \vec{x} \in \gamma(2[1, 2]\vec{v})\} = \{\lambda k\vec{v} + \vec{x} \mid \vec{x} \in \gamma(kI\vec{v})\}$.

```
def example15: Unit = {
  println("EXAMPLE 15")
  val k = 2
  val λ = 5
  val I = Interval(1,2)
  val (v, vecv) = (Vector(1,1), vec("v"))
  println(s"Suppose k(λ + I)v where k = $k, λ = $λ, I = $I and v = $v (arbitrarily)")
  val vecx = vec("x")
  val kIVOffset = KIV(KI(k, λ + I), v)
  assert(kIVOffset.isInstanceOf[RangeDomain])
  val kIV = KIV(KI(k, I), v)
  println(s"Observe k(λ + I)$vecv = $k($λ + $I)$vecv = $kIVOffset")
  print(s"Then γ(k(λ + I)$vecv) = γ($kIVOffset) = " + `γ`(kIVOffset))
  print(s" = {${10 * v} + ${2 * v}, ${10 * v} + ${4 * v}}")
  println(s" = {10$vecv + $vecx | $vecx ∈ γ($kIV)} = {λk$vecv + $vecx | $vecx ∈
↪  γ(kIV)}")

  assert(`γ`(kIVOffset) == (for (x <- `γ`(kIV)) yield λ*k*v + x))
}
```

```
EXAMPLE 15
Suppose k(λ + I)v where k = 2, λ = 5, I = [1,2] and v = Vector(1, 1) (arbitrarily)
Observe k(λ + I)v = 2(5 + [1,2])v = 2[6,7](1,1)
Then γ(k(λ + I)v) = γ(2[6,7](1,1)) = TreeSet(Vector(12, 12), Vector(14, 14)) =
↪  {Vector(10, 10) + Vector(2, 2), Vector(10, 10) + Vector(4, 4)} = {10v + x | x ∈
↪  γ(2[1,2](1,1))} = {λkv + x | x ∈ γ(kIV)}
```

Example 16. To illustrate the lemma, consider $kI\vec{v} \in \mathsf{Range}^n$ where $kI = 2[1, 2]$ and $\vec{v} = (-6, 3)$, so $k\vec{v} = (-12, 6)$. Also suppose $\vec{z} = (4, -2)$. Observe $\vec{z} | k\vec{v}$ since $-3\vec{z} = k\vec{v}$, but note $\vec{z} \nmid \vec{v}$. Then

$$\begin{aligned}
\gamma(kI\vec{v}) &= \gamma(2[1, 2](-6, 3)) \\
&= \{(-12, 6), (-24, 12)\} \\
&= \gamma(3[-2, -1](4, -2)) \\
&= \gamma((-3 \otimes 1[1, 2])(4, -2)) \\
&= \gamma(((-12, 6)/(4, -2) \otimes 1[1, 2])(4, -2)) \\
&= \gamma(((k\vec{v})/\vec{z} \otimes 1I)\vec{z})
\end{aligned}$$

Thus $kI\vec{v} \equiv ((k\vec{v})/\vec{z} \otimes 1I)\vec{z}$, which concurs with the lemma.

```
def example16: Unit = {
  println("EXAMPLE 16")
  val I = Interval(1,2)
  val kI = KI(2, I)
  val (v, vecv) = (Vector(-6, 3), vec("v"))
  val (kv, veckv) = (kI.k * v, vec("kv"))
  val (z, vecz) = (Vector(4, -2), vec("z"))

  val kIV = KIV(kI, v)
  val oneI = KI(1, I)

  print(s"Consider kIV ∈ RangeN")
  assert(kIV.isInstanceOf[RangeDomain])
  println(s" where kI = $kI and $vecv = $v, so $veckv = $kv.")
  print(s"Also suppose $vecz = $z. Observe $vecz | $veckv")
  assert(z `|` kv)
  print(s" since -3$vecz = $kv")
  assert(-3 * z == kv)
  println(s" but note $vecz ∤ $vecv.")
  assert(z `∤` v)
  println(s"Then γ(kIV) = γ($kIV)")
  println(s"            = ${`γ`(kIV)}")
  println(s"            = γ(${KIV((kv/z) ⊗ oneI, z)})")
  println(s"            = γ((${kv/z} ⊗ $oneI)$z)")
  println(s"            = γ(($kv/$z ⊗ $oneI)$z)")
  println(s"            = γ((k$vecv)/$vecz ⊗ 1I)$vecz")

  assert(`γ`(kIV) == `γ`(KIV((kv/z) ⊗ KI(1,I), z)) )
}
```

```
EXAMPLE 16
Consider kIV ∈ RangeN where kI = 2[1,2] and v = Vector(-6, 3), so kv = Vector(-12,
↪  6).
Also suppose z = Vector(4, -2). Observe z | kv since -3z = Vector(-12, 6) but note z
↪  ∤ v.
Then γ(kIV) = γ(2[1,2](-6,3))
            = TreeSet(Vector(-24, 12), Vector(-12, 6))
            = γ(3[-2,-1](4,-2))
            = γ((-3 ⊗ 1[1,2])Vector(4, -2))
            = γ((Vector(-12, 6)/Vector(4, -2) ⊗ 1[1,2])Vector(4, -2))
            = γ((kv)/z ⊗ 1I)z
```

Example 17. Consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_2 = 1[-1,0](3,3)$. Since $gcd(1(1,1), 1(3,3)) = (1,1)$ and, using the result from example 6 in the final

step, then:

$$\begin{aligned} r_1 \sqcup r_2 &= 1[-3,-1](1,1) \sqcup 1[-1,0](3,3) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (3,3)/(1,1) \otimes 1[-1,0]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 3 \otimes 1[-1,0]\big)(1,1) \\ &= (1[-3,-1] \sqcup 3[-1,0])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

The two k-intervals are scaled up so that they share a common vector (1,1) and then are joined. The scaling of the k-multiples reflects the factor by which the original vectors (1,1) and (3,3) are scaled down, with the $\otimes$ operator ensuring the resultant k-intervals are correctly defined. The result is shown in figure 12.

```
def example17: Unit = {
  println("EXAMPLE 17")
  val r1 = KIV(KI(1, Interval(-3,-1)), Vector(1,1))
  val r2 = KIV(KI(1, Interval(-1,0)), Vector(3,3))
  println(s"Consider the ranges r1 = $r1 and r2 = $r2.")
  println(s"Since gcd(${r1.k} * ${r1.v}, ${r2.k} * ${r2.v}) = " + gcdVectors(r1.k *
↪  r1.v, r2.k * r2.v).get)
  println(s"r1 ⊔ r2 = $r1 ⊔ $r2")
  println(s"        = ${r1 ⊔ (r2, "step1")}")
  println(s"        = ${r1 ⊔ (r2, "step2")}")
  println(s"        = ${r1 ⊔ (r2, "step3")}")
  println(s"        = ${r1 ⊔ r2}")
}
```

```
EXAMPLE 17
Consider the ranges r1 = 1[-3,-1](1,1) and r2 = 1[-1,0](3,3).
Since gcd(1 * Vector(1, 1), 1 * Vector(3, 3)) = Vector(1, 1)
r1 ⊔ r2 = 1[-3,-1](1,1) ⊔ 1[-1,0](3,3)
        = (Vector(1, 1)/Vector(1, 1) ⊗ 1[-3,-1] ⊔ Vector(3, 3)/Vector(1, 1) ⊗
        ↪  1[-1,0])Vector(1, 1)
        = (1 ⊗ 1[-3,-1] ⊔ 3 ⊗ 1[-1,0])Vector(1, 1)
        = (1[-3,-1] ⊔ 3[-1,0])Vector(1, 1)
        = 1[-3,0](1,1)
```

Example 18. Now consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_3 = 1[-1,0](2,2)$. Since $gcd(1(1,1), 1(2,2)) = (1,1)$ and, using the result from example 7 in the final

step, then:

$$\begin{aligned} r_1 \sqcup r_3 &= 1[-3,-1](1,1) \sqcup 1[-1,0](2,2) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (2,2)/(1,1) \otimes 1[-1,0]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 2 \otimes 1[-1,0]\big)(1,1) \\ &= (1[-3,-1] \sqcup 2[-1,0])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

Again the result is the same as that shown in figure 12 and the result is analogous to example 7 with the number line orientated along $(1,1)$.

```
def example18: Unit = {
  println("EXAMPLE 18")
  val r1 = KIV(KI(1, Interval(-3,-1)), Vector(1,1))
  val r3 = KIV(KI(1, Interval(-1,0)), Vector(2,2))
  println(s"Consider the ranges r1 = $r1 and r3 = $r3")
  println(s"Since gcd(${r1.k} * ${r1.v}, ${r3.k} * ${r3.v}) = " + gcdVectors(r1.k *
↪  r1.v, r3.k * r3.v).get)
  println(s"r1 ⊔ r3 = $r1 ⊔ $r3 ")
  println(s"        = ${r1 ⊔ (r3, "step1")}")
  println(s"        = ${r1 ⊔ (r3, "step2")}")
  println(s"        = ${r1 ⊔ (r3, "step3")}")
  println(s"        = ${r1 ⊔ r3}")
}
```

```
EXAMPLE 18
Consider the ranges r1 = 1[-3,-1](1,1) and r3 = 1[-1,0](2,2)
Since gcd(1 * Vector(1, 1), 1 * Vector(2, 2)) = Vector(1, 1)
r1 ⊔ r3 = 1[-3,-1](1,1) ⊔ 1[-1,0](2,2)
        = (Vector(1, 1)/Vector(1, 1) ⊗ 1[-3,-1] ⊔ Vector(2, 2)/Vector(1, 1) ⊗
        ↪  1[-1,0])Vector(1, 1)
        = (1 ⊗ 1[-3,-1] ⊔ 2 ⊗ 1[-1,0])Vector(1, 1)
        = (1[-3,-1] ⊔ 2[-1,0])Vector(1, 1)
        = 1[-3,0](1,1)
```

Example 19. To continue the development, consider the ranges $r_1 = 1[-3,-1](1,1)$ and $r_4 = 0[1,1](1,1)$. Since $gcd(1(1,1), 0(1,1)) = (1,1)$ and, using the result from

example 8 in the final step, then:

$$\begin{aligned} r_1 \sqcup r_4 &= 1[-3,-1](1,1) \sqcup 0[1,1](1,1) \\ &= \big((1,1)/(1,1) \otimes 1[-3,-1] \sqcup (0,0)/(1,1) \otimes 1[1,1]\big)(1,1) \\ &= \big(1 \otimes 1[-3,-1] \sqcup 0 \otimes 1[1,1]\big)(1,1) \\ &= (1[-3,-1] \sqcup 0[1,1])(1,1) \\ &= 1[-3,0](1,1) \end{aligned}$$

```
def example19: Unit = {
  println("EXAMPLE 19")
  val r1 = KIV(KI(1, Interval(-3,-1)), Vector(1,1))
  val r4 = KIV(KI(0, Interval(1,1)), Vector(1,1))
  println(s"Consider the ranges r1 = $r1 and r4 = $r4")
  println(s"Since gcd(${r1.k} * ${r1.v}, ${r4.k} * ${r4.v}) = " + gcdVectors(r1.k *
↪  r1.v, r4.k * r4.v).get)
  println(s"r1 ⊔ r4 = $r1 ⊔ $r4")
  println(s"        = ${r1 ⊔ (r4, "step1")}")
  println(s"        = ${r1 ⊔ (r4, "step2")}")
  println(s"        = ${r1 ⊔ (r4, "step3")}")
  println(s"        = ${r1 ⊔ r4}")
}
```

```
EXAMPLE 19
Consider the ranges r1 = 1[-3,-1](1,1) and r4 = 0[1,1](1,1)
Since gcd(1 * Vector(1, 1), 0 * Vector(1, 1)) = Vector(1, 1)
r1 ⊔ r4 = 1[-3,-1](1,1) ⊔ 0[1,1](1,1)
        = (Vector(1, 1)/Vector(1, 1) ⊗ 1[-3,-1] ⊔ Vector(0, 0)/Vector(1, 1) ⊗
        ↪  1[1,1])Vector(1, 1)
        = (1 ⊗ 1[-3,-1] ⊔ 0 ⊗ 1[1,1])Vector(1, 1)
        = (1[-3,-1] ⊔ 0[1,1])Vector(1, 1)
        = 1[-3,0](1,1)
```

Example 20. Now consider the ranges $r_5 = -3[1,1](1,1)$ and $r_4 = 0[1,1](1,1)$. Since $gcd(-3(1,1), 0(1,1)) = (3,3)$, then:

$$\begin{aligned} r_5 \sqcup r_4 &= -3[1,1](1,1) \sqcup 0[1,1](1,1) \\ &= \big((-3,-3)/(3,3) \otimes 1[1,1] \sqcup (0,0)/(3,3) \otimes 1[1,1]\big)(3,3) \\ &= \big(-1 \otimes 1[1,1] \sqcup 0 \otimes 1[1,1]\big)(3,3) \\ &= (-1[1,1] \sqcup 0[1,1])(3,3) \\ &= 1[-1,0](3,3) \end{aligned}$$

```
def example20: Unit = {
  println("EXAMPLE 20")
  val r5 = KIV(KI(-3, Interval(1,1)), Vector(1,1))
  val r4 = KIV(KI(0, Interval(1,1)), Vector(1,1))
  println(s"Consider the ranges r5 = $r5 and r4 = $r4")
  println(s"Since gcd(${r5.k} * ${r5.v}, ${r4.k} * ${r4.v}) = " + gcdVectors(r5.k *
↪  r5.v, r4.k * r4.v).get)
  println(s"r5 ⊔ r4 = $r5 ⊔ $r4")
  println(s"        = ${r5 ⊔ (r4, "step1")}")
  println(s"        = ${r5 ⊔ (r4, "step2")}")
  println(s"        = ${r5 ⊔ (r4, "step3")}")
  println(s"        = ${r5 ⊔ r4}")
}
```

```
Consider the ranges r5 = -3[1,1](1,1) and r4 = 0[1,1](1,1)
Since gcd(-3 * Vector(1, 1), 0 * Vector(1, 1)) = Vector(3, 3)
r5 ⊔ r4 = -3[1,1](1,1) ⊔ 0[1,1](1,1)
        = (Vector(-3, -3)/Vector(3, 3) ⊗ 1[1,1] ⊔ Vector(0, 0)/Vector(3, 3) ⊗
        ↪  1[1,1])Vector(3, 3)
        = (-1 ⊗ 1[1,1] ⊔ 0 ⊗ 1[1,1])Vector(3, 3)
        = (-1[1,1] ⊔ 0[1,1])Vector(3, 3)
        = 1[-1,0](3,3)
```

Example 21. Finally consider the ranges $r_6 = 1[1,3](0,1)$ and $r_7 = 1[0,1](0,2)$. Since $gcd(1(0,1), 1(0,2)) = (0,1)$, then:

$$\begin{aligned} r_6 \sqcup r_7 &= 1[1,3](0,1) \sqcup 1[0,1](0,2) \\ &= \big((0,1)/(0,1) \otimes 1[1,3] \sqcup (0,2)/(0,1) \otimes 1[0,1]\big)(0,1) \\ &= \big(1 \otimes 1[1,3] \sqcup 2 \otimes 1[0,1]\big)(0,1) \\ &= (1[1,3] \sqcup 2[0,1])(0,1) \\ &= 1[0,3](0,1) \end{aligned}$$

```
def example21: Unit = {
  println("EXAMPLE 21")
  val r6 = KIV(KI(1, Interval(1,3)), Vector(0,1))
  val r7 = KIV(KI(1, Interval(0,1)), Vector(0,2))
  println(s"Finally consider the ranges r6 = $r6 and r7 = $r7")
  println(s"Since gcd(${r6.k} * ${r6.v}, ${r7.k} * ${r7.v}) = " + gcdVectors(r6.k *
↪  r6.v, r7.k * r7.v).get)
  println(s"r6 ⊔ r7 = $r6 ⊔ $r7")
  println(s"        = ${r6 ⊔ (r7, "step1")}")
  println(s"        = ${r6 ⊔ (r7, "step2")}")
  println(s"        = ${r6 ⊔ (r7, "step3")}")
```

```
  println(s"        = ${r6 ⊔ r7}")
}
```

```
EXAMPLE 21
Finally consider the ranges r6 = 1[1,3](0,1) and r7 = 1[0,1](0,2)
Since gcd(1 * Vector(0, 1), 1 * Vector(0, 2)) = Vector(0, 1)
r6 ⊔ r7 = 1[1,3](0,1) ⊔ 1[0,1](0,2)
        = (Vector(0, 1)/Vector(0, 1) ⊗ 1[1,3] ⊔ Vector(0, 2)/Vector(0, 1) ⊗
        ↪  1[0,1])Vector(0, 1)
        = (1 ⊗ 1[1,3] ⊔ 2 ⊗ 1[0,1])Vector(0, 1)
        = (1[1,3] ⊔ 2[0,1])Vector(0, 1)
        = 1[0,3](0,1)
```

Example 22. Figure 13 uses the ranges $k_{10}I_{10}\vec{1}$, $k_{11}I_{11}\vec{1}$, $k_{12}I_{12}\vec{1}$ and $k_{13}I_{13}\vec{1}$ to illustrate this. The ranges in part (a), namely $k_{10}I_{10}\vec{1}$ and $k_{11}I_{11}\vec{1}$, represent two points, one at -6 and the other at -3. The two ranges in part (b), $k_{12}I_{12}\vec{1}$ and $k_{13}I_{13}\vec{1}$, likewise represent two points, -5 and -2, equally separated by a difference of 3. Observe that the points in part (b) are the merely those of part (a) translated by one unit. However, the joins differ dramatically. The join of $k_{10}I_{10}\vec{1}$ and $k_{11}I_{11}\vec{1}$ introduces no new points, whereas the join of $k_{11}I_{11}\vec{1}$ and $k_{13}I_{13}\vec{1}$ results in a range with double the number of points. This is because 3 divides both -3 and -6 but 1 is the largest divisor of -5 and -2. Put another way, by adding multiples of 3 to both -6 and -3 it is possible to reach the origin. Likewise, it is possible to reach the origin by adding multiples of 1 to both -5 and -2. It is this requirement to anchor a range at the origin that induces a loss of information in part (b).

```
def example22: Unit = {
  println("EXAMPLE 22")

  val ioneone = Interval(1,1)
  val vone = Vector(1)
  val kIFourOne = new NamedKIV(KIFour(-6, ioneone), vone, vec("1"))
  val kIFiveOne = new NamedKIV(KIFive(-3, ioneone), vone, vec("1"))
  val kITenOne = new NamedKIV(KITen(-5, ioneone), vone, vec("1"))
  val kIElevenOne = new NamedKIV(KIEleven(-2, ioneone), vone, vec("1"))

  println("Ranges of part a)")
  println(s"γ($kIFourOne) = ${`γ`(kIFourOne)}")
  println(s"γ($kIFiveOne) = ${`γ`(kIFiveOne)}")

  println("Ranges of part b)")
  println(s"γ($kITenOne) = ${`γ`(kITenOne)}")
  println(s"γ($kIElevenOne) = ${`γ`(kIElevenOne)}")
```

```
  println("Join of part a)")
  println(s"$kIFourOne ⊔ $kIFiveOne = ${kIFourOne.kiv} ⊔ ${kIFiveOne.kiv} = " +
↪  (kIFourOne ⊔ kIFiveOne))
  println("Points: " + `γ`(kIFourOne ⊔ kIFiveOne))

  println("Join of part b)")
  println(s"$kITenOne ⊔ $kIElevenOne = ${kITenOne.kiv} ⊔ ${kIElevenOne.kiv} = " +
↪  (kITenOne ⊔ kIElevenOne))
  println("Points: " + `γ`(kITenOne ⊔ kIElevenOne))
}
```

```
EXAMPLE 22

Ranges of part a)
γ(k₄I₄1) = TreeSet(Vector(-6))
γ(k₅I₅1) = TreeSet(Vector(-3))
Ranges of part b)
γ(k₁₀I₁₀1) = TreeSet(Vector(-5))
γ(k₁₁I₁₁1) = TreeSet(Vector(-2))
Join of part a)
k₄I₄1 ⊔ k₅I₅1 = -6[1,1](1) ⊔ -3[1,1](1) = 1[-2,-1](3)
Points: TreeSet(Vector(-6), Vector(-3))
Join of part b)
k₁₀I₁₀1 ⊔ k₁₁I₁₁1 = -5[1,1](1) ⊔ -2[1,1](1) = 1[-5,-2](1)
Points: TreeSet(Vector(-5), Vector(-4), Vector(-3), Vector(-2))
```

Example 23. Continuing with example 23, observe how the points -5 and -2 can be represented by $\vec{-8} + 3[1,2]\vec{1}$. The role of the $\vec{-8}$ is to define a translation of the points that gives a new set of points which are representable as a range whilst also relaxing the anchoring requirement.

```
def example23: Unit = {
  println("EXAMPLE 23")

  val m = March(Vector(-8), KIV(KI(3,Interval(1,2)), Vector(1)))
  println(s"Observe how the points -5 and -2 can be represented by $m.")
  println(s"γ(m) = " + `γ`(m))
}
```

```
EXAMPLE 23
Observe how the points -5 and -2 can be represented by (-8) + 3[1,2](1).
γ(m) = TreeSet(Vector(-5), Vector(-2))
```

Example 24. To illustrate march translation, let $m_1 = (2,3) + 1[0,1](0,-2)$ and $\vec{v} = (2,0)$. Observe $gcd((0,3),(0,-2)) = (0,1)$. Then:

$$\begin{aligned} m_1 \ominus \vec{v} &= (((2,3) - (2,0)) + 1[0,1](0,-2))^{-1} \\ &= ((0,3) + 1[0,1](0,-2))^{-1} \\ &= 1((0,3)/(0,1) + (0,-2)/(0,1) \cdot [0,1])(0,1) \\ &= 1(3 + (-2) \cdot [0,1])(0,1) \\ &= 1(3 + [-2,0])(0,1) \\ &= 1[1,3](0,1) \end{aligned}$$

Figure 14 illustrates this march translation. Observe $\gamma(m_1 \ominus \vec{v}) = \gamma(1[1,3](0,1)) = \{(0,1),(0,2),(0,3)\}$. Hence although m_1 describes two points, its translation, $m_1 \ominus \vec{v}$, describes three, demonstrating that translation can induce a loss of information.

```
val m1 = new NamedMarch(Vector(2,3), KIV(KI(1, Interval(0,1)), Vector(0,-2)), 1)
def example24: Unit = {
  println("EXAMPLE 24")
  val v = Vector(2,0)
  println(s"Let $m1 = ${m1.m} and v = $v.")
  println(s"See graph: γ($m1) = ${`γ`(m1)}")
  println(s"Observe gcd(${m1.s - v}, ${m1.k * m1.v}) = ${gcdVectors(m1.s - v, m1.k *
↪  m1.v).get}. Then")
  println(m1.allSteps(m1.toString, "v", v))
}
```

```
EXAMPLE 24
Let m₁ = (2,3) + 1[0,1](0,-2) and v = Vector(2, 0).
See graph: γ(m₁) = TreeSet(Vector(2, 1), Vector(2, 3))
Observe gcd(Vector(0, 3), Vector(0, -2)) = Vector(0, 1). Then
m₁ ⊖ v = (((2,3) - (2,0)) + 1[0,1](0,-2))⁻¹
       = ((0,3) + 1[0,1](0,-2))⁻¹
       = 1((0,3)/(0,1) + (0,-2)/(0,1) · [0,1])(0,1)
       = 1(3 + -2 · [0,1])(0,1)
       = 1(3 + [-2,0])(0,1)
       = 1[1,3](0,1)

See graph: γ(m₁ ⊖ v) = TreeSet(Vector(0, 1), Vector(0, 2), Vector(0, 3))
```

Example 25. As a further demonstration of loss of information, consider $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} = (2,-2) + 2[1,2](0,1)$ and suppose $\vec{v} = (1,3)$, as shown in figure 15.

Observe $I_2 \neq [1,1]$ and $gcd(\vec{s}_2 - \vec{v}, k_2\vec{y}) = gcd((1,-5),(0,2)) = \perp$, hence case 4 of definition 17 applies. Thus:

$$\begin{aligned} m_2 \ominus \vec{v} &= (((2,-2) - (1,3)) + 2[1,2](0,1))^{-1} \\ &= ((1,-5) + 2[1,2](0,1))^{-1} \\ &= \top \end{aligned}$$

Observe the translated points of m_2 do not lie on a line through the origin, so the only range that can enclose them is $\top$.

```
val m2 = new NamedMarch(Vector(2,-2), KIV(KI(2, Interval(1,2)), Vector(0,1)), 2)
def example25: Unit = {
  println("EXAMPLE 25")

  val v = Vector(1,3)
  println(s"Consider $m2 = ${m2.m} and suppose v = $v.")
  println(s"See graph: γ(m₂) = ${`γ`(m2)}")
  println(s"Observe I₂ ≠ [1,1] and gcd(s₂ - v, k₂y) = gcd(${m2.s - v}, ${m2.k * m2.v})
↪   = ${gcdVectors(m2.s - v, m2.k * m2.v).getOrElse("⊥")}")
  println(m2.allSteps("m₂", "v", v))
}
```

```
EXAMPLE 25
Consider m₂ = (2,-2) + 2[1,2](0,1) and suppose v = Vector(1, 3).
See graph: γ(m₂) = TreeSet(Vector(2, 0), Vector(2, 2))
Observe I₂ ≠ [1,1] and gcd(s₂ - v, k₂y) = gcd(Vector(1, -5), Vector(0, 2)) = ⊥
m₂ ⊖ v = (((2,-2) - (1,3)) + 2[1,2](0,1))⁻¹
       = ((1,-5) + 2[1,2](0,1))⁻¹
       = ⊤
See graph: γ(m₂ ⊖ v) = Z^N
```

Example 26. To illustrate that information is not necessarily lost, consider $m_3 = \vec{s}_3 + k_3 I_3 \vec{m} = (3,-2) + 1[0,2](-1,2)$ and $\vec{v} = (2,0)$. Observe $gcd((1,-2),(-1,2)) =$

$(1,-2)$. Then:

$$\begin{aligned} m_3 \ominus \vec{v} &= (((3,-2)-(2,0))+1[0,2](-1,2))^{-1} \\ &= ((1,-2)+1[0,2](-1,2))^{-1} \\ &= 1((1,-2)/(1,-2)+(-1,2)/(1,-2)\cdot[0,2])(1,-2) \\ &= 1(1+(-1)\cdot[0,2])(1,-2) \\ &= 1(1+[-2,0])(1,-2) \\ &= 1[-1,1](1,-2) \end{aligned}$$

Note $\gamma(m_3 \ominus \vec{v}) = \gamma(1[-1,1](1,-2)) = \{(-1,2),(0,0),(1,-2)\}$. Since $k_3\vec{m}$ is a multiple of $\vec{s}_3 - \vec{v}$, dimension reduction incurs no loss of information. The resulting translation is shown in figure 16.

```
val m3 = new NamedMarch(Vector(3,-2), KIV(KI(1, Interval(0,2)), Vector(-1,2)), 3)
def example26: Unit = {
  println("EXAMPLE 26")

  val v = Vector(2,0)
  println(s"Consider $m3 = ${m3.m} and v = $v.")
  println(s"See graph: γ(m₃) = ${`γ`(m3)}")
  println(s"Observe gcd(${m3.s - v}, ${m3.k * m3.v}) = ${gcdVectors(m3.s - v,m3.k *
↪  m3.v).getOrElse("⊥")}")
  println(m3.allSteps("m₃", "v", v))
}
```

```
EXAMPLE 26
Consider m₃ = (3,-2) + 1[0,2](-1,2) and v = Vector(2, 0).
See graph: γ(m₃) = TreeSet(Vector(1, 2), Vector(2, 0), Vector(3, -2))
Observe gcd(Vector(1, -2), Vector(-1, 2)) = Vector(1, -2)
m₃ ⊖ v = (((3,-2) - (2,0)) + 1[0,2](-1,2))⁻¹
       = ((1,-2) + 1[0,2](-1,2))⁻¹
       = 1((1,-2)/(1,-2) + (-1,2)/(1,-2) · [0,2])(1,-2)
       = 1(1 + -1 · [0,2])(1,-2)
       = 1(1 + [-2,0])(1,-2)
       = 1[-1,1](1,-2)

See graph: γ(m₃ ⊖ v) = TreeSet(Vector(-1, 2), Vector(0, 0), Vector(1, -2))
```

Example 27. Consider the degenerate march $m_4 = \vec{s}_4 + k_4 I_4 \vec{n} = (2,0) + 0[1,1]\vec{1}$ and suppose $\vec{v} = (2,0)$. Since $I_4 = [1,1]$ and $(\vec{s}_4 - \vec{v}) + k_4\vec{v}_4 = \vec{0}$, the first case of

definition 17 applies:

$$\begin{aligned} m_4 \ominus \vec{v} &= (((2,0) - (2,0)) + 0[1,1]\vec{1})^{-1} \\ &= ((0,0) + 0[1,1]\vec{1})^{-1} \\ &= 0[1,1]\vec{1} \end{aligned}$$

The initial and translated march both represent a single point.

```
val m4 = new NamedMarch(Vector(2,0), KIV(KI(0, Interval(1,1)), Vector(1,1)), 4)
def example27: Unit = {
  println("EXAMPLE 27")

  val v = Vector(2,0)
  println(s"Consider the degenerate march $m4 = ${m4.m} and suppose v = $v.")
  println(s"See graph: γ(m₄) = ${`γ`(m4)}")
  println("Since I₄ = [1,1] and (s₄ - v) + k₄v₄ = 0, the first case of definition 16
↪  applies:")
  assert((m4.s - v) + (m4.k * m4.v) == Vector(0,0))
  println(m4.allSteps("m₄", "v", v))
}
```

```
EXAMPLE 27
Consider the degenerate march m₄ = (2,0) + 0[1,1](1,1) and suppose v = Vector(2, 0).
See graph: γ(m₄) = TreeSet(Vector(2, 0))
Since I₄ = [1,1] and (s₄ - v) + k₄v₄ = 0, the first case of definition 16 applies:
m₄ ⊖ v = (((2,0) - (2,0)) + 0[1,1](1,1))⁻¹
       = ((0,0) + 0[1,1](1,1))⁻¹
       = 0[1,1](1,1)
See graph: γ(m₄ ⊖ v) = TreeSet(Vector(0, 0))
```

Example 28. As a final example of march translation, which provides scaffolding for the development that follows, consider $m_5 = \vec{s_5} + k_5 I_5 \vec{p} = (-1,-2) + 1[0,2](2,2)$ and $m_6 = \vec{s_6} + k_6 I_6 \vec{q} = (4,3) + 1[0,1](-4,-4)$. Now suppose $\vec{v} = \vec{s_6} + k_6\vec{q} = (4,3) + 1(-4,-4) = (0,-1)$. Observe $gcd((-1,-1),(2,2)) = (1,1)$ and

$gcd((4,4),(-4,-4)) = (4,4)$. Then:

$$\begin{aligned} m_5 \ominus \vec{v} &= (((-1,-2) - (0,-1)) + 1[0,2](2,2))^{-1} \\ &= ((-1,-1) + 1[0,2](2,2))^{-1} \\ &= 1((-1,-1)/(1,1) + (2,2)/(1,1) \cdot [0,2])(1,1) \\ &= 1(-1 + 2 \cdot [0,2])(1,1) \\ &= 1(-1 + [0,4])(1,1) \\ &= 1[-1,3](1,1) \end{aligned}$$

and

$$\begin{aligned} m_6 \ominus \vec{v} &= (((4,3) - (0,-1)) + 1[0,1](-4,-4))^{-1} \\ &= ((4,4) + 1[0,1](-4,-4))^{-1} \\ &= 1((4,4)/(4,4) + (-4,-4)/(4,4) \cdot [0,1])(4,4) \\ &= 1(1 + (-1) \cdot [0,1])(4,4) \\ &= 1(1 + [-1,0])(4,4) \\ &= 1[0,1](4,4) \end{aligned}$$

```
val m5 = new NamedMarch(Vector(-1,-2), KIV(KI(1, Interval(0,2)), Vector(2,2)), 5)
val m6 = new NamedMarch(Vector(4,3), KIV(KI(1, Interval(0,1)), Vector(-4,-4)), 6)
def example28:  Unit = {
  println("EXAMPLE 28")

  println(s"Consider $m5 = ${m5.m} and $m6 = ${m6.m}.")
  val v = m6.s + m6.k * m6.v
  println(s"Since I₅ ≠ [1,1], then v = s₆ + k₆q = ${m6.s} + ${m6.k}*${m6.v} = $v.")
  print(s"Observe gcd(${m5.s - v}, ${m5.k * m5.v}) = ${gcdVectors(m5.s - v,m5.k *
↪  m5.v).getOrElse("⊥")}")
  println(s" and gcd(${m6.s - v}, ${m6.k * m6.v}) = ${gcdVectors(m6.s - v,m6.k *
↪  m6.v).getOrElse("⊥")}.")

  println("Then")
  println(m5.allSteps("m₅", "v", v))
  println("and")
  println(m6.allSteps("m₆", "v", v))
}
```

```
EXAMPLE 28
Consider m₅ = (-1,-2) + 1[0,2](2,2) and m₆ = (4,3) + 1[0,1](-4,-4).
```

```
Since I₅ ≠ [1,1], then v = s₆ + k₆q = Vector(4, 3) + 1*Vector(-4, -4) = Vector(0,
↪  -1).
Observe gcd(Vector(-1, -1), Vector(2, 2)) = Vector(1, 1) and gcd(Vector(4, 4),
↪  Vector(-4, -4)) = Vector(4, 4).
Then
m₅ ⊖ v = (((-1,-2) - (0,-1)) + 1[0,2](2,2))⁻¹
       = ((-1,-1) + 1[0,2](2,2))⁻¹
       = 1((-1,-1)/(1,1) + (2,2)/(1,1) · [0,2])(1,1)
       = 1(-1 + 2 · [0,2])(1,1)
       = 1(-1 + [0,4])(1,1)
       = 1[-1,3](1,1)

See graph: γ(m₅ ⊖ v) = TreeSet(Vector(-1, -1), Vector(0, 0), Vector(1, 1), Vector(2,
↪  2), Vector(3, 3))
and
m₆ ⊖ v = (((4,3) - (0,-1)) + 1[0,1](-4,-4))⁻¹
       = ((4,4) + 1[0,1](-4,-4))⁻¹
       = 1((4,4)/(4,4) + (-4,-4)/(4,4) · [0,1])(4,4)
       = 1(1 + -1 · [0,1])(4,4)
       = 1(1 + [-1,0])(4,4)
       = 1[0,1](4,4)

See graph: γ(m₆ ⊖ v) = TreeSet(Vector(0, 0), Vector(4, 4))
```

Example 29. To illustrate join of marches consider $m_1 = \vec{s}_1 + k_1 I_1 \vec{x} = (2,3) + 1[0,1](0,-2)$ and $m_2 = \vec{s}_2 + k_2 I_2 \vec{y} = (2,-2) + 2[1,2](0,1)$ as per examples 25 and 26, whose points are shown in black and blue respectively in the top left graph of figure 17. Since $I_1 \neq [1,1]$ then $\vec{v} = \vec{s}_2 + k_2 \vec{y} = (2,-2) + 2(0,1) = (2,0)$. Observe $m_1 \ominus \vec{v} = 1[1,3](0,1)$ from example 25 and, using similar working, $m_2 \ominus \vec{v} = ((0,-2) + 2[1,2](0,1))^{-1} = 1[0,1](0,2)$. Thus, using the range join working of example 22, $m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v} = r_6 \sqcup r_7 = 1[0,3](0,1)$. Hence $m_1 \sqcup m_2 = \vec{v} + (m_1 \ominus \vec{v} \sqcup m_2 \ominus \vec{v}) = (2,0) + 1[0,3](0,1)$. The join is shown in red in figure 17.

```
def example29: Unit = {
  println("EXAMPLE 29")

  println(s"Consider $m1 = ${m1.m} and $m2 = ${m2.m}.")
  println(s"See graph: γ($m1) = ${`γ`(m1)}")
  println(s"See graph: γ($m2) = ${`γ`(m2)}")
  val v = m2.s + m2.k * m2.v
  val vecv = vec("v")
  println(s"Since I₁ ≠ [1,1], then $vecv = s₂ + k₂y = ${m2.s} + ${m2.k}*${m2.v} = $v.")

  println(s"Observe $m1 ⊖ $vecv = ${m1 `⊖` v} and $m2 ⊖ $vecv = ${m2 `⊖` (v, "step2")}
↪  ${m2 `⊖` v}.")
```

```
  println(s"Thus, $m1 ⊖ $vecv ⊔ $m2 ⊖ $vecv = ${(m1 ⊖ v) ⊔ (m2 ⊖ v)}.")
  println(s"Hence $m1 ⊔ $m2 = $vecv + ($m1 ⊖ $vecv ⊔ $m2 ⊖ $vecv) = ${v + ((m1 ⊖ v) ⊔
↪  (m2 ⊖ v))}")
}
```

```
EXAMPLE 29
Consider m₁ = (2,3) + 1[0,1](0,-2) and m₂ = (2,-2) + 2[1,2](0,1).
See graph: γ(m₁) = TreeSet(Vector(2, 1), Vector(2, 3))
See graph: γ(m₂) = TreeSet(Vector(2, 0), Vector(2, 2))
Since I₁ ≠ [1,1], then v = s₂ + k₂y = Vector(2, -2) + 2*Vector(0, 1) = Vector(2, 0).
Observe m₁ ⊖ v = 1[1,3](0,1) and m₂ ⊖ v = ((0,-2) + 2[1,2](0,1))⁻¹ 1[0,1](0,2).
Thus, m₁ ⊖ v ⊔ m₂ ⊖ v = 1[0,3](0,1).
Hence m₁ ⊔ m₂ = v + (m₁ ⊖ v ⊔ m₂ ⊖ v) = (2,0) + 1[0,3](0,1)
```

Example 30. Consider m_3 and m_4 as per examples 27 and 28, whose points are shown in black and blue respectively in the top right graph of figure 17. Then $\vec{v} = \vec{s_4} + k_4\vec{n} = (2,0) + 0(1,1) = (2,0)$. As shown in the aforementioned examples, $m_3 \ominus \vec{v} = 1[-1,1](1,-2)$ and $m_4 \ominus \vec{v} = 0[1,1]\vec{1}$. But $\gamma(m_4 \ominus \vec{v}) = \gamma(0[1,1]\vec{1}) = \{(0,0)\} \subseteq \{(-1,2),(0,0),(1,-2)\} = \gamma(1[-1,1](1,-2)) = \gamma(m_3 \ominus \vec{v})$. Thus $m_3 \ominus \vec{v} \sqcup m_4 \ominus \vec{v} = m_3 \ominus \vec{v} = 1[-1,1](1,-2)$. Hence $m_3 \sqcup m_4 = \vec{v} + (m_3 \ominus \vec{v} \sqcup m_4 \ominus \vec{v}) = (2,0) + 1[-1,1](1,-2)$, as illustrated by the red dots.

```
def example30: Unit = {
  println("EXAMPLE 30")
  val (v, vecv) = (m4.s + m4.k * m4.v, vec("v"))
  println(s"Consider $m3 and $m4 from previous examples. Then $vecv = ${m4.s} +
↪  ${m4.k}*${m4.v} = $v.")
  println(s"See graph: γ(m3) = ${`γ`(m3)}")
  println(s"See graph: γ(m4) = ${`γ`(m4)}")

  print(s"As shown in the aforementioned examples, $m3 ⊖ $vecv = ${m3 ⊖ v}")
  println(s" and $m4 ⊖ $vecv = ${m4 ⊖ v}.")

  println(s"But γ($m4 ⊖ $vecv) = γ(${m4 ⊖ v}) = ${`γ`(m4 ⊖ v)} ⊆ ${`γ`(m3 ⊖ v)} =
↪  γ(${m3 ⊖ v}) = γ($m3 ⊖ $vecv)}")
  assert(`γ`(m4 ⊖ v) `⊆` `γ`(m4 ⊖ v))
  println(s"Thus $m3 ⊖ $vecv ⊔ $m4 ⊖ $vecv = $m3 ⊖ $vecv = ${m3 ⊖ v}")
  println(s"Hence $m3 ⊔ $m4 = $vecv + ($m3 ⊖ $vecv ⊔ $m4 ⊖ $vecv) = ${m3 ⊔ m4}")
  println(s"See graph: γ($m3 ⊔ $m4) = ${`γ`(m3 ⊔ m4)}")
}
```

```
EXAMPLE 30
Consider m₃ and m₄ from previous examples. Then v = Vector(2, 0) + 0*Vector(1, 1) =
↪  Vector(2, 0).
See graph: γ(m₃) = TreeSet(Vector(1, 2), Vector(2, 0), Vector(3, -2))
```

```
See graph: γ(m₄) = TreeSet(Vector(2, 0))
As shown in the aforementioned examples, m₃ ⊖ v = 1[-1,1](1,-2) and m₄ ⊖ v =
↪  0[1,1](1,1).
But γ(m₄ ⊖ v) = γ(0[1,1](1,1)) = TreeSet(Vector(0, 0)) ⊆ TreeSet(Vector(-1, 2),
↪  Vector(0, 0), Vector(1, -2)) = γ(1[-1,1](1,-2)) = γ(m₃ ⊖ v)}
Thus m₃ ⊖ v ⊔ m₄ ⊖ v = m₃ ⊖ v = 1[-1,1](1,-2)
Hence m₃ ⊔ m₄ = v + (m₃ ⊖ v ⊔ m₄ ⊖ v) = (2,0) + 1[-1,1](1,-2)
See graph: γ(m₃ ⊔ m₄) = TreeSet(Vector(1, 2), Vector(2, 0), Vector(3, -2))
```

Example 31. Consider $m_5 = (-1,-2)+1[0,2](2,2)$ and $m_6 = (4,3)+1[0,1](-4,-4)$ from example 29, whose points are shown in black and blue respectively in the lower left graph of figure 17. Then, as shown in example 29, $m_5 \ominus \vec{v} = 1[-1,3](1,1)$ and $m_6 \ominus \vec{v} = 1[0,1](4,4)$ where $\vec{v} = (0,-1)$. Using a similar method to that shown in, say, example 22, it follows $m_5 \ominus \vec{v} \sqcup m_6 \ominus \vec{v} = 1[-1,4](1,1)$. Hence $m_5 \sqcup m_6 = \vec{v} + (m_5 \ominus \vec{v} \sqcup m_6 \ominus \vec{v}) = (0,-1) + 1[-1,4](1,1)$.

```
def example31: Unit = {
  println("EXAMPLE 31")
  val (v, vecv) = (m6.s + m6.k * m6.v, vec("v"))

  println(s"Consider $m5 = ${m5.m} and $m6 = ${m6.m}")
  println(s"See graph: γ($m5) = ${`γ`(m5)}")
  println(s"See graph: γ($m6) = ${`γ`(m6)}")
  println(s"Then $m5 ⊖ $vecv = ${m5 `⊖` v} and $m6 ⊖ $vecv = ${m6 `⊖` v} where $vecv =
↪  $v")

  println(s"It follows $m5 ⊖ $vecv ⊔ $m6 ⊖ $vecv = ${(m5 `⊖` v) ⊔ (m6 `⊖` v)}")
  println(s"Hence $m5 ⊔ $m6 = $vecv + ($m5 ⊖ $vecv ⊔ $m6 ⊖ $vecv) = ${m5 ⊔ m6}")
}
```

```
EXAMPLE 31
Consider m₅ = (-1,-2) + 1[0,2](2,2) and m₆ = (4,3) + 1[0,1](-4,-4)
See graph: γ(m₅) = TreeSet(Vector(-1, -2), Vector(1, 0), Vector(3, 2))
See graph: γ(m₆) = TreeSet(Vector(0, -1), Vector(4, 3))
Then m₅ ⊖ v = 1[-1,3](1,1) and m₆ ⊖ v = 1[0,1](4,4) where v = Vector(0, -1)
It follows m₅ ⊖ v ⊔ m₆ ⊖ v = 1[-1,4](1,1)
Hence m₅ ⊔ m₆ = v + (m₅ ⊖ v ⊔ m₆ ⊖ v) = (0,-1) + 1[-1,4](1,1)
```

Example 32. Consider $m_2 = (2,-2) + 2[1,2](0,1)$ and $m_7 = (1,1) + 2[0,1](0,1)$. Then $\vec{v} = \vec{s_7} + k_7\vec{r} = (1,1) + 2(0,1) = (1,3)$. First, $m_2 \ominus \vec{v} = \top$, as derived in example 26. Analogous to example 29, it follows $m_7 \ominus \vec{v} = 1[-1,0](0,2)$. Thus $m_2 \ominus \vec{v} \sqcup m_7 \ominus \vec{v} = \top \sqcup 1[-1,0](0,2) = \top$. Hence $m_2 \sqcup m_7 = \vec{v} + (m_2 \ominus \vec{v} \sqcup m_7 \ominus \vec{v}) = \vec{v} + \top = \top$. This is shown in the lower right graph of figure 17.

```
def example32: Unit = {
  println("EXAMPLE 32")
  val m7 = new NamedMarch(Vector(1,1), KIV(KI(2, Interval(0,1)), Vector(0,1)), 7)
  val (v, vecv) = (m7.s + m7.k * m7.v, vec("v"))

  println(s"Consider $m2 = ${m2.m} and $m7 = ${m7.m}")
  println(s"See graph: γ($m2) = ${`γ`(m2)}")
  println(s"See graph: γ($m7) = ${`γ`(m7)}")
  println(s"Then $vecv = s₇ + k₇τ = ${m7.s} + ${m7.k}*${m7.v} = $v.")
  println(s"First, $m2 ⊖ $vecv = ${m2 ⊖ v}, as derived in example 25.")
  println(s"Analogous to example 28, it follows $m7 ⊖ $vecv = ${m7 `⊖` v}")

  println(s"Thus, $m2 ⊖ $vecv ⊔ $m7 ⊖ $vecv = ${(m2 `⊖` v) ⊔ (m7 `⊖` v)}")
  println(s"Hence $m2 ⊔ $m7 = $vecv + ($m2 ⊖ $vecv ⊔ $m7 ⊖ $vecv) = $vecv + ${((m2 `⊖`
↪  v) ⊔ (m7 `⊖` v))} = ${m2 ⊔ m7}")
}
```

```
EXAMPLE 32
Consider m₂ = (2,-2) + 2[1,2](0,1) and m₇ = (1,1) + 2[0,1](0,1)
See graph: γ(m₂) = TreeSet(Vector(2, 0), Vector(2, 2))
See graph: γ(m₇) = TreeSet(Vector(1, 1), Vector(1, 3))
Then v = s₇ + k₇τ = Vector(1, 1) + 2*Vector(0, 1) = Vector(1, 3).
First, m₂ ⊖ v = τ, as derived in example 25.
Analogous to example 28, it follows m₇ ⊖ v = 1[-1,0](0,2)
Thus, m₂ ⊖ v ⊔ m₇ ⊖ v = τ
Hence m₂ ⊔ m₇ = v + (m₂ ⊖ v ⊔ m₇ ⊖ v) = v + τ = τ
```

Appendix D

Unwinding Graphs (more detail)

This section presents the unwinding graphs given in Chapter 4 but where the vertices are annotated with their identifier, formula and automaton location. This information may not be visible when printed on an A4 page but can be viewed using an electronic previewer.

Figure 68: Step 0: initialization

Figure 69: Step 1: Expansion of 0

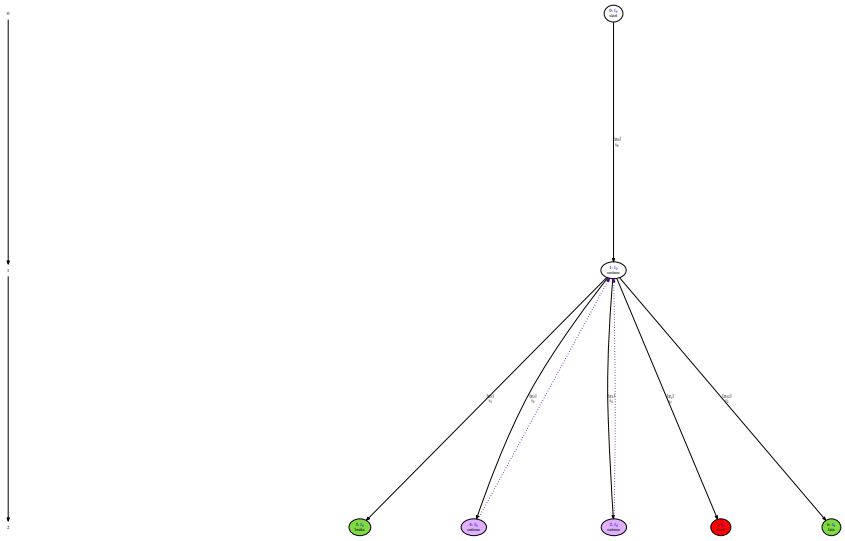


Figure 70: Step 2: Expansion of 1

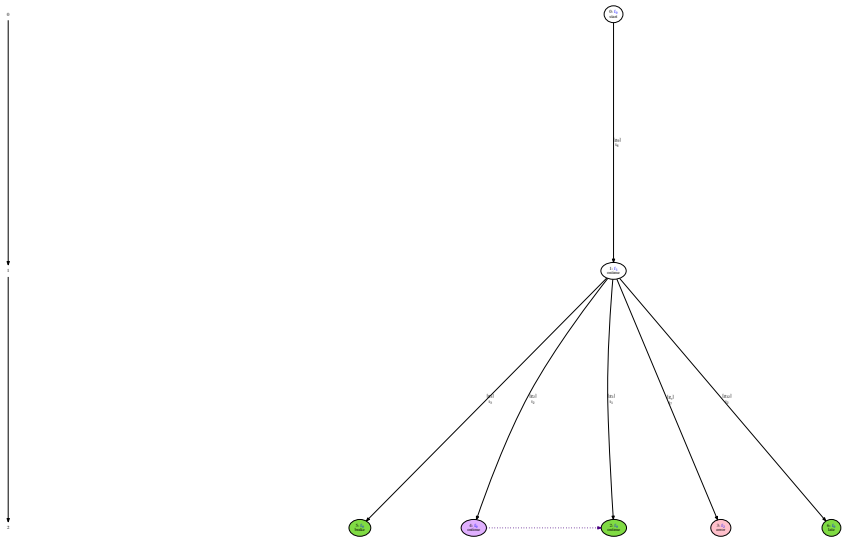


Figure 71: Step 3: Refinement of 3

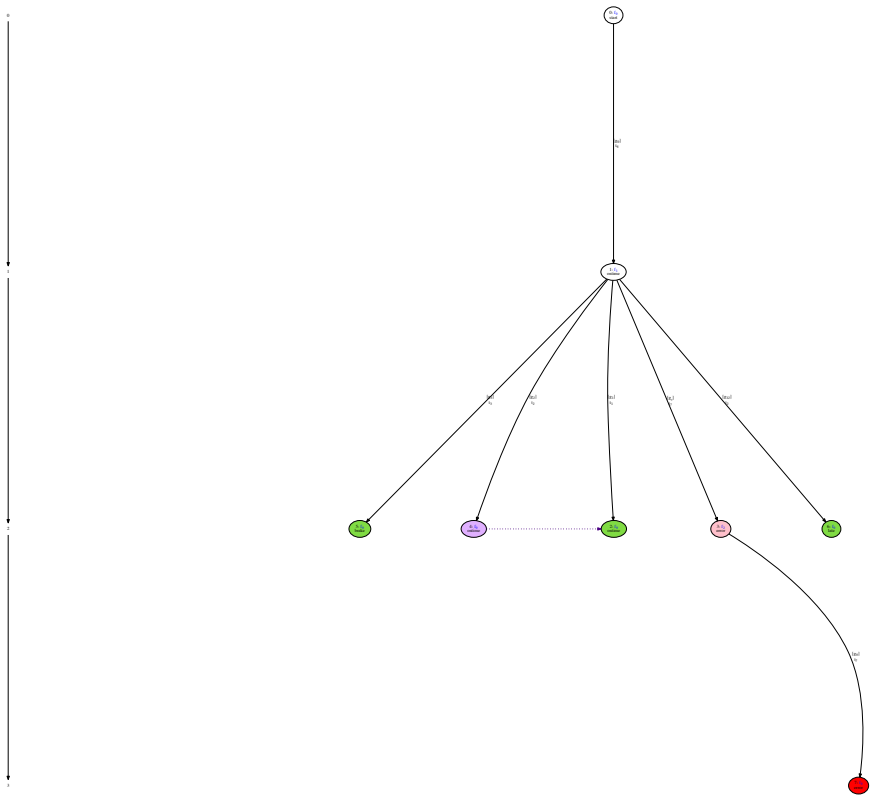


Figure 72: Step 4: Expansion of 3

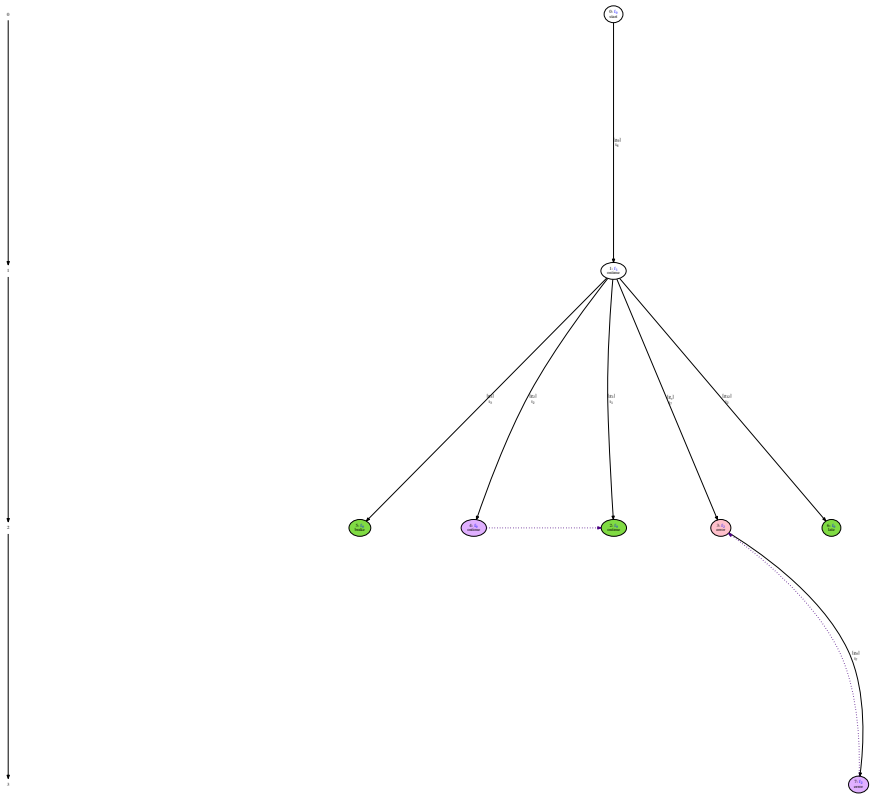


Figure 73: Step 5: Refinement of 7

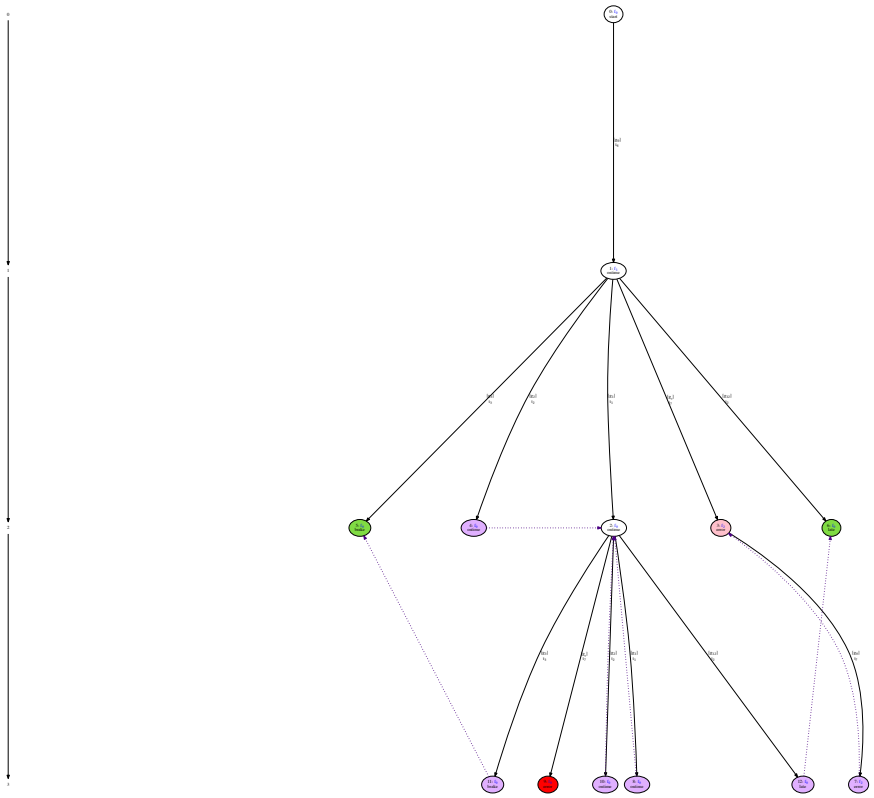


Figure 74: Step 6: Expansion of 2

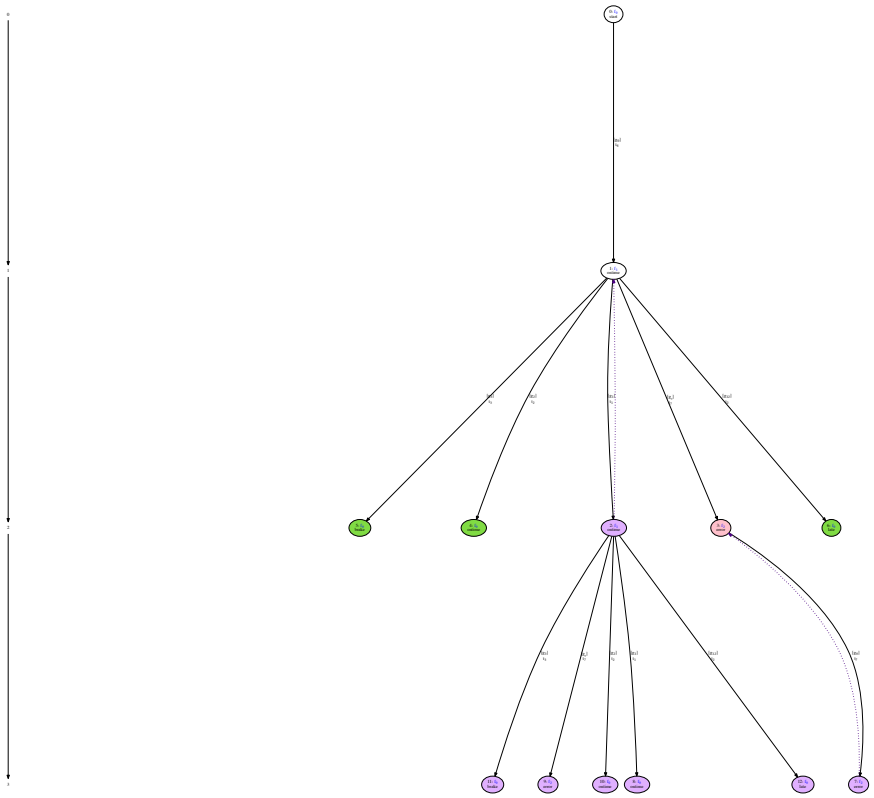


Figure 75: Step 7: Refinement of 9

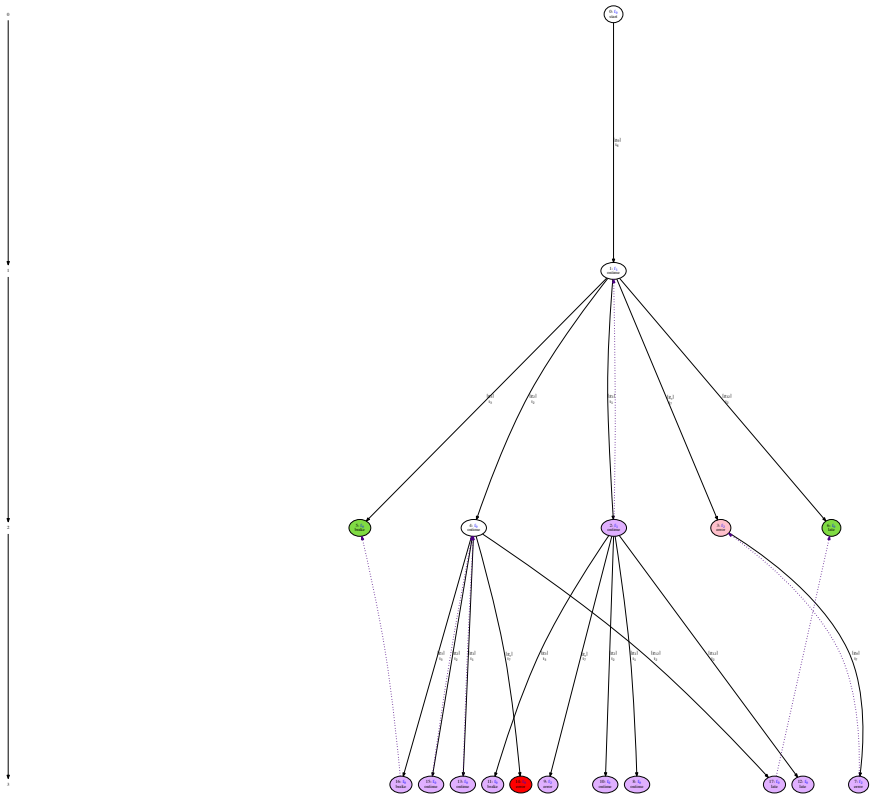


Figure 76: Step 8: Expansion of 4

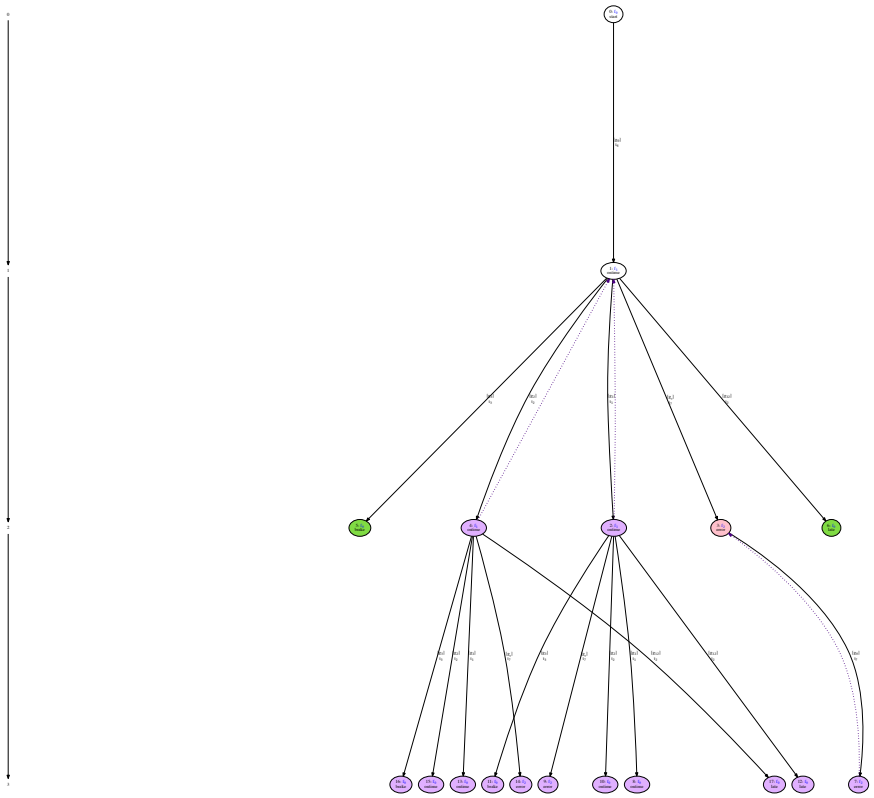


Figure 77: Step 9: Refinement of 14

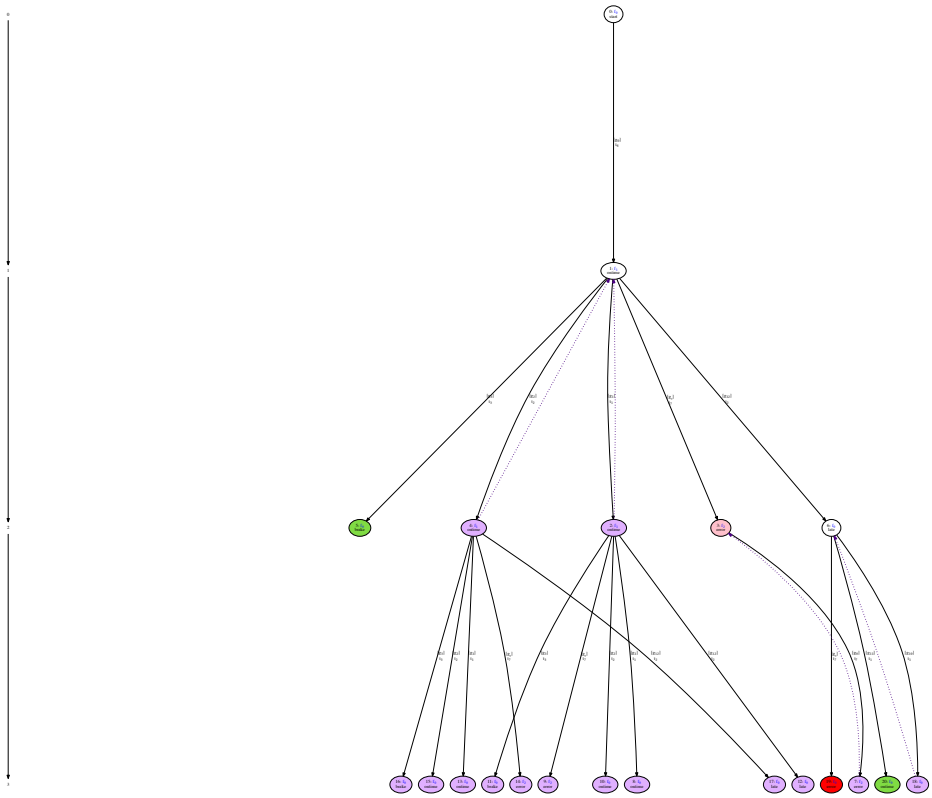


Figure 78: Step 10: Expansion of 6

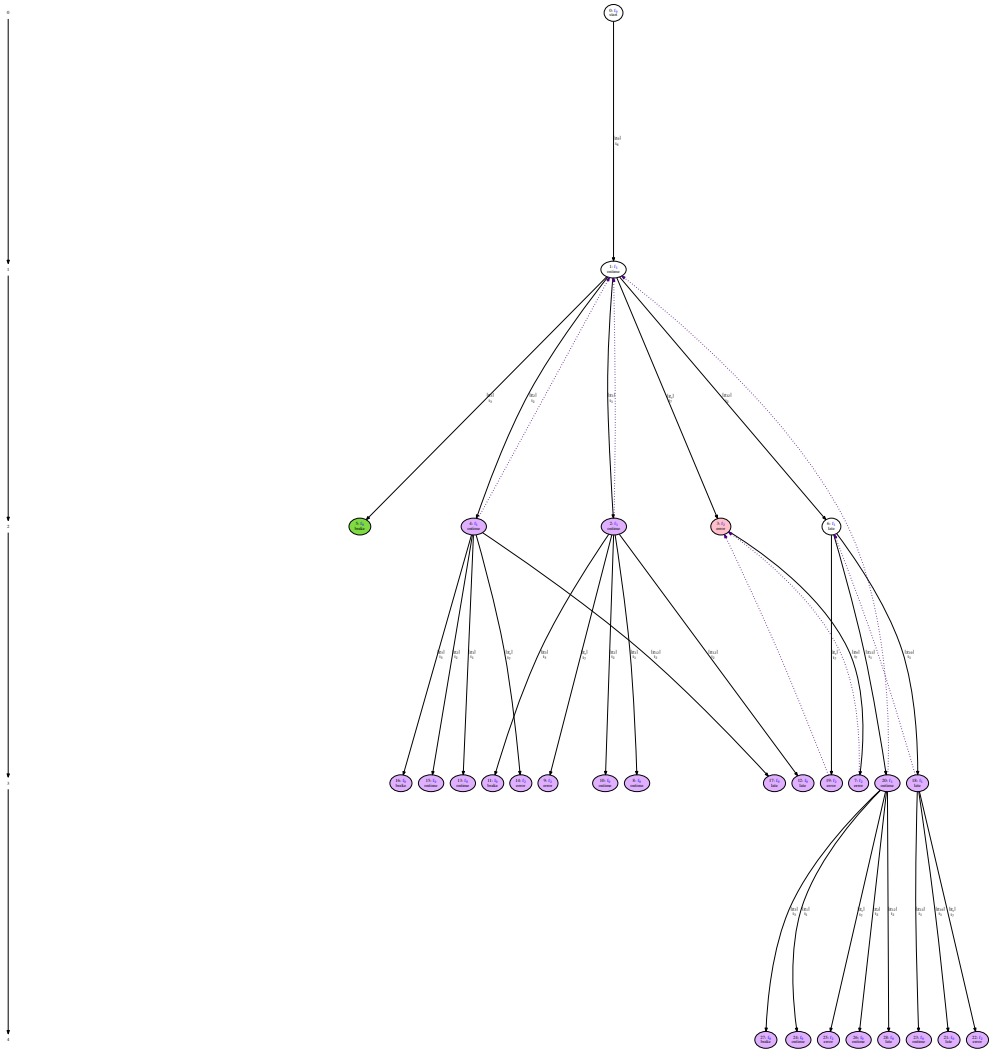


Figure 79: Step 15: Refinement of 25

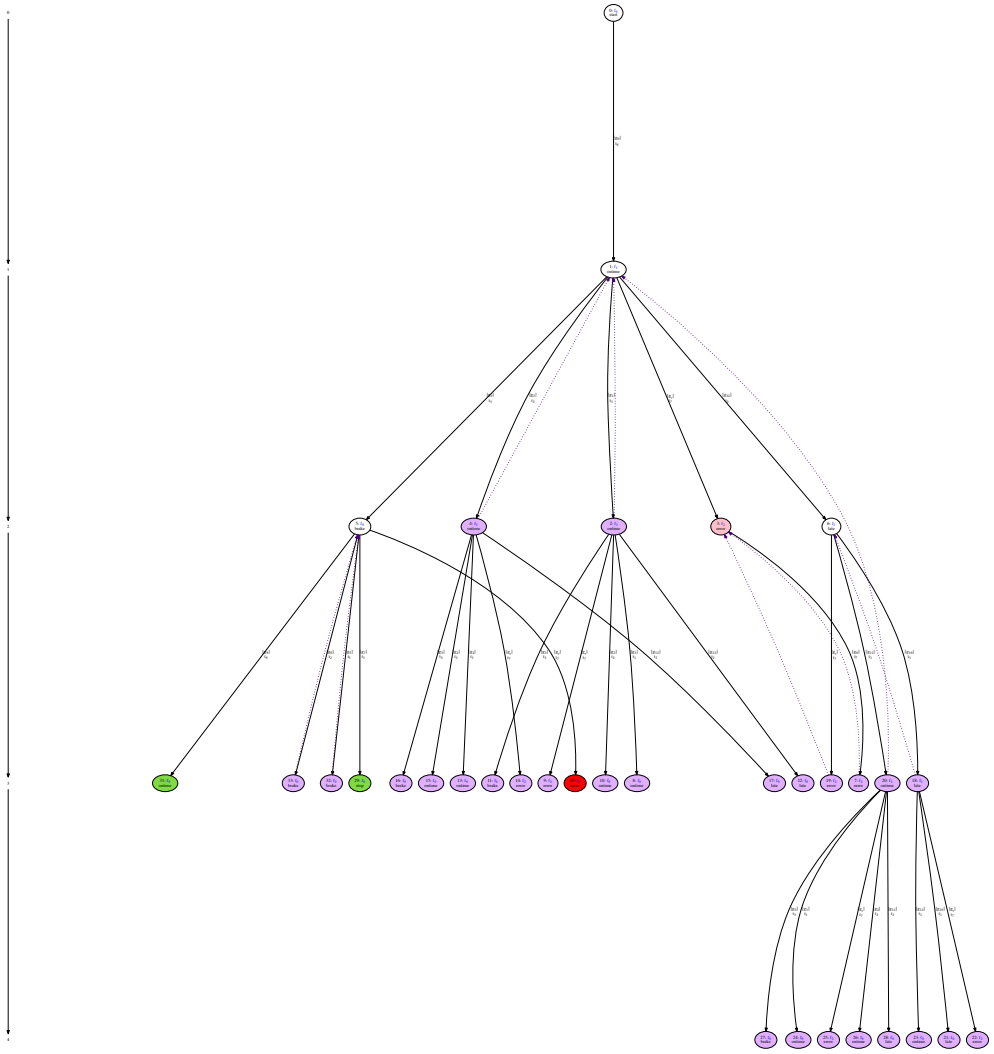


Figure 80: Step 16: Expansion of 5

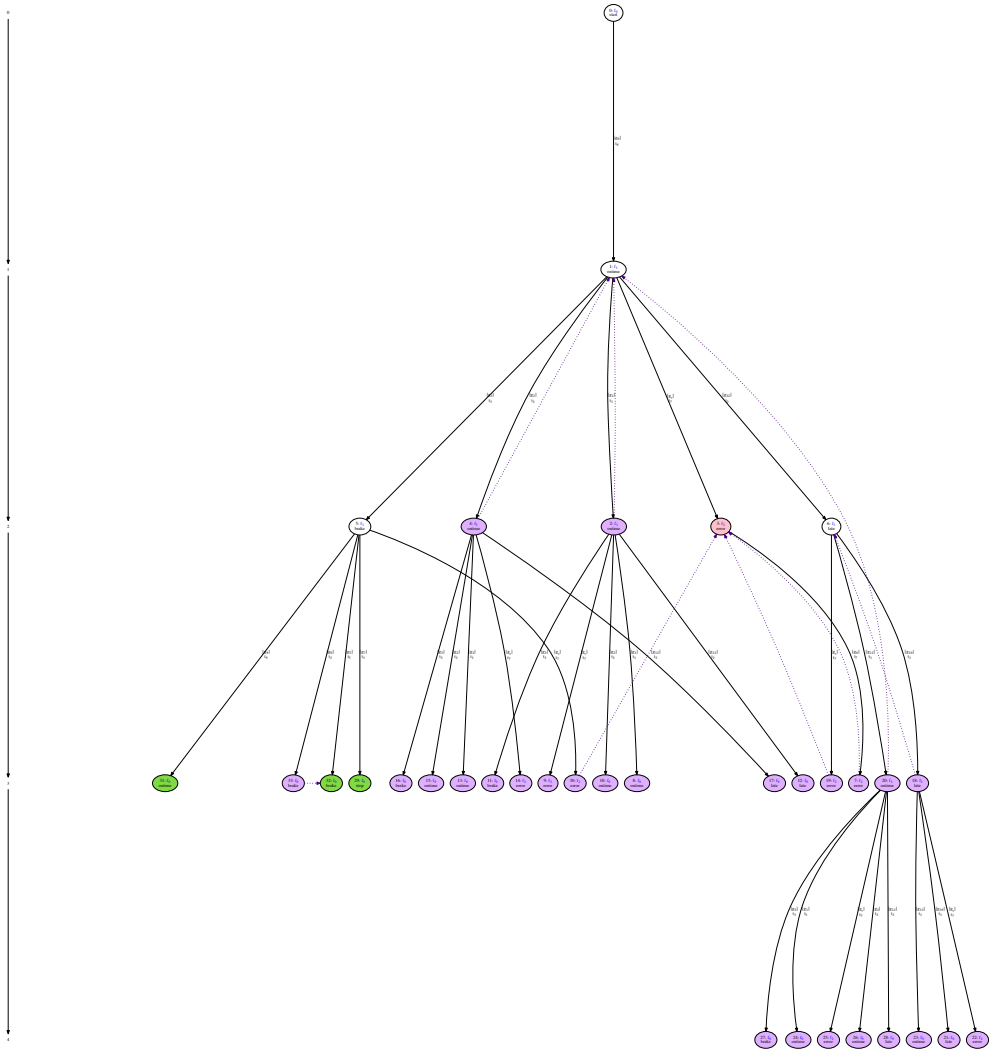


Figure 81: Step 17: Refinement of 30

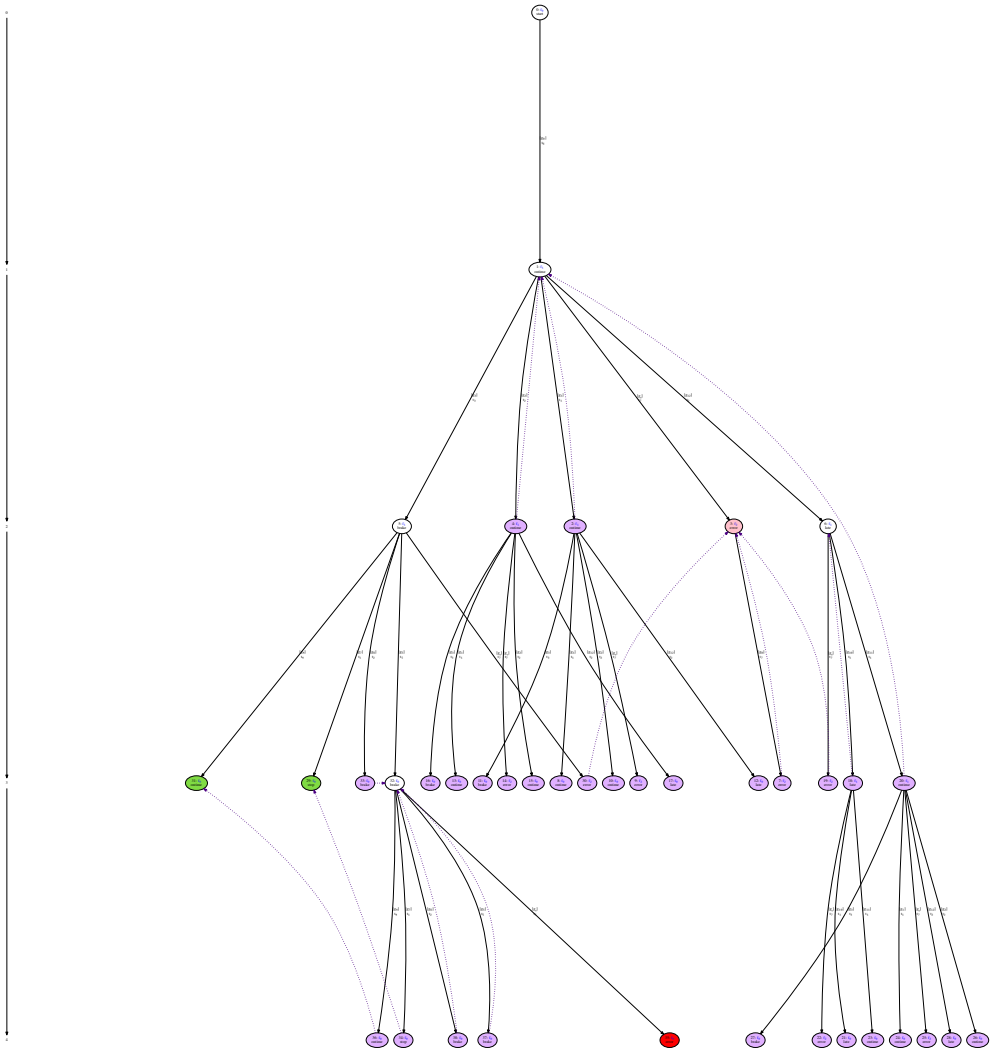


Figure 82: Step 18: Expansion of 32

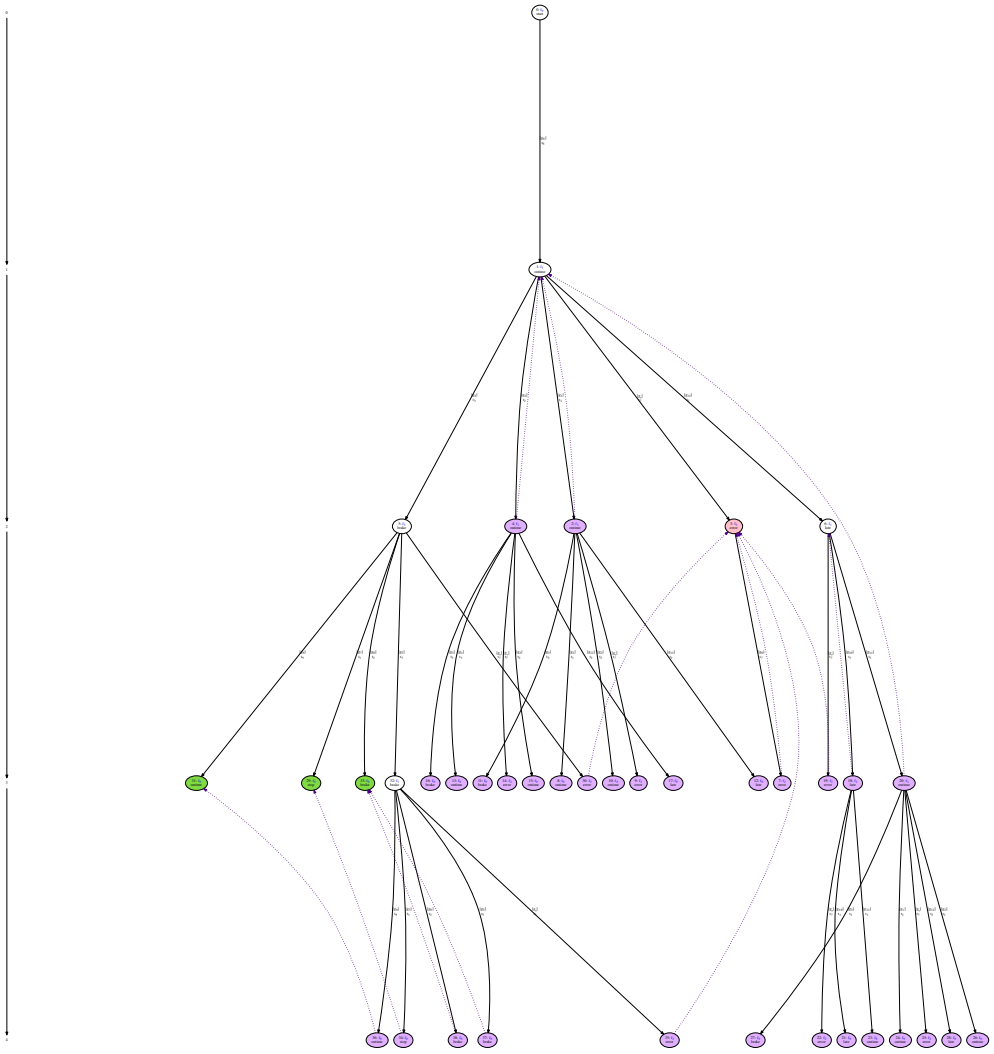


Figure 83: Step 19: Refinement of 35

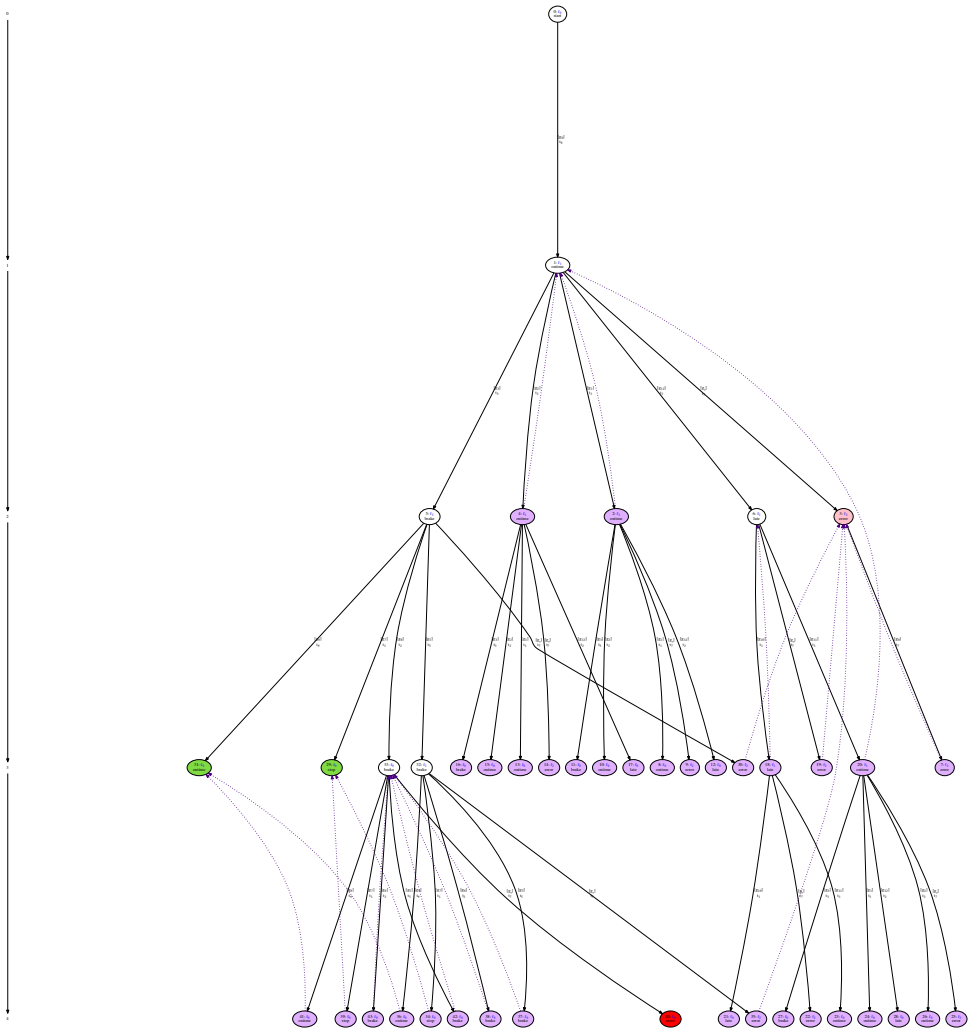


Figure 84: Step 20: Expansion of 33

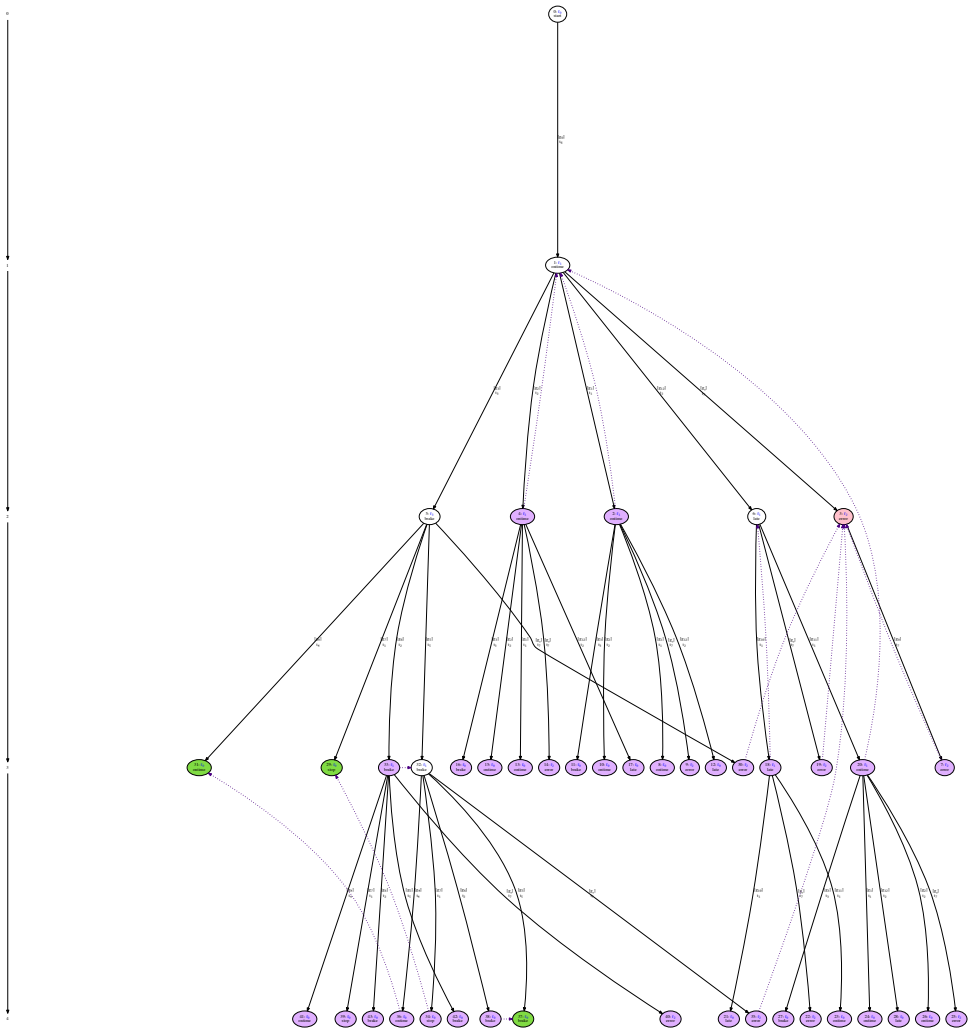


Figure 85: Step 21: Refinement of 40

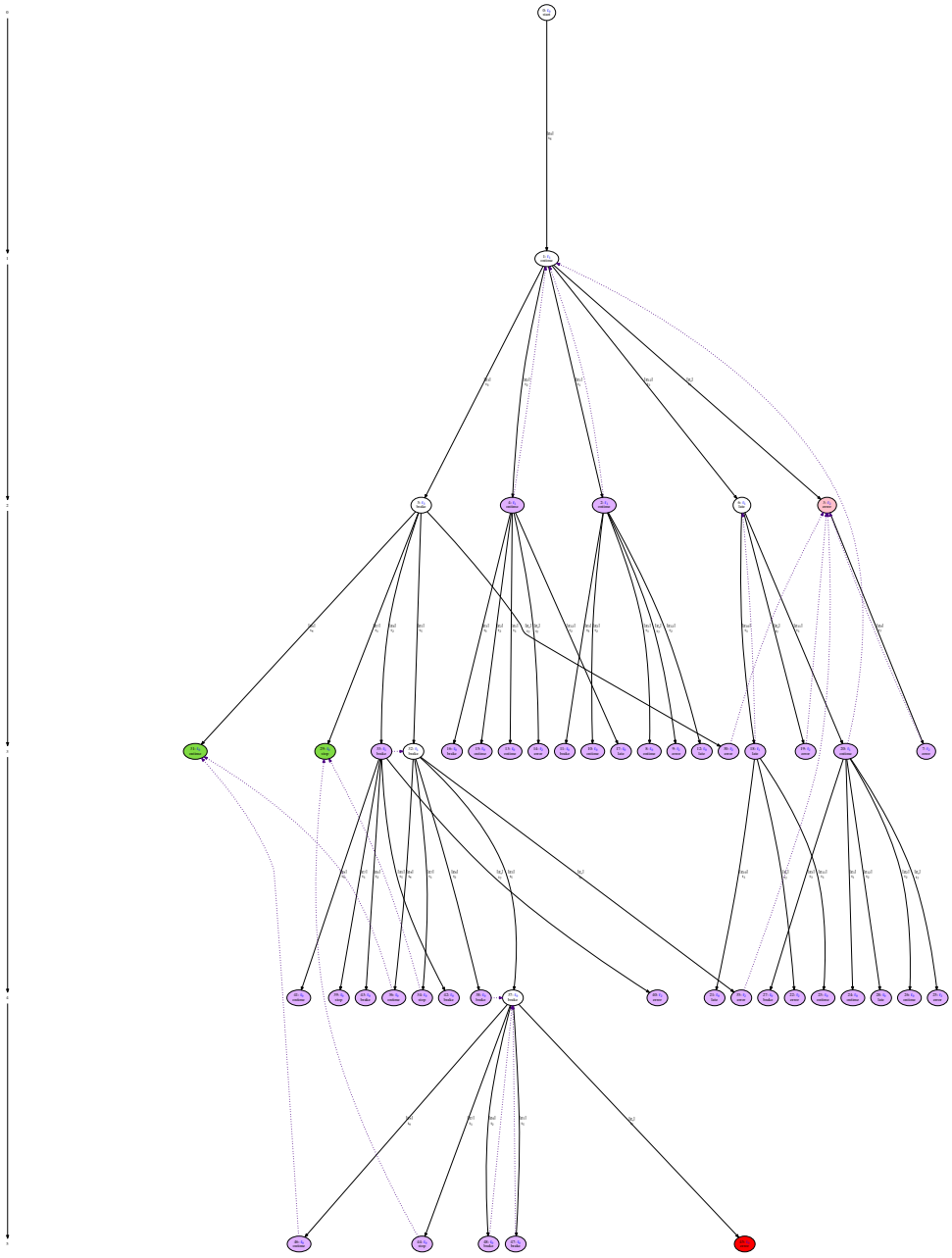


Figure 86: Step 22: Expansion of 37

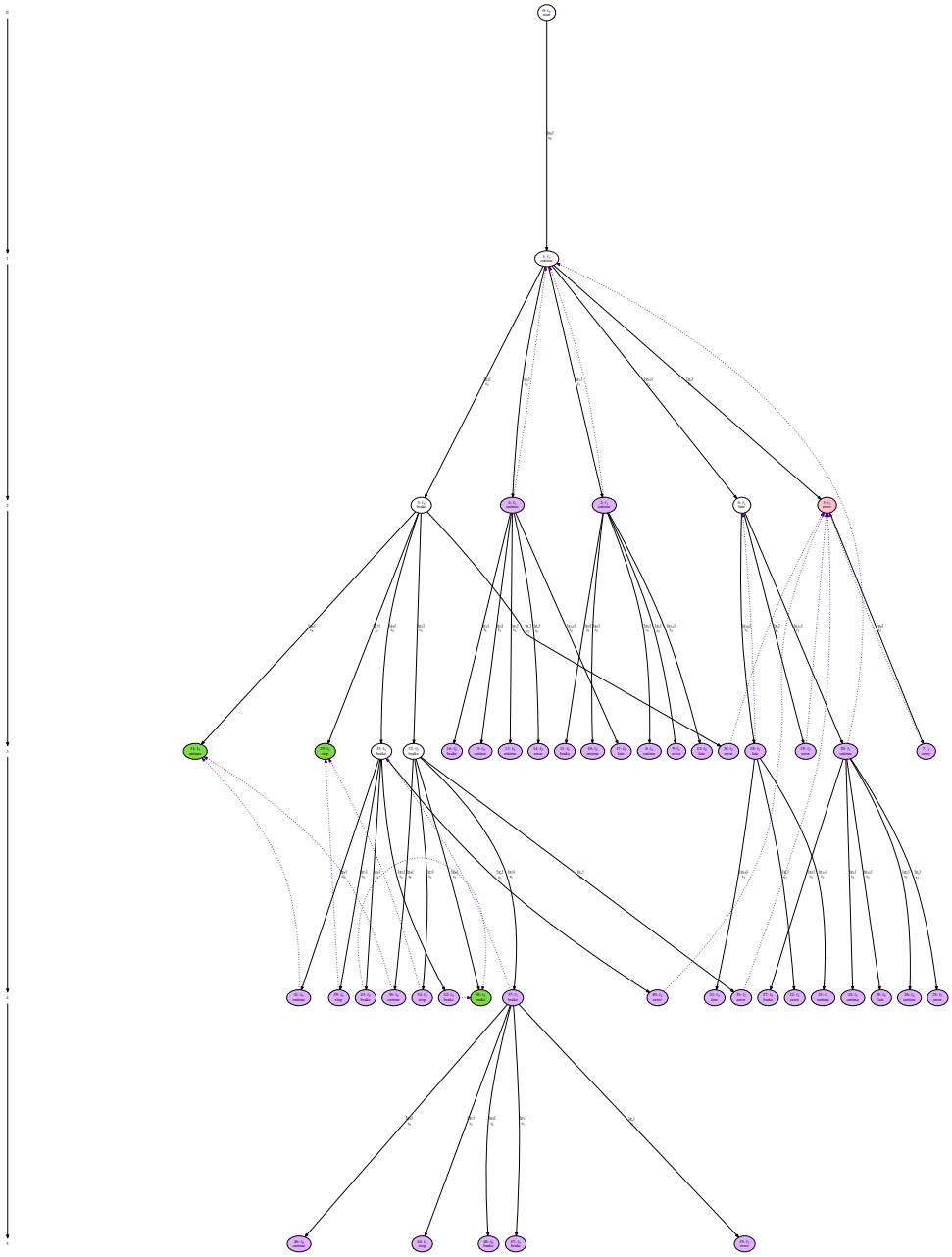


Figure 87: Step 23: Refinement of 45

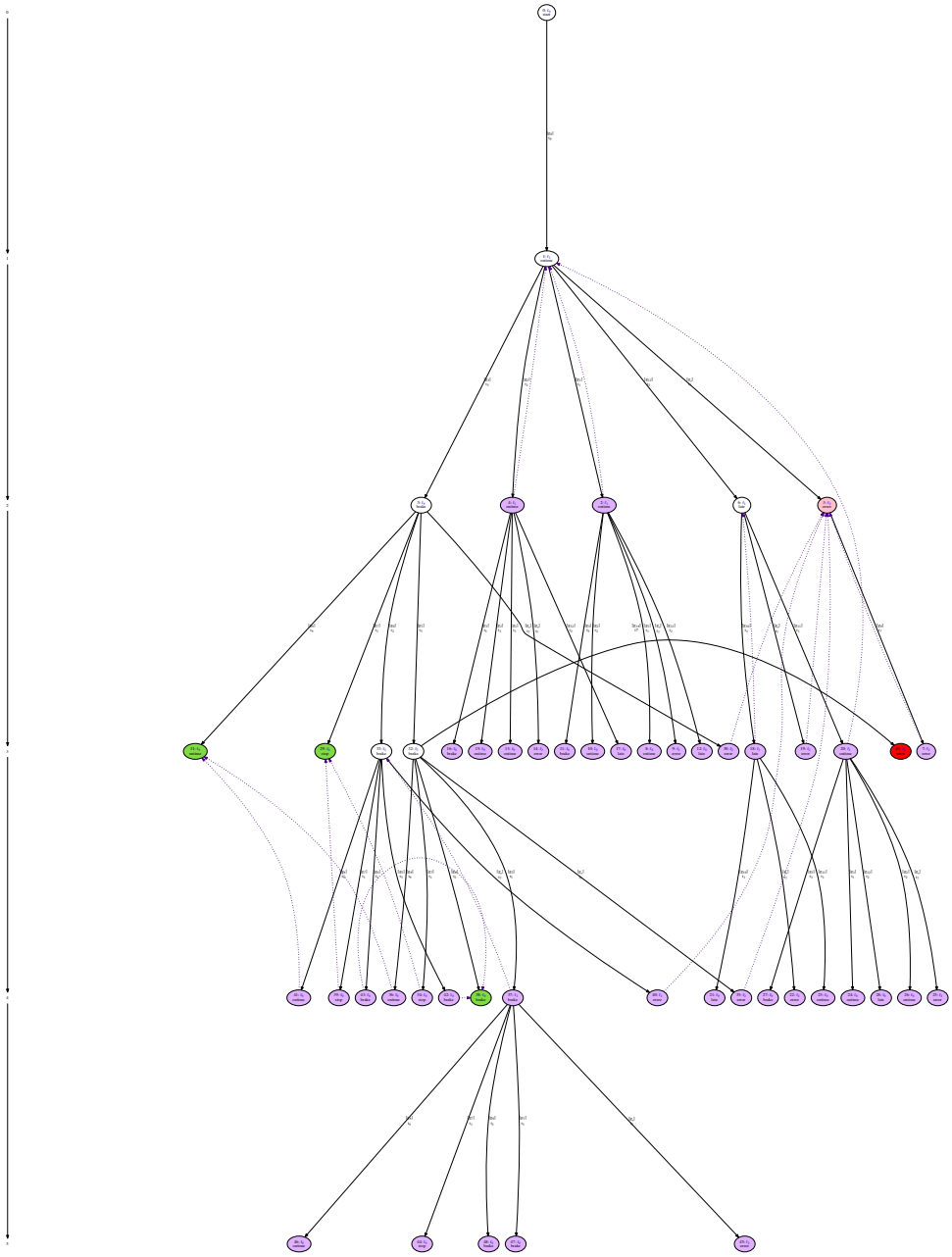


Figure 88: Step 24: Set g_1 at 32

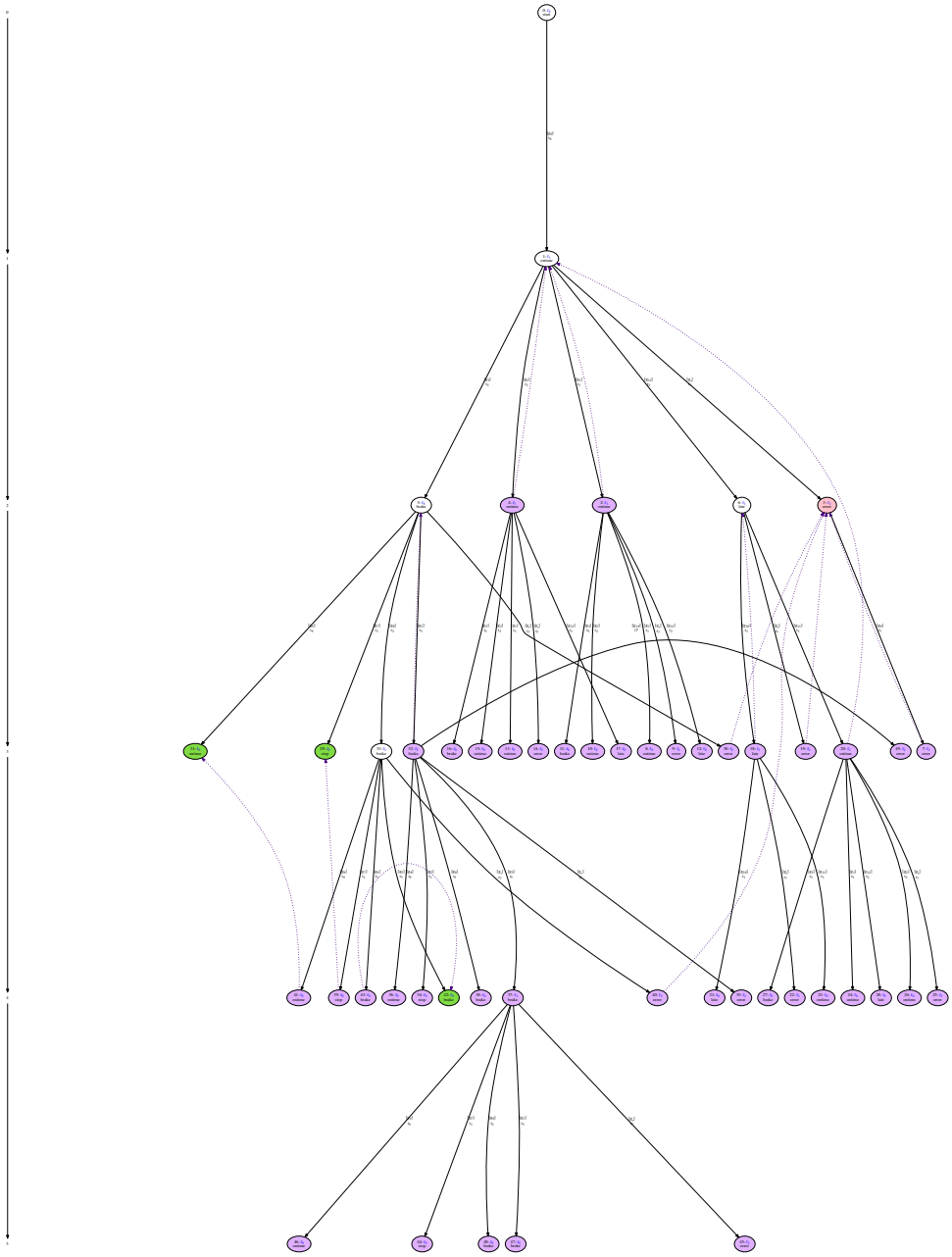


Figure 89: Step 25: Refinement of 49

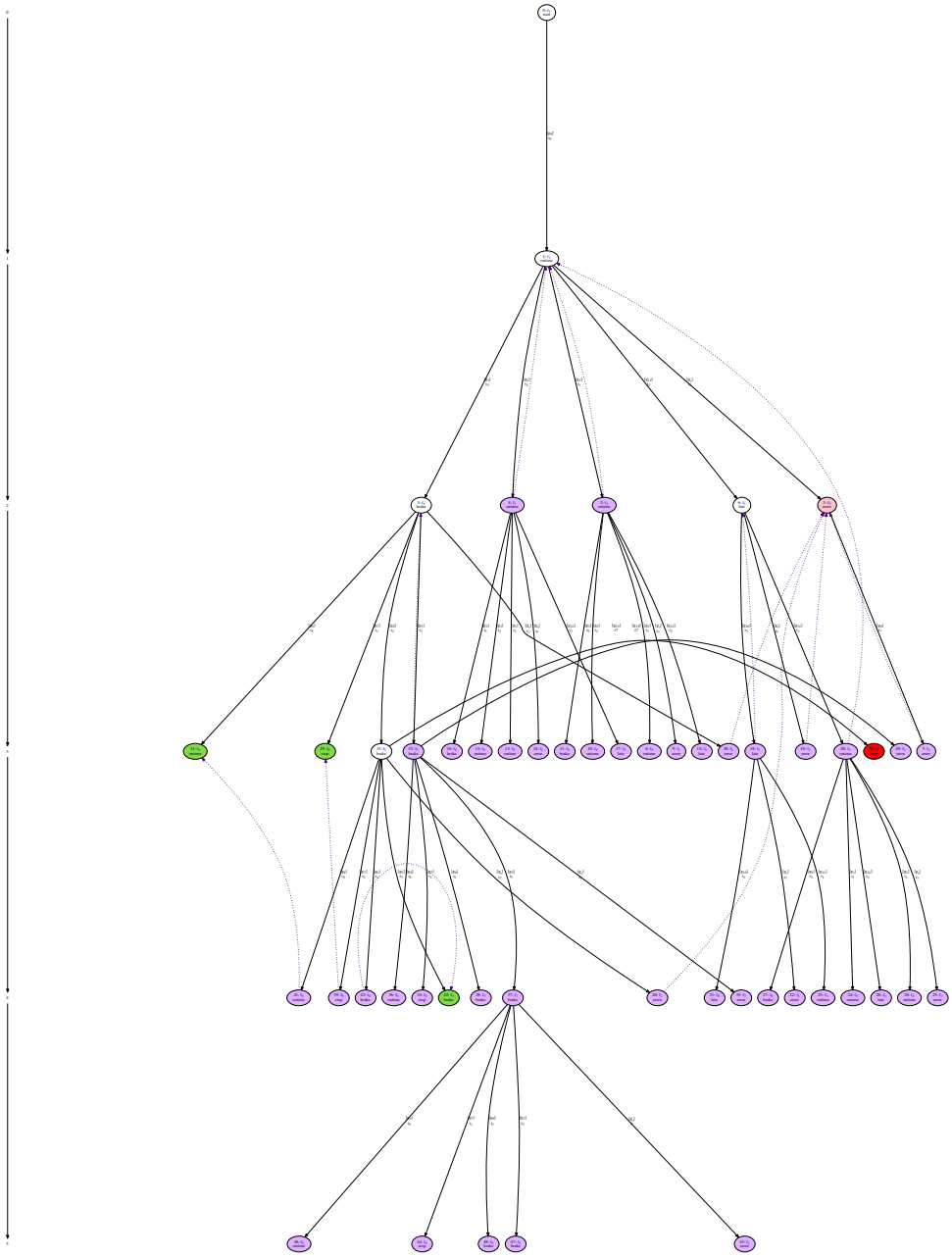


Figure 90: Step 26: Set g_1 at 33

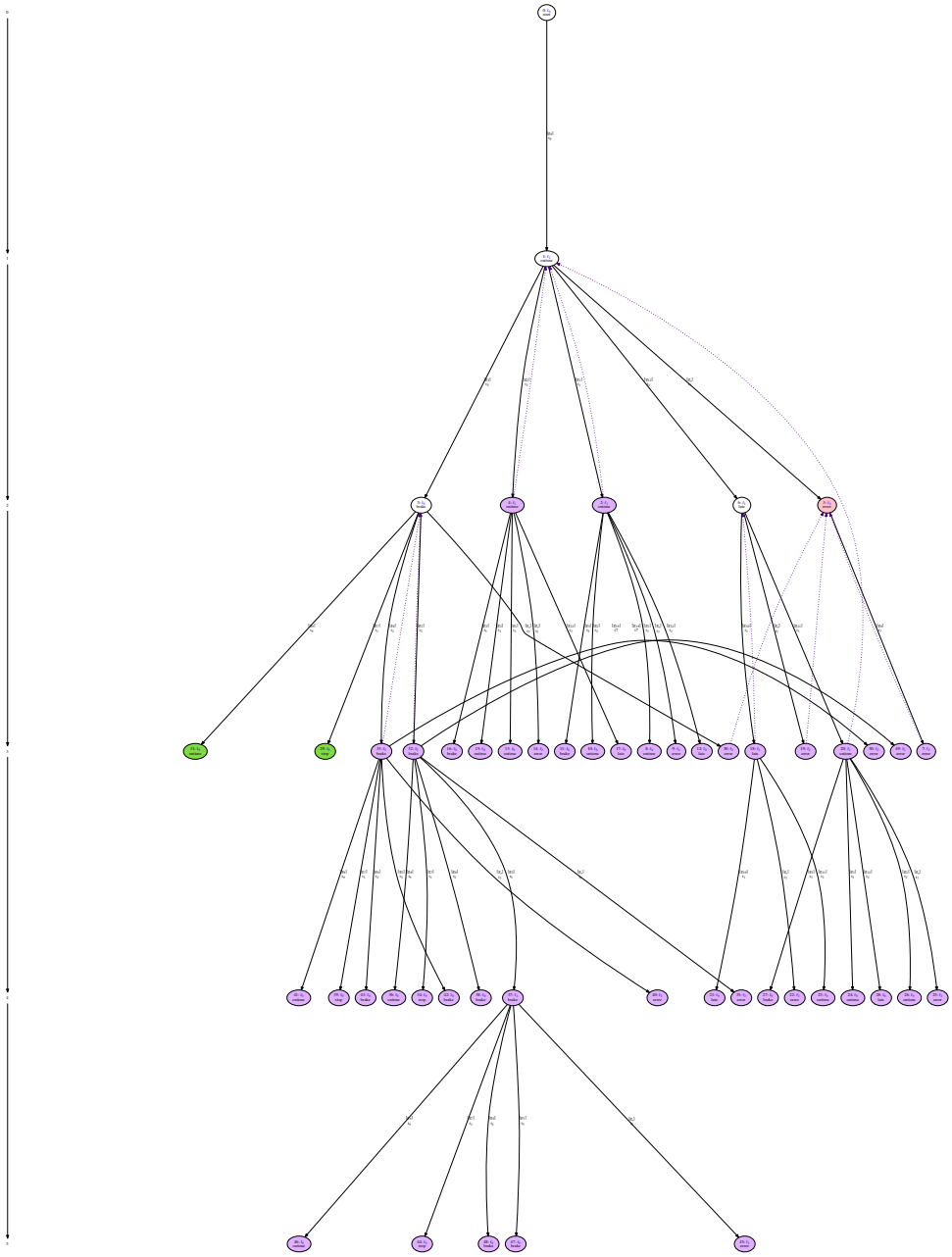


Figure 91: Step 27: Refinement of 50

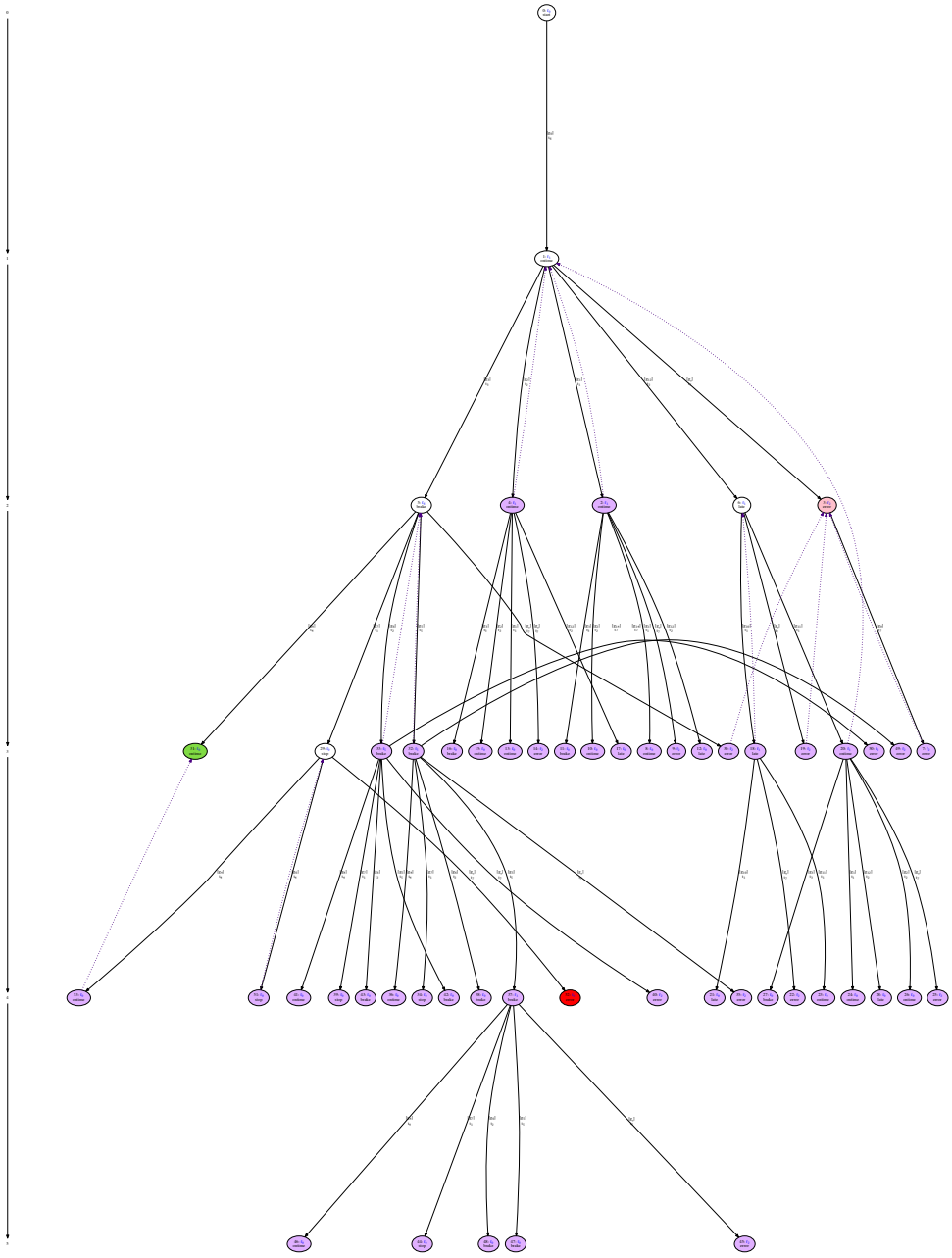


Figure 92: Step 28: Expansion of 29

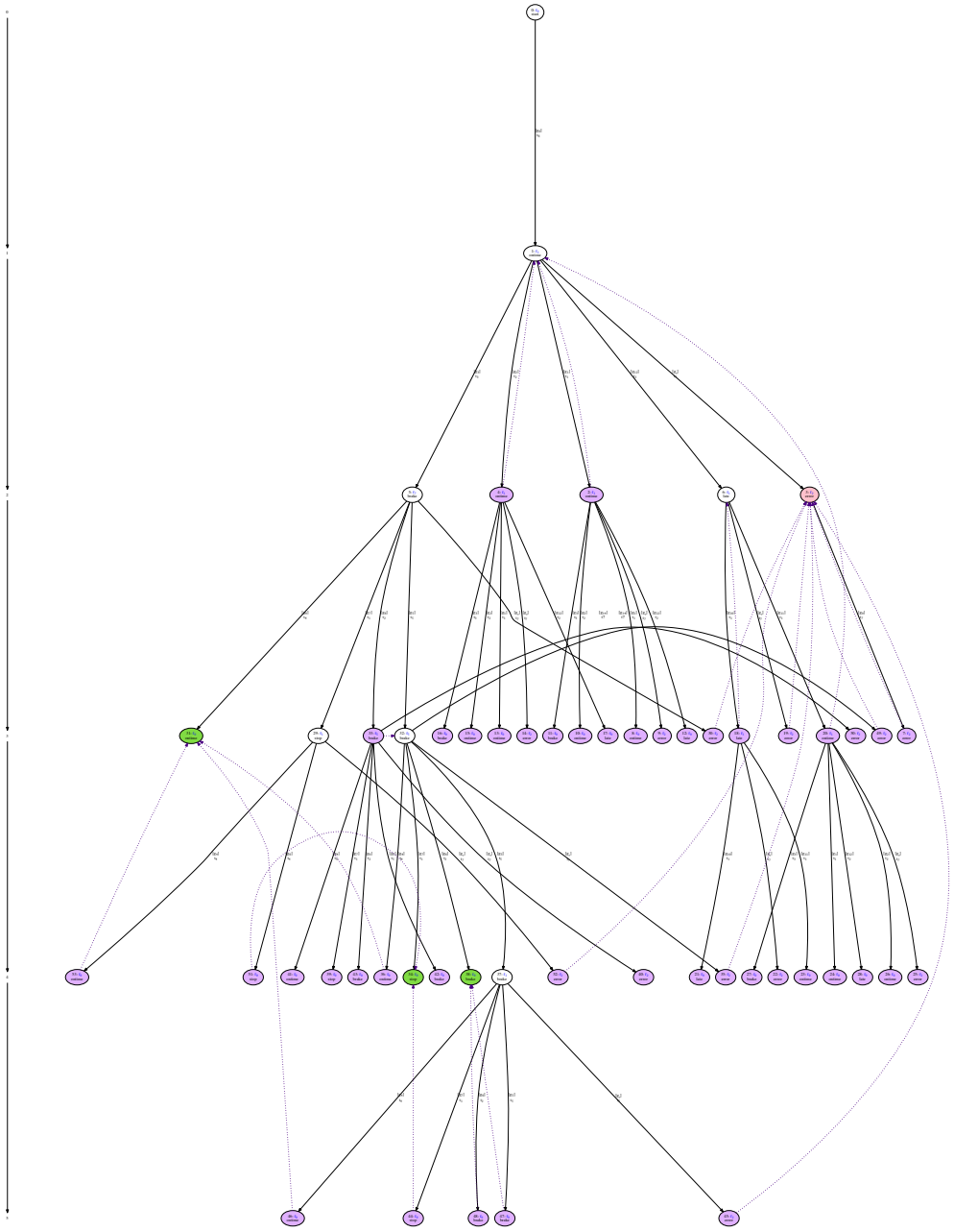


Figure 93: Step 29: Refinement of 52

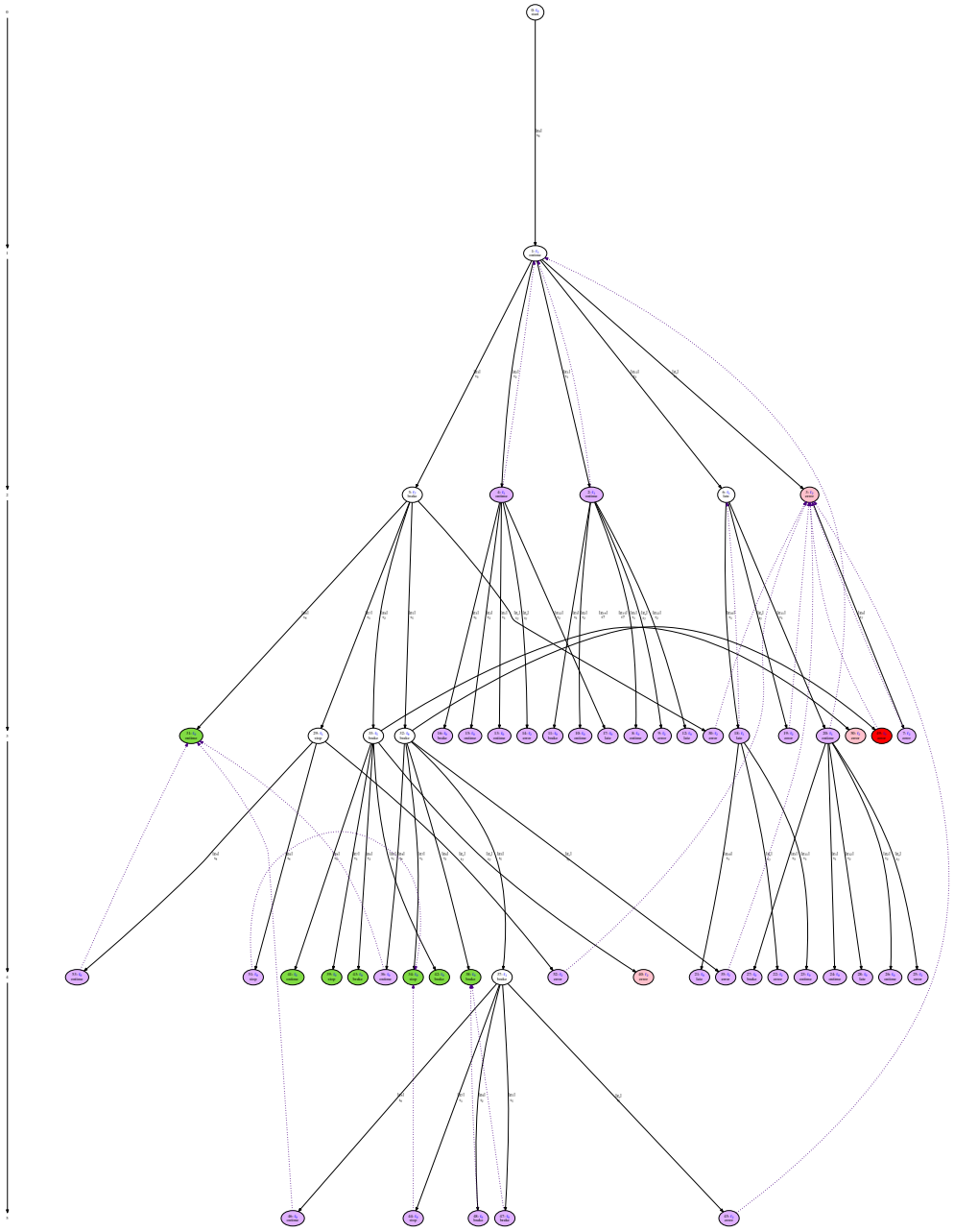


Figure 94: Step 30: Set g_2 at 30

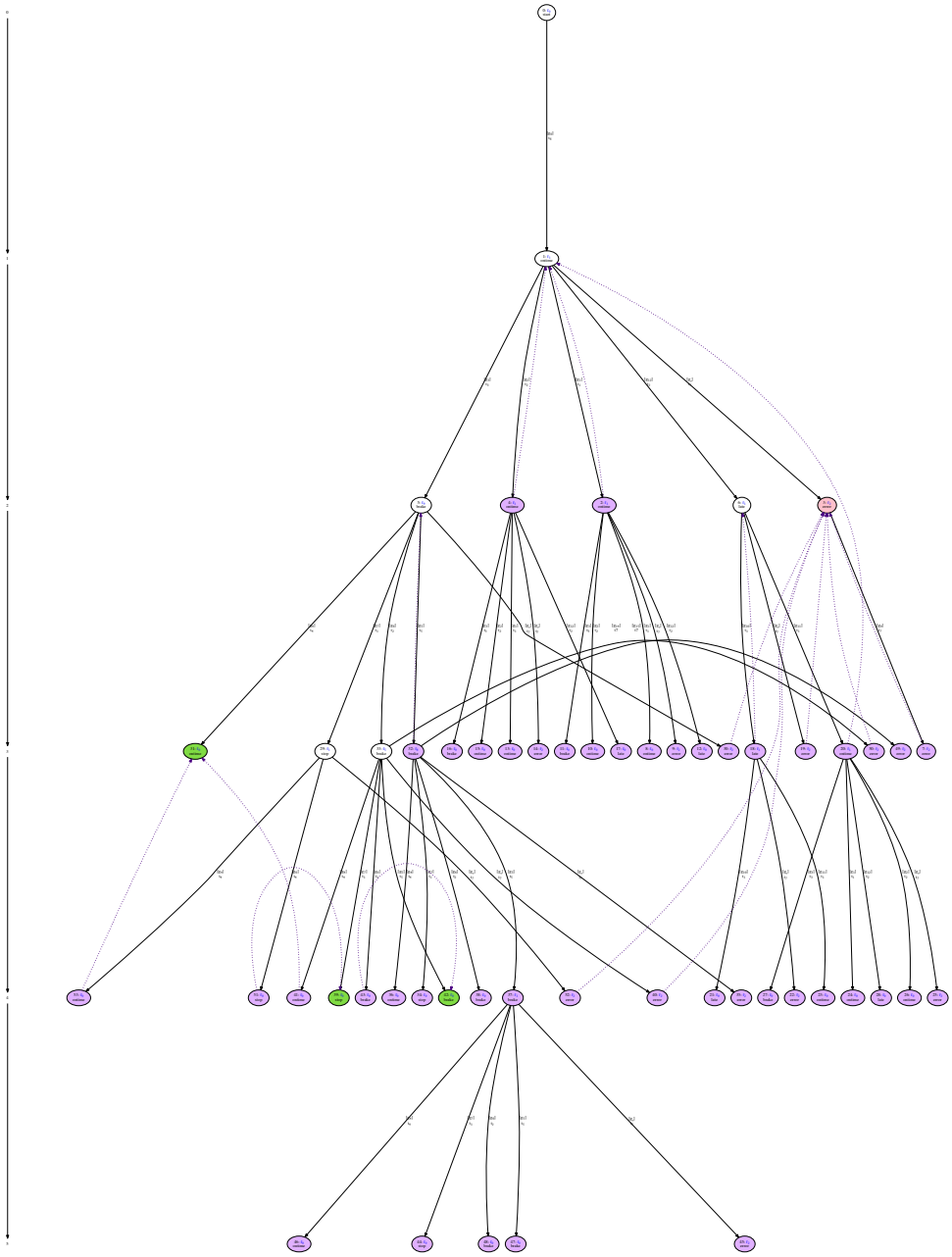


Figure 95: Step 31: Refinement of 49

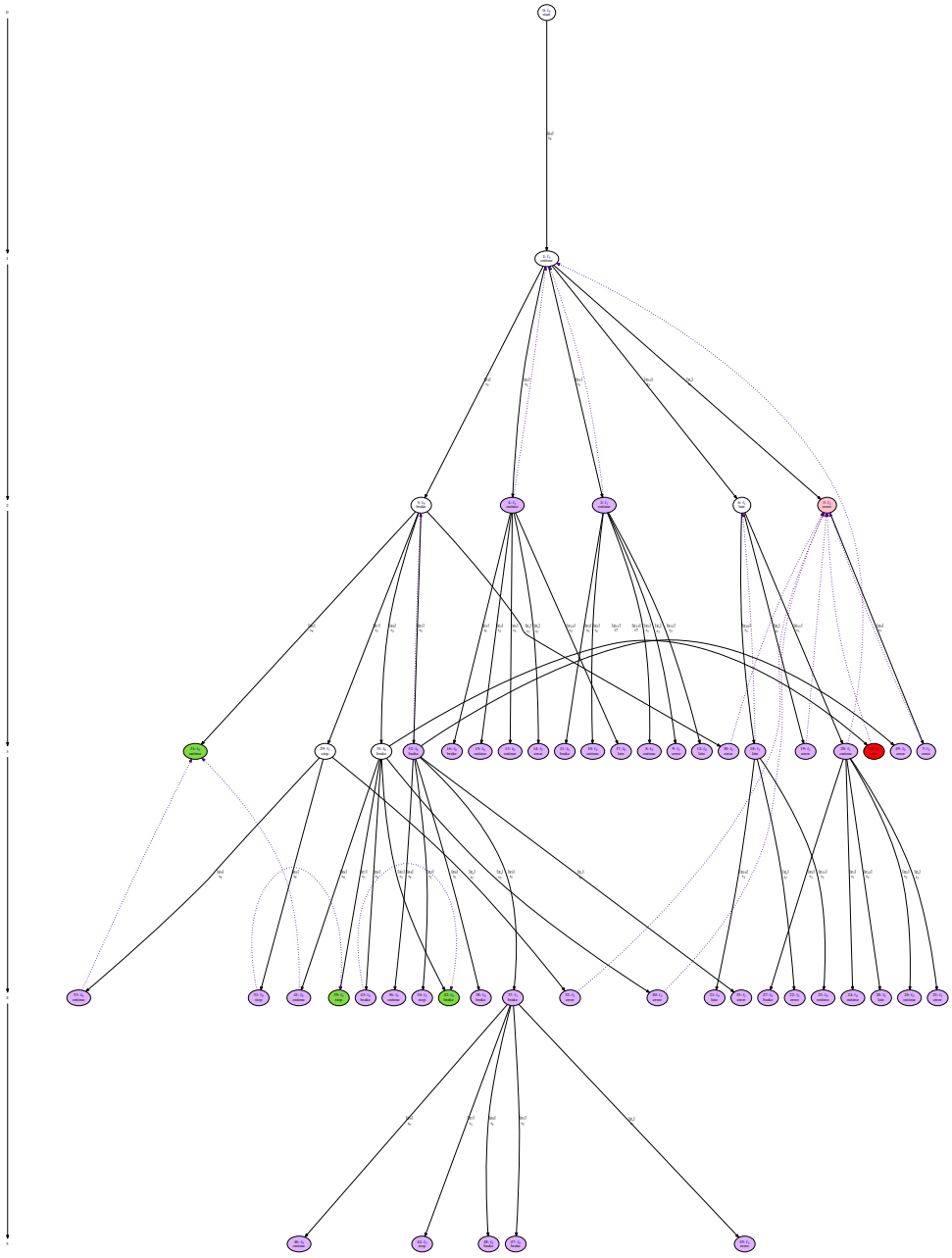


Figure 96: Step 32: Set g_2 at 32

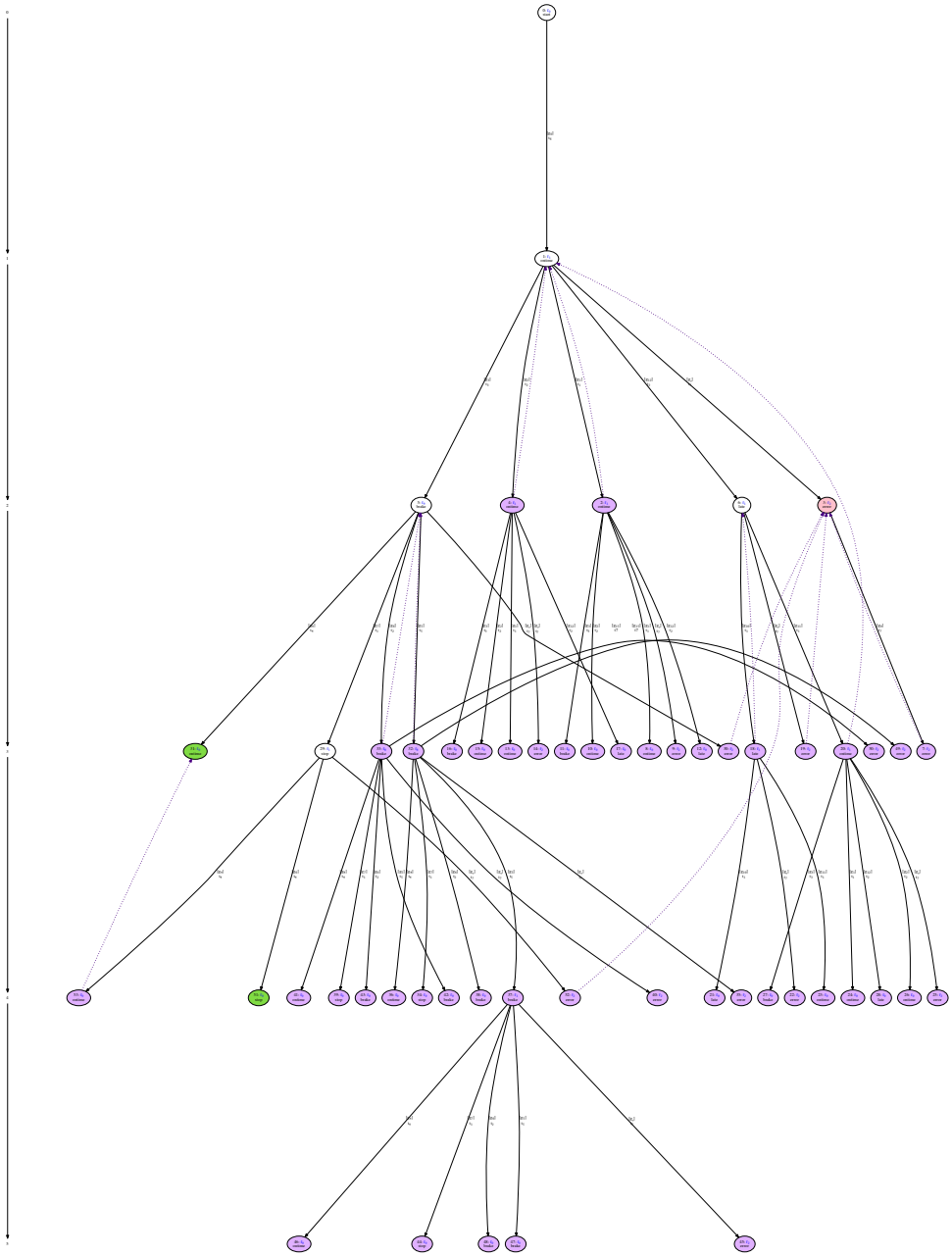


Figure 97: Step 33: Refinement of 50

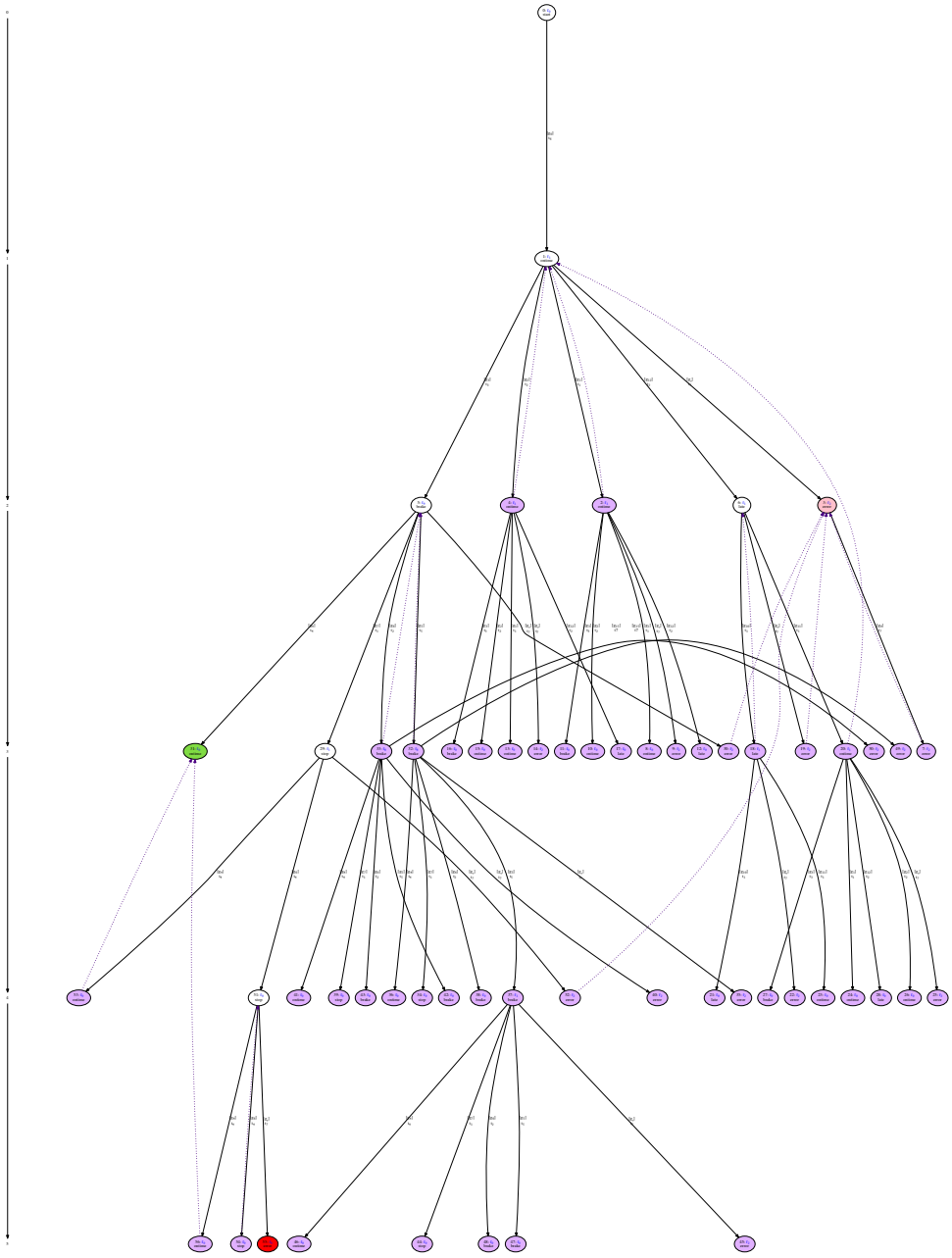


Figure 98: Step 34: Expansion of 51

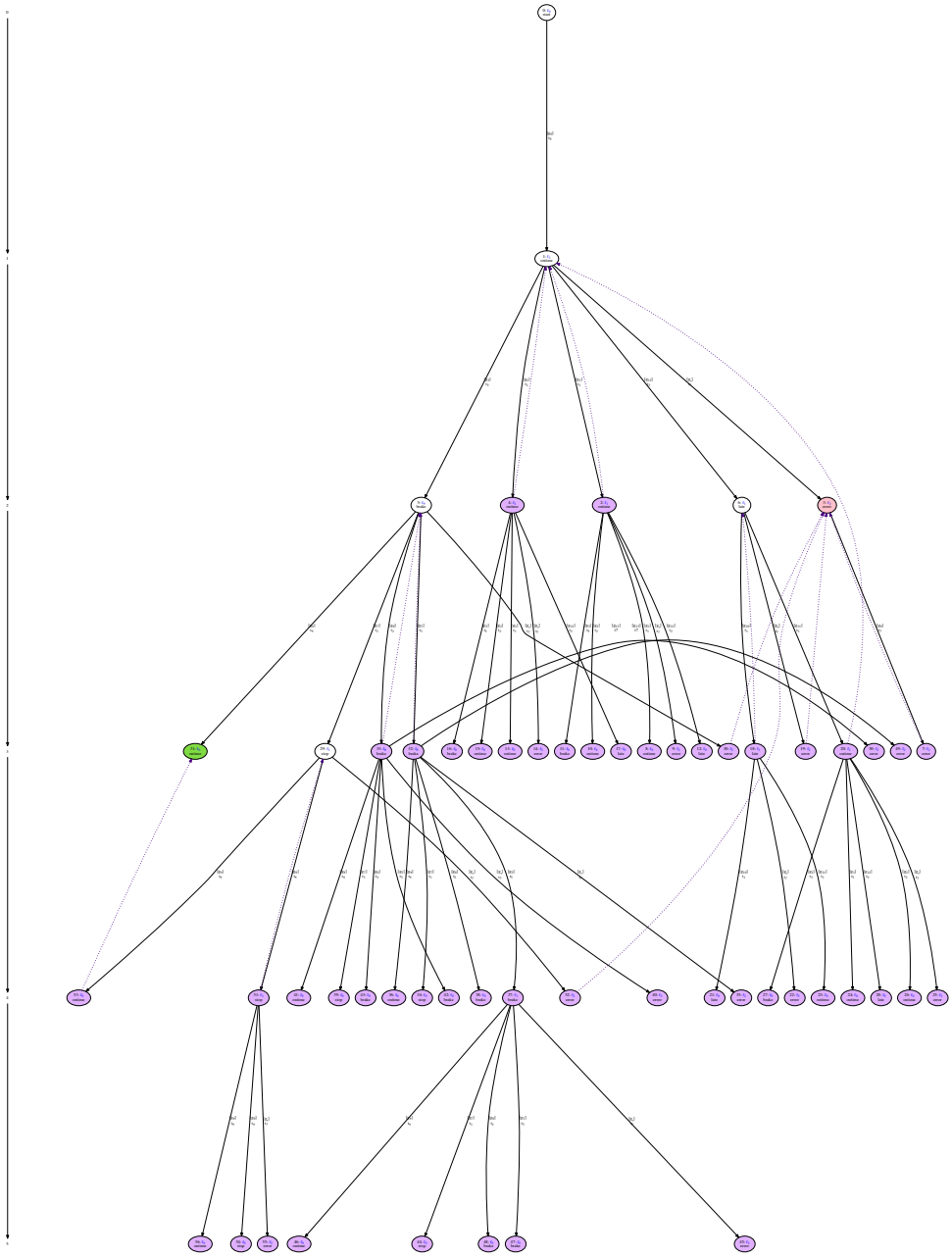


Figure 99: Step 35: Refinement of 55

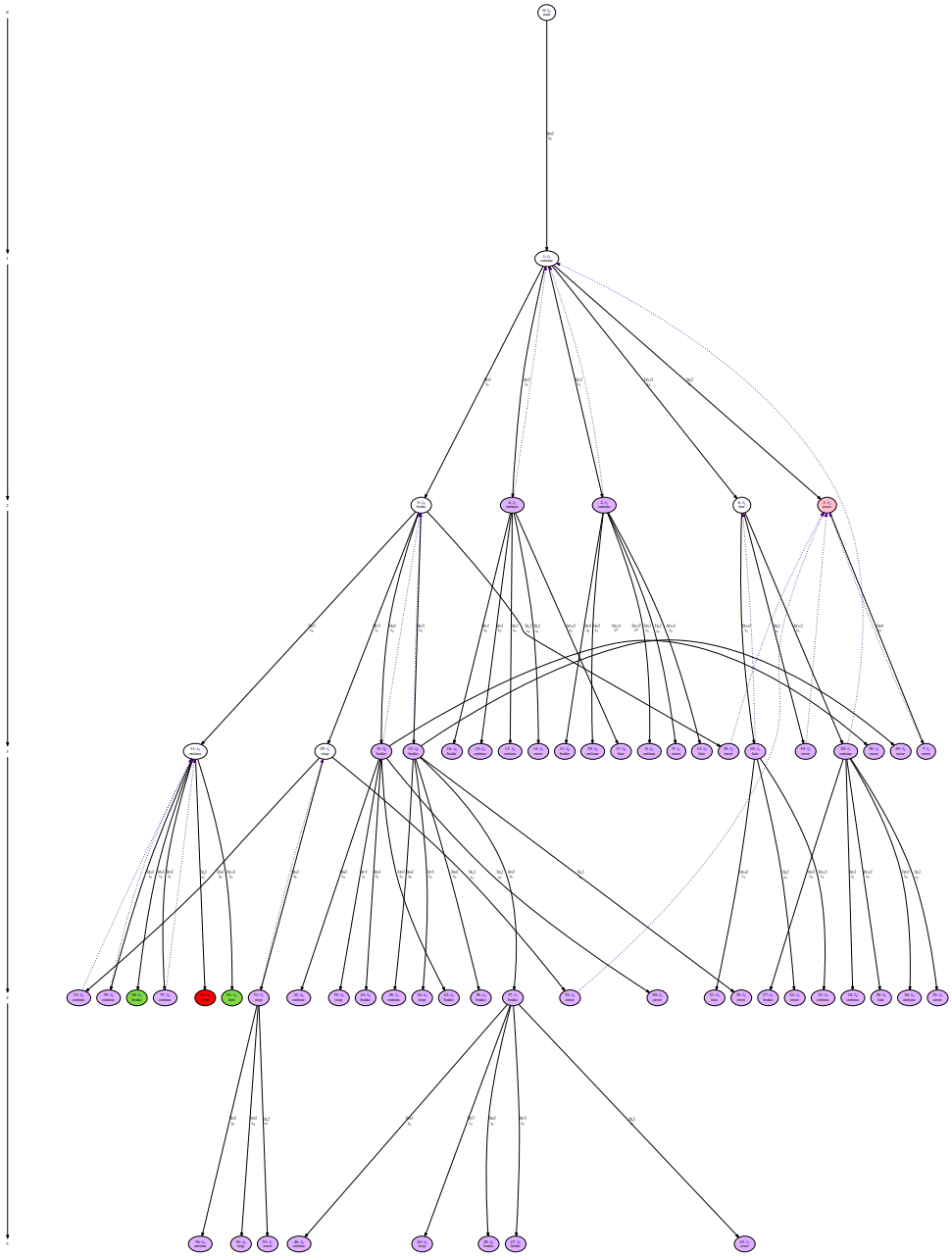


Figure 100: Step 36: Expansion of 31

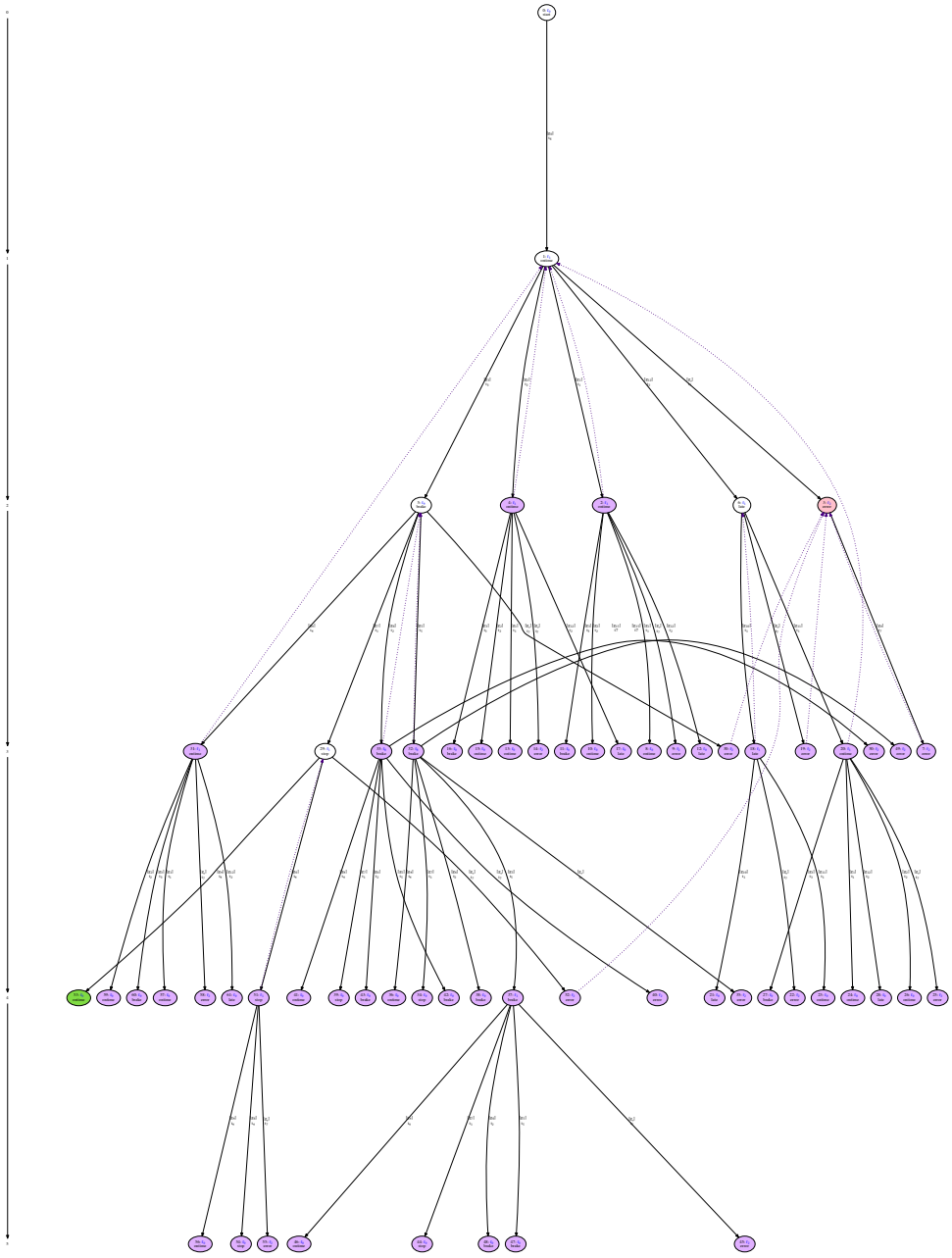


Figure 101: Step 37: Refinement of 58

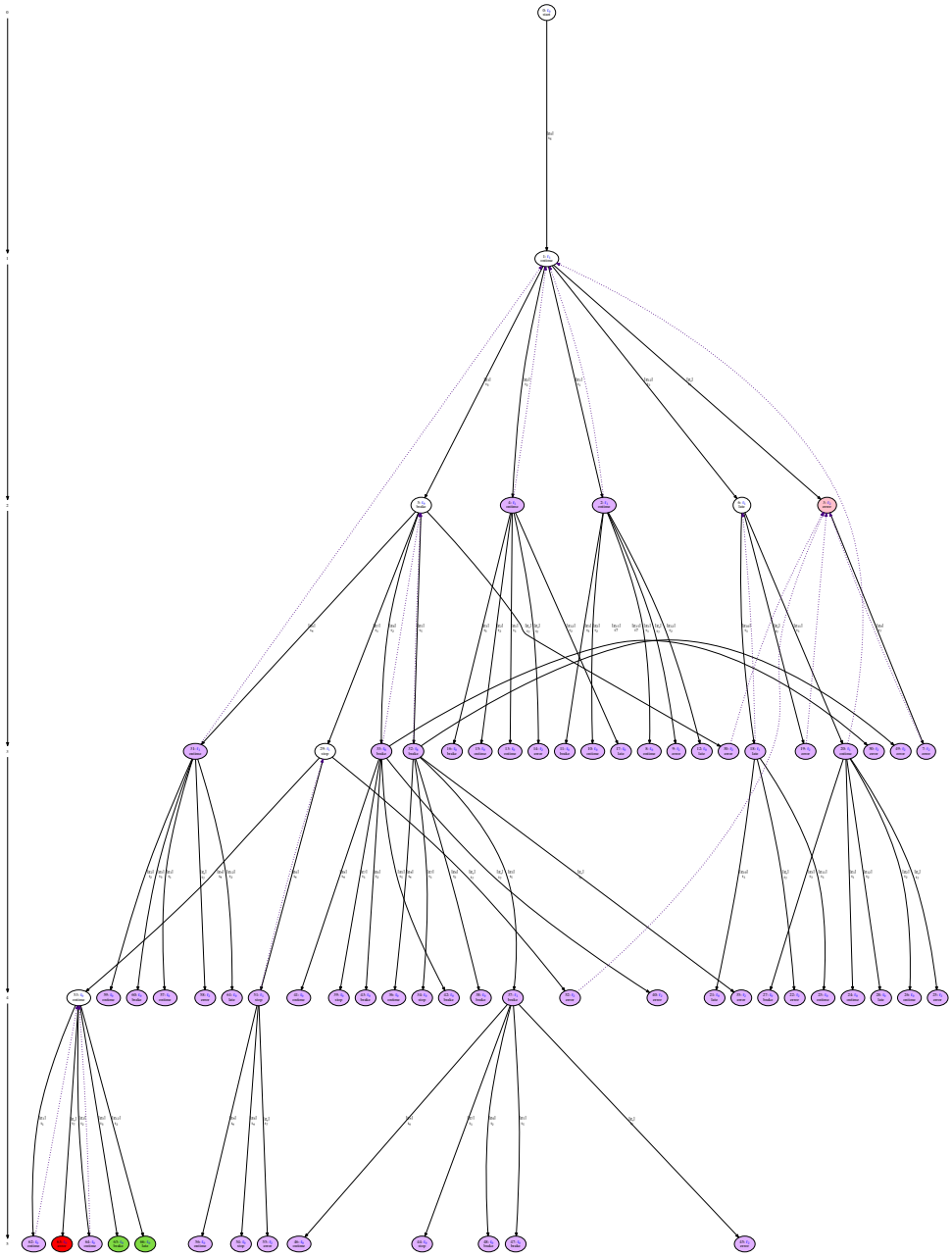


Figure 102: Step 38: Expansion of vertex 53

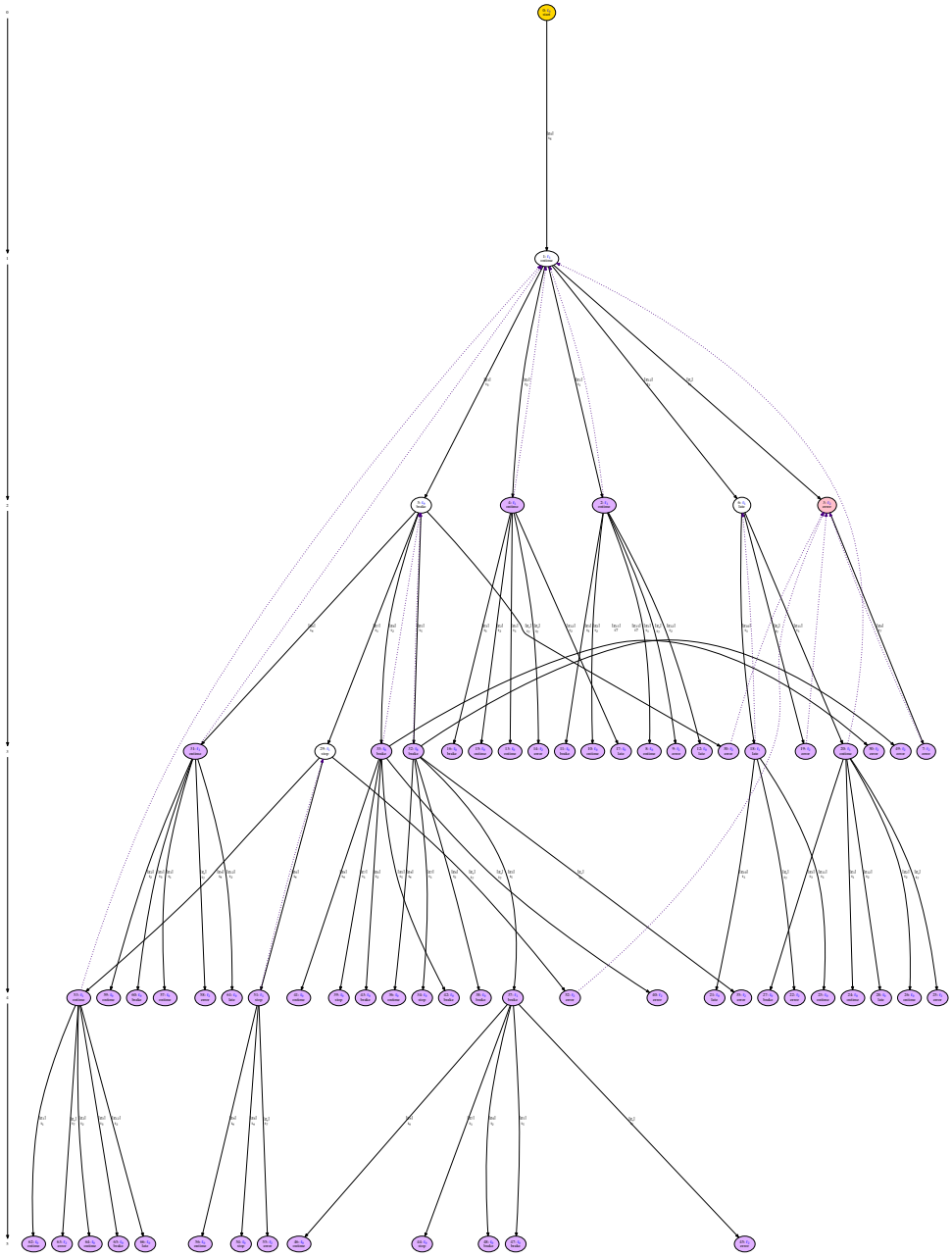


Figure 103: Step 39: Refinement of 63