Program Verification Summary

Ulla Aeschbacher

4.7.19

Contents

0 Overview

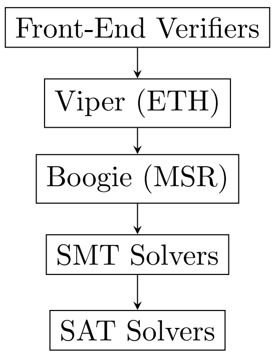


Figure 1: Overview

0.0.1 Questions at each level:

- What features and abstractions are provided?
- How are problems solved?
- How do we encode higher-level problems?

1 SAT Solving Algorithms

1.0.1 The SAT problem: Given a propositional formula with free variables, can we find a way to assign the free variables to make the formula true?

1.0.2 SAT solver: An efficient tool to automatically answer this question. This is the classic NP-complete problem. Thus the worst case complexity is exponential. But average cases encountered in practice can be handled much faster.

1.1 Definitions

1.1.1 Propositional variables: An alphabet $p, q, r, \ldots, p_1, p_2$

1.1.2 Propositional formulas: $A, B ::= p \mid \top \mid \bot \mid \neg A \mid A \wedge B \mid A \vee B \mid A \Rightarrow B \mid A \Leftrightarrow B$

1.1.3 Propositional model: A propositional model M maps propositional values to truth values

1.1.4 Satisfaction: A formula A is satisfied by a model M, written $M \models A$, by usual semantics.

1.1.5 Validity: A formula A is valid iff for all models M holds $M \models A$.

1.1.6 Satisfiability: A formula A is satisfiable iff for some model M holds $M \models A$.

1.1.7 Unsatisfiability: A formula A is unsatisfiable iff for no model M holds $M \models A$.

1.1.8 Entailment: A formula A entails a formula B, written $A \models B$, iff for all models M holds that if $M \models A$ then $M \models B$.

1.1.9 Equivalence: Formulas A and B are equivalent, written $A \equiv B$, iff for all models M holds $M \models A$ iff $M \models B$.

1.1.10 Propositional Equivalences:

- $\neg\neg A \equiv A$ and $\neg\top \equiv \bot$ and $\neg A \equiv A \Rightarrow \bot$
- $A \wedge B \equiv B \wedge A$ and $A \vee B \equiv B \vee A$ and $A \Leftrightarrow B \equiv B \Leftrightarrow A$
- $A \wedge \top \equiv A$ and $A \wedge \bot \equiv \bot$ and $A \vee \top \equiv \top$ and $A \vee \bot \equiv A$
- $(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$ and $(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$
- $\neg(A \vee B) \equiv \neg A \wedge \neg B$ and $\neg(A \wedge B) \equiv \neg A \vee \neg B$
- $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A) \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$
- $A \Rightarrow B \vee C \equiv A \wedge \neg B \Rightarrow C$ and $A \wedge B \Rightarrow C \equiv A \Rightarrow \neg B \vee C$

1.1.11 Literal: A literal is a variable or the negation of one. We write $\sim l$ for the negation of literal l.

1.1.12 Clause: A clause is a disjunction of literals $p \vee q \vee r \vee \ldots$. The empty clause is defined to be $\bot$.

1.1.13 Unit Clause: A unit clause is a single literal.

1.1.14 Conjunctive Normal Form CNF: A formula is in conjunctive normal form iff it is a conjunction of clauses. An empty conjunction is defined to be $\top$.

1.2 DPLL Algorithm

1.2.1 Equi-Satisfiability: A and B are equi-satisfiable iff either both or neither are satisfiable.

1.2.2 How SAT solving algorithms operate: SAT solving algorithms typically rewrite the input formula into an equi-satisfiable one in CNF, then rewrite the formula into further equi-satisfiable CNF formulas and/or perform a back-tracking search to explore different potential models. They terminate when the current formula is clearly equivalent to true or false.

1.2.3 Simple Conversion to CNF:

1. Rewrite all $A \Rightarrow B$ and $A \Leftrightarrow B$.
2. Push all negations inward.
3. Rewrite $\neg\neg A$ to A.
4. Eliminate $\top$ and $\bot$.
5. Distribute disjunctions over conjunctions.
6. Simplify.

1.2.4 Davis-Putnam-Logemann-Loveland DPLL Algorithm: We assume an input formula A in CNF, equivalently representable as a set of clauses or a set of literals. E.g. $(p \vee r) \wedge (\neg r \vee s \vee t)$ becomes $\{\{p, r\}, \{\neg r, s, t\}\}$. The algorithm rewrites clauses until the set is empty, indicating *sat*, or a clause is empty, indicating *unsat*. The algorithm builds up a partial model M, which assigns truth values to only some variables. We represent partial models using finite sets of literals. E.g. $M = \{p, \neg r\}$.

1.2.4.1 Pure Literal Rule: If p occurs only positively/negatively in A, delete clauses of A in which p occurs. Then update M to $M \cup \{p\}/M \cup \{\neg p\}$.

1.2.4.2 Unit Propagation: If l is a unit clause in A, remove all clauses from A which have l as a disjunct and update all clauses in A containing $\sim l$ by removing that disjunct. Then update M to $M \cup \{l\}$.

1.2.4.3 Decision: If p occurs both positively and negatively in clauses of A, apply the algorithm to $(M \cup \{p\}, A \wedge p)$. If we get (sat, M) then return that. Otherwise, apply the algorithm to $(M \cup \{\neg p\}, A \wedge \neg p)$ and return the result.

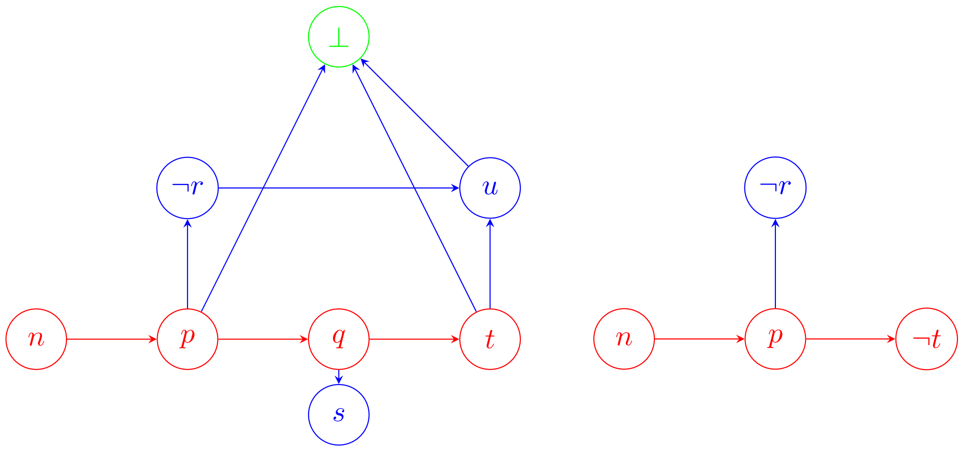
1.2.5 Implication graph: We can visualize the search for a model with an implication graph. We use a red circle to indicate decision literals and red arrows to indicate the order of decisions. We use blue circles for literals added due to unit propagation and blue arrows to record its dependencies. When a clause becomes empty, we add a green conflict node and add blue arrows analogously to unit propagation, so we can see which decisions had to do with the conflict.

1.3 CDCL Algorithm

1.3.1 Conflict-Driven Clause Learning CDCL Algorithm: This algorithm is better than DPLL in practice, but less so for random SAT examples. In practice, clauses are also periodically removed during SAT solver runs.

1.3.1.1 Back-jumping: When a conflict is found, identify the relevant decision literals, then pop the relevant literal and as many other non-relevant literals from the stack as possible. Then flip the relevant literal and go on.

1.3.1.2 Clause Learning: When a conflict is found, draw a cut in the graph, separating the conflict node from all decision literals. All edges should go from left to right. For nodes with outgoing edges crossing the boundary, save the disjunction of the negations of their literals. When back-jumping, conjoin this new clause to the current formula.

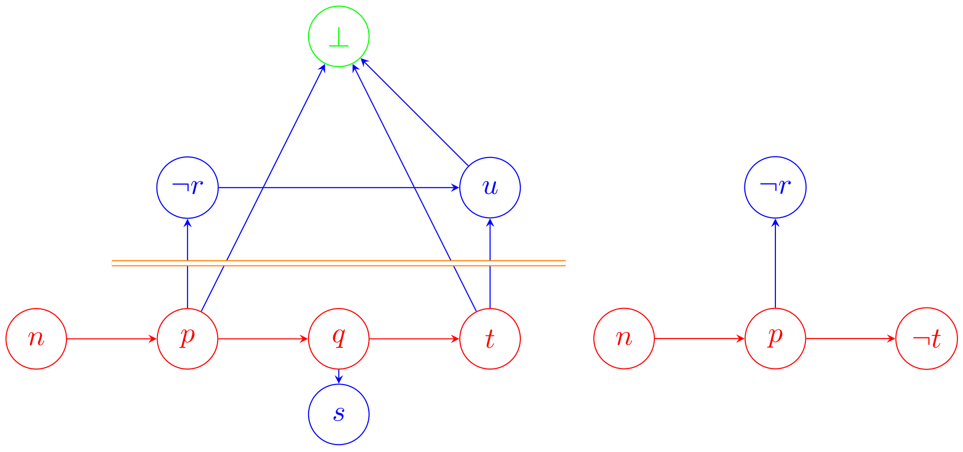


$((n \vee p) \wedge (\neg n \vee p \vee s) \wedge (\neg p \vee \neg r) \wedge (\neg u \vee t)$
$\wedge (r \vee q \vee t) \wedge (\neg q \vee s) \wedge (\neg p \vee t \vee u)$
$\wedge (\neg p \vee \neg t \vee \neg u) \wedge (r \vee \neg t \vee u),$

$\{n, p, q, t\})$

$((n \vee p) \wedge (\neg n \vee p \vee s) \wedge (\neg p \vee \neg r) \wedge (\neg u \vee t)$
$\wedge (r \vee q \vee t) \wedge (\neg q \vee s) \wedge (\neg p \vee t \vee u)$
$\wedge (\neg p \vee \neg t \vee \neg u) \wedge (r \vee \neg t \vee u)$
$\boldsymbol{\wedge \neg t},$
$\{n, p\})$

Figure 2: Example implication graph before and after back-jumping.



$((n \vee p) \wedge (\neg n \vee p \vee s) \wedge (\neg p \vee \neg r) \wedge (\neg u \vee t)$
$\wedge (r \vee q \vee t) \wedge (\neg q \vee s) \wedge (\neg p \vee t \vee u)$
$\wedge (\neg p \vee \neg t \vee \neg u) \wedge (r \vee \neg t \vee u),$

$\{n, p, q, t\})$

$((n \vee p) \wedge (\neg n \vee p \vee s) \wedge (\neg p \vee \neg r) \wedge (\neg u \vee t)$
$\wedge (r \vee q \vee t) \wedge (\neg q \vee s) \wedge (\neg p \vee t \vee u)$
$\wedge (\neg p \vee \neg t \vee \neg u) \wedge (r \vee \neg t \vee u)$
$\boldsymbol{\wedge \neg t \wedge (\neg p \vee \neg t)},$
$\{n, p, q, t\})$

Figure 3: Example implication graph before and after clause learning.

2 Encoding to SAT

2.0.1 Why SAT is not enough: Many problems include features beyond propositional logic, like

- first-order terms: variables, constants, functions, ...
- theories: integer arithmetic, sets, floating points, ...
- quantifiers: forall and exists

In most cases, such features are best handled natively with SMT, but in some cases, these features can be encoded into propositional logic. We can then directly use a SAT solver.

2.1 Definitions

2.1.1 Sort: We fix a set of sorts(types) $T_1, T_2, \ldots$ typically including Bool.

2.1.2 Term variables: For each sort, we fix an alphabet of term variables $x, y, \ldots, x_1, x_2 \ldots$.

2.1.3 Function symbols: We fix a set of function symbols $f, g, h, \ldots, f_1, f_2, \ldots$, each with a function signature.

2.1.4 First-order terms: We define first-order terms $t, s ::= x \mid f(t_1, t_2, \ldots)$.

2.1.5 Signature: A signature is a set of sorts and function symbols over those sorts.

2.1.6 First-order assertions: For a given signature, first-order assertions A are defined by $A, B ::= t \mid \neg A \mid A \wedge B \mid A \vee B \mid A \Rightarrow B \mid A \Leftrightarrow B \mid \forall x : T.A \mid \exists x : T.A$, where t is of sort Bool.

2.1.7 Value in a model: The value of a term t in a model M, written $\lceil t \rfloor_M$, is defined by $\lceil x \rfloor_M = M(x)$ and $\lceil f(t_1, t_2, \ldots) \rfloor_M = M(f)(\lceil t_1 \rfloor_M, \lceil t_2 \rfloor_M, \ldots)$.

2.1.8 Satisfaction: A formula A is satisfied by a model M, written $M \models A$, is defined by

- $M \models t$ iff $\lceil t \rfloor_M = true$
- $M \models \neg A$ iff $M \not\models A$
- $M \models \forall x : T.A$ iff $\forall v \in M(T), M[x \to v] \models A$
- $M \models \exists x : T.A$ iff $\exists v \in M(T), M[x \to v] \models A$

2.1.9 Interpretation: Sorts and function symbols are either interpreted or uninterpreted. Interpreted symbols have fixed interpretations, independent of the model.For example, Bool is interpreted and the boolean equality "=" is interpreted. Models specify the interpretations fro the free, uninterpreted symbols.

2.2 Ackermannization

2.2.1 General encoding recipe:

- Find a representation for the problem states.
- Define an encoding from the original problem into this representation.

- Identify and represent missing properties, make sure you don't add any properties.
- Optionally optimize redundancy in the representation/encoding.

2.2.2 Equality:

2.2.2.1 Signature definition: Consider a signature consisting of only a single uninterpreted sort T plus the interpreted sort Bool with constants $\top$ and $\bot$, plus the interpreted equality on T, written $=$. We consider only quantifier-free formulas.

2.2.2.2 Ackermannization: For each application of equality $x = y$ in the original formula, introduce a fresh variable $eq_{x,y}$ of sort Bool and replace all $x = y$ with $eq_{x,y}$. We also have to conjoin reflexivity ($eq_{x,x}, \ldots$), symmetry ($eq_{x,y} \Leftrightarrow eq_{y,x}$) and transitivity ($eq_{x,y} \wedge eq_{y,z} \Rightarrow eq_{x,z}$) to the formula then they are equi-satisfiable.

2.2.3 Equality and uninterpreted functions:

2.2.3.1 Signature definition: Consider a signature consisting of any number of uninterpreted sorts plus Bool with $\top$ and $\bot$, any number of uninterpreted function symbols over these sorts plus the equality functions "=" on each uninterpreted sort. Consider only quantifier-free formulas.

2.2.3.2 Ackermannization: For each function application $f(x)$ in the original formula with only variables as a parameter, introduce a fresh variable f_x of the same sort as the return sort of f and replace all occurrences of $f(x)$ with f_x. Repeat this process until the original formula contains no function symbols. Each time we introduce a variable f_x to replace $f(x)$, for each variable f_y already introduced to replace $f(y)$, we conjoin $x = y \Rightarrow f_x = f_y$ to the formula, then they are equi-satisfiable.

2.2.4 Finitely-bounded quantifiers: In general, reasoning about quantifiers needs more than a SAT solver, however finitely-bounded quantifiers can be easily handled. Two examples:

- $\forall x : Int.\ 0 \leq x \wedge x \leq 2 \Rightarrow A \equiv 0 \Rightarrow A \wedge 1 \Rightarrow A \wedge 2 \Rightarrow A$
- $\exists x : Int.\ 0 \leq x \wedge x \leq 2 \wedge A \equiv (0 \wedge A) \vee (1 \wedge A) \vee (2 \wedge A)$

3 SMT Solving

3.0.1 The SMT problem: Given a first-order logic formula with interpreted and uninterpreted symbols from possibly several theories, does there exist a model?

3.0.2 Approaches to SMT: There are three approaches for theory support:

- Eager SMT desugars a theory into a SAT problem. This is only possible for certain limited theories.
- Lazy SMT integrates theory support into the propositional search.
- We can also model a theory using other supported features. This is typically done by combining uninterpreted functions/sorts and quantified axioms.

We will only look at lazy SMT in this lecture.

3.1 Definitions

3.1.1 Theory: A theory T is a pair of a signature $sig(T)$ and a set of models $models(T)$. The domain of all $models(T)$ must be $sig(T)$ and all elements of $models(T)$ must interpret interpreted symbols as usual. One example is the theory of non-linear arithmetic T_Z has $sig(T_Z) = \{Int, 0, 1, +, -, \cdot, /, \leq\}$.

3.1.2 S-formula: For a signature S, a formula A is a S-formula iff A contains only function symbols belonging to S and uninterpreted constant symbols.

3.1.3 T-formula: A formula A is a T-formula iff A is a $sig(T)$-formula.

3.1.4 T-satisfiability: A formula A is T-satisfiable iff there exists $M \in models(T)$ such that $M \models A$.

3.1.5 T-entailment: A set of T-formulas Γ T-entails a T-formula A, written $\Gamma \models_T A$, iff for all $M \in models(T)$ holds that if M satisfies all formulas in Γ, then $M \models A$.

3.1.6 T-consistency: A set of T-formulas Γ is T-consistent iff $\Gamma \not\models_T \bot$.

3.1.7 Disjointedness: Two theories T_1 and T_2 are disjoint if their signatures overlap only on standard elements and uninterpreted constants.

3.1.8 Atom: An atom is a formula without propositional connectives or quantifiers. For example $f(a) = b$ or $m \cdot n \leq 42$, but not $\neg(f(a) = b)$.

3.1.9 First-order literal: A first-order literal is an atom or its negation.

3.1.10 T-atom: A T-atom is a T-formula which is also an atom.

3.1.11 S-atom: A S-atom is a S-formula which is also an atom.

3.1.12 T-literal: A T-literal is a T-formula which is also a literal.

3.1.13 S-literal: A S-literal is a S-formula which is also a literal.

3.1.14 Theory solver: A theory solver should be able to answer the following questions:

- Is a given set of T-literals Γ T-consistent? If so what is a T-model? If not, what is a preferably minimal T-inconsistent subset of Γ?

- For a T-consistent set Γ of T-literals, are there extra implied literals?
- Fir a T-consistent set Γ, are there any implied interface equalities?

3.2 DPLL(T) Algorithm

3.2.1 Propositional abstraction: For a given signature S we define a signature S^P containing only a fresh uninterpreted Bool constant for each S-atom. We then fix a bijective mapping from the S-atoms to the S^P-atoms. For a S-formula A, the propositional abstraction of A, written A^P, is defined by replacing all atoms in A with their image in this mapping.

3.2.2 Propositional unsatisfiability: An S-formula A is propositionally unsatisfiable if $A^P \models \bot$. $A^P \models \bot \Rightarrow A \models \bot$ but not necessarily vice versa.

3.2.3 Propositional entailment: An S-formula A propositionally entails an S-formula B, iff $A^P \models B^P$. $A^P \models B^P \Rightarrow A \models B$ but not necessarily vice versa.

3.2.4 Adapting DPLL to DPLL(T): We run a DPLL-like search on the propositional abstraction A^P of the input T-formula A until we either reach a conflict, where we backtrack/-jump as usual, or find a model represented by a set of literals Γ. If it is a T-model, we ask the theory solver if T is T-consistent. If yes, we are done. Otherwise we backtrack in the original search. The theory solver can tell us which literals contradict so we can do some CDCL-style clause learning.

3.3 Theory Combination

3.3.1 Theory combination: We want to solve problems that have multiple disjoint theories.

3.3.2 Extension: A model M' is an extension of a model M if for every uninterpreted sort $S \in \mathit{dom}(M)$, $S \in \mathit{dom}(M')$ and $M(S) \subseteq M'(S)$ and for every term t holds that if $\lceil t \rfloor_M$ is defined then $\lceil t \rfloor'_M$ is defined and $\lceil t \rfloor_M = \lceil t \rfloor_M$.

3.3.3 Stably-infinite: A theory T is stably-infinite, if for any $M \in \mathit{models}(T)$ and T-formula A such that $M \models A$, we have $M' \models A$ in an extension M' of M.

3.3.4 Purification: For each literal l in A, involving terms from both theories, select a subterm t of l containing symbols from only one theory but not the other. Choose a fresh constant symbol c. Rewrite l in A by replacing t with c and conjoin $c = t$ to A. Repeat this until each literal's functions are from a single theory plus the shared constants.

3.3.4.1 Shared Constants: The newly introduced constants plus the old variables are together called the shared constants.

3.3.5 Craig's Interpolation Theorem: For a first-order A and B holds that if $A \models B$ then there exists C such that $A \models C$ and $C \models B$ and the only function symbols in C are those occurring in both A and B and such that C is not A or B.

3.3.6 Non-deterministic Nelsen-Oppen combination: Suppose we have an input formula A including only two disjoint, stably-infinite theories T_1 and T_2 for which we have theory solvers. Then we can proceed as follows:

1. Purify A to B, let S be the shared constants.

2. Pick any equivalence relation on S and according to it conjoin either an equality or disequality to B for each pair of elements in S.

3. Use the theory extension of DPLL modified as follows: when we have to check a candidate model using the theory solvers, we check the consistency of the model but filtered for each theory solver. If both find the models consistent, we return *sat*, otherwise we process a conflict as usual.

4. If the DPLL search fails to find a model, we return to step 2 and choose a new equivalence relation on S. If all have been tried (this could take exponential time), we return *unsat*.

3.3.7 Convexity: A theory T is convex if for all finite sets of literals Γ and for all non-empty disjunctions of equalities $\bigvee_{i\in\{0,\ldots,n\}} x_i = y_i$ holds that if $\Gamma \models_T \bigvee_{i\in\{0,\ldots,n\}} x_i = y_i$ then $\Gamma \models_T x_j = y_j$ for some $j \in \{0, \ldots, n\}$.

3.3.8 Implications for convex theories: For convex theories T it is sufficient to share only the equality information but not the inequality information between the solvers. This is because if there exists a model M which is consistent with theory T and satisfies all of the implied equalities according to T, there will also exist a model in which all other pairs of shared constants are not equal to each other. For example if theory T says that $x = z$ is implied by the model, but does not say anything about (x, y) or (y, z) it is guaranteed that there is also a model M' in which $x \neq y$ and $y \neq z$ because otherwise $(x = y \vee y = z)$ would have been implied.

3.3.9 Deterministic Nelsen-Oppen combination: Suppose we have an input formula A including only two disjoint, convex, stably-infinite theories T_1 and T_2 for which we have theory solvers. Then we can proceed as follows:

1. Purify A to B, let S be the shared constants.

2. Use the theory extension on DPLL modified as follows: when we have to check a candidate model using the theory solvers, we check the consistency of the model, but filtered for each theory solver. If a theory solver finds an inconsistency, we process the conflict as usual. If a theory solver finds a model consistent, ask it for the set of implied equality literals. If there are any new ones, add these to the current model and run the other solver. Repeat this until no new equalities are added.

3. If no solver finds the model to be inconsistent, we return *sat*, otherwise we return *unsat*.

4 Quantifiers

4.0.1 Approaches: We can either focus on model finding for which both an eager and a lazy approach exist. Or we can use E-matching, which does not generate models, but is good for *unsat* results which are important for program verification.

4.1 Definitions

4.1.1 Quantified Literal: A quantified literal is a formula $\forall x : T\ A$ or its negation $\neg\forall x : T\ A$. Remember that $\exists x : T\ \neg A \equiv \neg\forall x : T\ A$.

4.1.2 Ground term: A ground term is a term containing no (quantified) variables.

4.1.3 Equalities:

- $\forall x_1 \forall x_2 A \equiv \forall x_2 \forall x_1 A$
- $\exists x_1 \exists x_2 A \equiv \exists x_2 \exists x_1 A$
- $\neg\forall x A \equiv \exists x \neg A$ and $\neg\exists x A \equiv \forall x \neg A$
- $\forall x(A \wedge B) \equiv (\forall x A) \wedge (\forall x B)$
- $\exists x(A \vee B) \equiv (\exists x A) \vee (\exists x B)$
- If $x \notin FV(A)$ then $\exists x A \equiv A \equiv \forall x A$ and $\forall x(A \vee B) \equiv A \vee (\forall x B)$ and $\exists x(A \wedge B) \equiv A \wedge (\exists x B)$ where $FV(A)$ are the free variables in A

4.2 Eager Approach

4.2.1 Skolemization: Existential quantifiers in positive positions can be eliminated by replacing $\exists x A$ with $A[c/x]$ where c is a fresh constant. In general, Skolemization replaces an $\exists$-bound variable with a fresh function of the $\forall$-bound variables enclosing it. For example $\forall x : Int\ \exists y : Int\ (y > x)$ becomes $\forall x : Int\ (f(x) > x)$.

4.2.2 Eager Quantifier Elimination: We can allow only formulas of the form $\exists x_1 \dots \exists x_m$ $\forall y_1 \dots \forall y_n A$ where A is quantifier free and we have no function symbols except constants and equality. Then we can apply Skolemization to remove the $\exists$ quantifiers. After that it is sufficient to instantiate the $\forall$ quantifiers for each constant. For example $\forall x : T\ A$ becomes $A[c_1/x] \wedge A[c_2/x] \wedge \dots$. The result is equi-satisfiable and quantifier-free.

4.3 Lazy Approach

4.3.1 Extended clause: An extended clause is a disjunction of first-order literals and quantified literals.

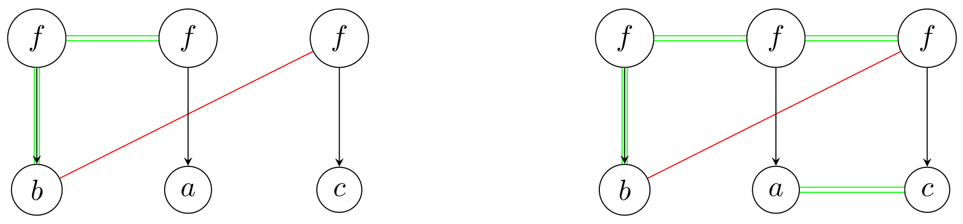
4.3.2 Extended CNF: A formula is in extended CNF iff it is a conjunction of extended clauses. Two examples:

- $(\forall x : T\ p(x)) \Rightarrow \forall y : T\ q(y)$ has extended CNF form $(\neg\forall x : T\ p(x)) \vee \forall y : T\ q(y)$
- $(\exists x : T\ \neg p(x)) \vee \forall y : T\ q(y)$ has extended CNF form $\neg p(c) \vee \forall y : T\ q(y)$

4.3.3 Lazy Quantifier Elimination: We extend propositional abstraction to also abstract quantified literals. For example $\neg p(c) \vee \forall y : T\ q(y)$ becomes $\neg a \vee b$. We do not cover more details in this lecture.

4.4 E-Matching

4.4.1 E-graph: We use this graph to represent equalities and disequalities. We have a node labelled f for each ground term $f(\ldots)$. We have directed, indexed edges to each function argument. We express equality with a green double line and disequality with a red single line. When adding an a to b equality edge, we need to find the pairs of nodes for each function f. If the arguments on the two f nodes are pairwise equal, we equate them too. This is an efficient way to track (dis)equalities and yields a theory solver for T_E.



$f(b) = b, f(b) = f(a), \neg(f(c) = b)$ $\qquad$ $f(b) = b, f(b) = f(a), \neg(f(c) = b), a = c$

Figure 4: Example E-graphs

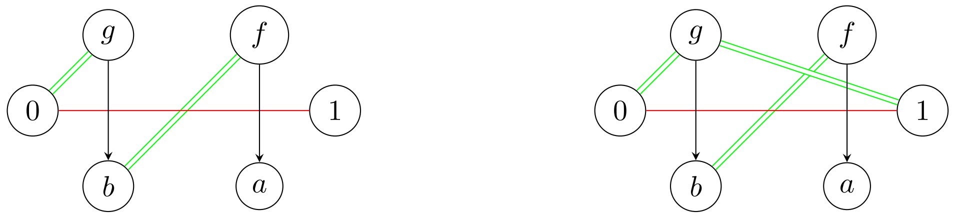
4.4.2 Extended quantifier syntax: We add two extensions to $\forall$-quantified formulas:

4.4.2.1 Merging quantifiers: We allow multiple adjacent $\forall$-quantifiers to be merged into one. For example $\forall x_1 \forall x_2 A$ becomes $\forall x_1, x_2 A$.

4.4.2.2 Triggers: We allow a trigger to be attached to any $\forall$-quantifier. For example $\forall x\{t\}A$. This term t is of any sort and satisfies the following: t must contain all the variables quantified by the quantifier, t may not contain interpreted function symbols except for constants and t must contain at least one non-constant function symbol. For example $\forall x : Int\ \{f(x)\} f(x) < b$.

4.4.3 Matching a trigger: A term g matches a trigger t if g is a ground term $t[t'/x]$. A $\forall$-quantifier will only be instantiated if a ground term matches it. For example in the formula $g(f(a)) = 0 \wedge \forall x : Int\ \{g(f(x))\} g(f(x)) = 1$ the term $g(f(a))$ matches the trigger $g(f(x))$, causing the instantiation $g(f(a)) = 1$.

4.4.4 E-matching: A quantifier $\forall x\{t\}A$ will only be instantiated when there are ground terms t' and t'' such that t'' occurs in our current formula to satisfy / our current model and $t[t'/x] = t''$ is true in our current model. Triggers are matched modulo equalities. E-matching can be efficiently implemented by pattern-matching triggers against the current E-graph.



$$g(b) = 0 \land b = f(a)$$
$$\land \forall x : Int\ \{g(f(x))\}\ g(f(x)) = 1$$

$$g(b) = 0 \land b = f(a)$$
$$\land \forall x : Int\ \{g(f(x))\}\ g(f(x)) = 1$$

Figure 5: E-graph before and after E-matching

4.4.5 Integrating E-matching:

1. Run a DPLL-like search on the propositional abstraction A^P of the input formula A.
2. Maintain the current (dis)equality information in an E-graph.
3. When a quantified literal is added to the candidate model, record the quantifier for potential E-matching later.
4. Periodically, run an E-matching engine on the current E-graph. Look for new instantiations of recorded quantifiers and add them. This expands the current formula for the DPLL search. This procedure will only return *unsat* or *unknown*, no model.

4.4.6 Selecting triggers: Triggers may be too restrictive, leading to missed relevant instances or too permissive, leading to too many instantiations. Triggers may in general consist of sets of terms, either $\{f(x), g(x)\}$, where it will only be instantiated when we have both $f(t)$ and $g(t)$ for some t or $\{f(x)\}\{g(x)\}$ where it will be instantiated if we have either $f(t)$ or $g(t)$ for some t.

4.4.7 Matching Loop: A situation that leads to infinite instantiations is called a matching loop.

5 Encoding to SMT

5.1 Defining the Language

5.1.1 Domains: Domains enable the definition of additional types, mathematical functions, and axioms that provide their properties. Syntactically, domains consist of a name (for the newly-introduced type), and a block in which a number of function declarations and axioms can be introduced.

5.1.2 Functions: Domain functions may have neither a body nor a specification. These are uninterpreted total mathematical functions.

5.1.3 Axioms: Domain axioms consist of name (following the axiom keyword), and a definition enclosed within braces (which is a boolean expression which may not read the program state in any way).

```
domain Nat {
    function zero(): Nat
    function succ(n: Nat): Nat
    function plus(m: Nat, n: Nat): Nat
    axiom plus_zero {
        forall n: Nat :: {plus(zero(), n)} plus(zero(), n) == n
    }
}
```

Code 1: Simple example program.

5.1.4 Methods: We write methods taking arbitrary parameters to test properties of our axiomatization. Method bodies will be a sequence of assume and assert statements.

```
method test(p: IntPair, q: IntPair) {
    assert pair(1, 2) != pair(2, 1)
    assert fst(p) == fst(q) && snd(p) == snd(q) ==> p == q
}
```

Code 2: Simple example method.

5.2 Adding Functionality

5.2.1 Tags: To define which case of the data type definition a value comes from, we add a tag function.

5.2.2 Limited functions: To deal with matching loops, we want to unroll the definition just once for each original occurrence of the function. To do that we add limited functions.

5.2.3 Sequences: Sequences can be indexed by integers and store some (fixed) type T of values. Sequences are not functional lists and are not classical ADT's, because there can be many ways to construct the same sequence. In other words, there is no canonical constructor for arbitrary-length sequences. To check for equality we nee extensionality, i.e. observational equality.

```
function zero(): Nat
function succ(n: Nat): Nat
function tag(n: Nat): Int
axiom tag_zero {
    tag(zero()) == 0
}
axiom tag_succ {
    forall n: Nat :: {succ(n)} tag(succ(n)) == 1
}
axiom all_tags {
    forall n: Nat :: {tag(n)} (tag(n) == 0 && n == zero())
        || (tag(n) == 1 && exists m: Nat :: n == succ(m))
}
```

Code 3: Using tags.

```
function plus(m: Nat, n: Nat): Nat
axiom plus_def {
    forall m: Nat, n: Nat :: {plus(m, n)}
        plus(m, n) == (tag(m) == 0 ? n : succ(plusL(pred(m), n)))
}
function plusL(m: Nat, n: Nat): Nat
axiom plus_limited {
    forall m: Nat, n: Nat :: {plus(m, n)} plus(m, n) == plusL(m, n)
}
```

Code 4: Using limited functions.

6 Weakest Preconditions

6.1 Definitions

6.1.1 Partial correctness: A program s is partially correct with respect to pre- and post-conditions A_1 and A_2 iff all executions of s starting from states satisfying A_1 are free of runtime errors and any such executions which terminate will do so in states satisfying A_2.

6.1.2 Hoare triple: We express the notion of partial correctness using a Hoare triple $\{A_1\}s\{A_2\}$.

6.1.3 Definition of small imperative language: Our language has program variables $x, y, z, \ldots$, expressions $e, e_1, e_2, \ldots$, assertions A and statements s. We assume small-step operational semantics. A runtime configuration is a pair (s, σ) of a statement s and a runtime state σ

6.1.3.1 Statements:

- Standard statements: `skip`, x `:=` e, s_1`;` s_2, `if` (b) $\{s_1\}$ `else` $\{s_2\}$, `while` (b) $\{s\}$
- Verification statements: `assert` A, `assume` A
- Non-deterministic statements: s_1`[]`s_2, `havoc` x

6.1.3.2 Runtime state: A runtime state is a mapping σ from variables to values.

6.1.3.3 Trace: A trace is either an infinite sequence of runtime configurations or a finite sequence of runtime configurations appended with one of the following: a single runtime state σ (normal), the symbol *error* (for a failing execution) or the symbol *magic* (when the trace just kinda disappears).

6.1.4 Partial correctness again: We write $\models \{A_1\}s\{A_2\}$ and say s is partially correct with respect to precondition A_1 and post-condition A_2 if the following is true: The set of all possible traces starting from some (s, σ_1) with $\sigma_1 \in \Sigma_1 = \{\sigma_1 \mid \sigma_1 \models A_1\}$ does not contain any failing traces and for final states σ_2 of each of the complete traces in the set we have $\sigma_2 \models A_2$.

6.2 Weakest Precondition Function

6.2.1 Predicate transformer semantics: These semantics describe how statements transform assertions. We will start from one postcondition and work backwards through s. At each sub-statement, we find the logically weakest precondition which works for this statement.

6.2.2 Weakest precondition function: The weakest precondition function $wlp(s, A)$ returns an assertion such that we have

- Soundness: for all s, A: $\models \{wlp(s, A)\}s\{A\}$
- Minimality: for all s, A_1, A_2: if $\models \{A_1\}s\{A_2\}$ then $A_1 \models wlp(s, A_2)$
- Computability: for all s, A: $wlp(s, A)$ is computable

If we had this, we could build a program verifier checking $\models \{A_1\}s\{A_2\}$ by computing $wlp(s, A_2)$ and then checking that $A_1 \wedge \neg wlp(s, A_2)$ is *unsat*.

6.2.3 *wlp* for our small imperative language: This is sound, but the completeness depends on loop invariants.

- $wlp(\texttt{skip}, A) = A$
- $wlp(x := e, A) = A[e/x]$
- $wlp(s_1\texttt{;}\ s_2, A) = wlp(s_1, wlp(s_2, A))$
- $wlp(\texttt{if (}b\texttt{)}\ \{s_1\}\ \texttt{else}\ \{s_2\}, A) = (b \Rightarrow wlp(s_1, A)) \wedge (\neg b \Rightarrow wlp(s_2, A))$
- $wlp(\texttt{while (}b\texttt{) invariant}\ A_I\ \{s\},\ A) = wlp(\texttt{assert}\ A_I\texttt{; havoc}\ x_1\texttt{; ...; havoc}\ x_n\texttt{;}$
  $\texttt{((assume}\ A_I \wedge b\texttt{;}\ s\texttt{; assert}\ A_I\texttt{; assume false) [] (assume}\ A_I \wedge \neg b\texttt{)),}\ A)$
  $= A_I \wedge \forall \vec{y}\ \texttt{((}A_I \wedge b \Rightarrow wlp(s\texttt{,}\ A_I\texttt{))}\ \wedge\ \texttt{(}A_I \wedge \neg b \Rightarrow A\texttt{))}\ \texttt{[}\vec{y}/\vec{x}\texttt{]}$
- $wlp(\texttt{assert}\ A_1, A) = A_1 \wedge A$
- $wlp(\texttt{assume}\ A_1, A) = A_1 \Rightarrow A$
- $wlp(s_1\,\texttt{[]}\,s_2, A) = wlp(s_1, A) \wedge wlp(s_2, A)$
- $wlp(\texttt{havoc}\ x, A) = \forall v A[v/x]$

7 Refined Verification Condition Generation

7.1 Problems with *wlp*

7.1.1 Problems with our *wlp* definition: Our current *wlp* definition can generate (exponentially) large formulas, because for branching and non-deterministic statements we copy A, it doesn't tell us which assertion(s) in the program potentially fail and we don't know how many assertion violations are possible.

7.1.2 Dynamic single assignment: As a solution to the above problems, we could define $wlp(s_1 \texttt{ [ ] } s_2, A) = (p \Leftrightarrow A) \Rightarrow wlp(s_1, p) \land wlp(s_2, p)$ then we wouldn't duplicate A. But this doesn't work as long as we have variables with changing values. Thus we first need to eliminate assignments. A program is defined to be in dynamic single assignment (DSA), if in each trace of the program, each variable is assigned to at most once.

7.1.3 Converting to DSA:

1. Desugar loops.
2. Introduce versions of variables, track the latest version of each variable and use this in expressions. Assignments increment the version. `havoc` is replaced with `skip` but also increments the version.
3. Process branches independently. Per variable, if there are different final versions used in the two branches, introduce a version unused in both branches and assign the latest value to this in each branch.
4. Eliminate all variable assignments $x := e$ and replace them by `assume` $x = e$.
5. Optionally we can also rewrite `skip` to `assume true` and `if (`b`)` $\{s_1\}$ `else` $\{s_2\}$ to `(assume` b`;` s_1`) [] (assume` $\neg b$`;` s_2`)`.

7.2 Definition of wlp^*

7.2.1 Generalizing *wlp*: We want to track multiple verification conditions separately. For this we generalize our *wlp* operator to wlp^* working on a multiset of assertions Δ. Each element of our multiset comes from a distinct program point.

7.2.2 wlp^* for our small imperative language: Most of the statements behave the same as in *wlp* (just replacing A with Δ) except for the following:

- $wlp^*(s_1\texttt{;}\ s_2, \Delta) = wlp^*(s_1, wlp^*(s_2, A))$
- $wlp^*(\texttt{assert } A_1, \Delta) = \Delta \cup \{A_1\}$
- $wlp^*(\texttt{assume } A_1, \Delta) = \{A_1 \Rightarrow A \mid A \in \Delta\}$
- $wlp^*(s_1 \texttt{[ ]} s_2, \Delta) = wlp^*(s_1, \Delta) \cup wlp^*(s_2, \Delta)$

To verify a Hoare triplet $\{A_1\}s\{A_2\}$ we check the entailment $A_1 \models A$ for each $A \in wlp^*(s, \{A_2\})$. If we also record the program point at which each element of our multisets originated, we can now easily report error locations. The disadvantage of this is that theory-specific work may be repeated for each entailment that is checked, as well as quantifier instantiations. This can make it rather slow.

8 Boogie: Intermediate Verification Language

8.0.1 Intermediate verification language: An intermediate verification language is designed for encoding higher-level program verification problems. The language features are tailored to express verification problems. The goal is that many higher-level languages can be encoded.

8.0.2 Boogie design choices: Boogie is designed to provide reusable language features for encoding imperative, heap-based programs, representing verification requirements and adding custom types, axioms and assumptions. It is also designed to be human readable and writable.

8.1 Basic Functionality

8.1.1 Boogie language: The correctness of programs means that there are no failing traces for each procedure. Procedures are given preconditions and postconditions, loops must be provided with appropriate invariants. Boogie has a verifier to check correctness and weakest preconditions are efficiently calculated by the Boogie verifier. It localizes errors and provides traces which reach the error location(s).

8.1.2 Procedures: A procedure consists of a name, optional type parameters, named input parameters and return parameters and a body. It can include pre- and postconditions as well as loop invariants.

8.1.3 Functions: A function consists of a name, optional type parameter, named or unnamed input parameters and return parameter and an optional body.

8.1.4 Axioms: Axioms declare an expression that should be assumed to be true for the entire lifetime of the program. A consequence of this is that axioms cannot refer to mutable global variables. They can also have triggers.

8.1.5 Custom type constructors: One can declare new types with the type constructor.

8.1.6 `old` expressions: An expression `old(g)` can be used in a postcondition to refer to the value that a global variable `g` had in the pre-state of the procedure call. But using `j == old(j)` for all untouched variables is cumbersome and non-modular.

8.1.7 `modifies` clause: We introduce the `modifies` clause per procedure to specify what values do change.

```
var i: int;
var j: int:
procedure inc_i()
    modifies i:
    ensures i == old(i) + 1:
{
    i := i + 1:
}
```

Code 6: `old` and `modifies` keywords.

```
type Pair S T;

function pair <S,T>(i: S, j: T) : Pair S T;
function fst <S,T>(p: Pair S T) : S;
function snd <S,T>(p: Pair S T) : T;

axiom (forall <S,T> s: S, t: T ::
    {pair(s,t)} fst(pair(s,t)) == s);
axiom (forall <S,T> s: S, t: T ::
    {pair(s,t)} snd(pair(s,t)) == t);
axiom (forall <S,T> p: Pair S T ::
    {fst(p)}{snd(p)} pair(fst(p),snd(p)) == p);

type IntPair = Pair int int;

procedure test() {
    assert snd(pair(1, 2)) == 2;
    assert pair(1, 2) != pair(2, 1);
    assert (forall p : IntPair, q: IntPair ::
    fst(p) == fst(q) && snd(p) == snd(q) ==> p == q);
}

procedure destruct(p: IntPair) returns (a: int, b: int);
    ensures a == fst(p) && b == snd(p);

procedure testTwo()
{
    var i: int;
    var j: int;
    call i,j := destruct(pair(1,2));
    assert j == 2;
}
```

Code 5: Simple Boogie program.

8.2 Weakest Preconditions

8.2.1 How to treat procedure calls: There are two possibilities, as well as hybrid models of both:

- Inline procedure calls: this can be done in Boogie via an {`:inline` d} attribute before the procedure name, d denotes the depth up to which it is inlined. The pros are that all the information is there and there is less effort for the user. The cons are that we leak implementation details, performance suffers, there is no support for recursion and inline code changes need re-verification.
- Use specification only: this is node in Boogie by default. The pros are increases in modularity and performance, possibility of recursion and re-verification is only needed when the specification changes. The cons are that it is more effort for the user to write pre- and postconditions and that the completeness depends on the specification.

8.2.2 *wlp* for procedure calls: In Boogie, we have the following *wlp* rule for method calls:

- $wlp(\texttt{call } \vec{z} \texttt{ := p(}\vec{e}\texttt{)}, A) = wlp(\texttt{assert pre [}\vec{e}\texttt{/}\vec{x}\texttt{]; } \vec{o} \texttt{ := } \vec{g}\texttt{; havoc } \vec{z}\texttt{;}$ $\texttt{assume post [}\vec{e}\texttt{[}\vec{o}\texttt{/}\vec{g}\texttt{]/}\vec{x}\texttt{][}\vec{z}\texttt{/}\vec{y}\texttt{][}\vec{o}\texttt{/old(}\vec{g}\texttt{)], } A)$

$\vec{o}$ are fresh local variables and `p` is declared as `procedure p(`$\vec{x}$`) returns (`$\vec{y}$`) requires pre; ensures post; modifies` $\vec{g}$. We assume that the variables $\vec{z}$ do not occur in paramters $\vec{e}$.

8.3 Maps, Arrays and Heaps

8.3.1 Maps: Maps are built-in in Boogie and are declared with a domain and a range. Maps are immutable, updating a map defines a new map. They are not extensional.

```
var c: [int] bool;
var d: [int, int] int;
var a: [int] int;
var b: [int] int;
var i: int;
i := a[3];            //map lookup
b := a[3 := 0];       //map update into b
a := a[4 := 2];       //map update
a[4] := 2             //map update
```

Code 7: Maps in Boogie.

8.3.2 Polymorphic maps: Type synonyms declare a name for a particular type, like we saw with `IntPair` above. Combined with polymorphic types this allows for recursive types.

8.3.3 Sequences: We can use maps to model sequences.

8.3.4 Object-based heap: We can also use maps to model an object-based heap with object allocation using an alloc field. Richer heap notions usually require a higher-level language, as framing quickly becomes brittle.

```
type Key T;
function makeKey <T> (v: T) returns (Key T);

type PolyMap = <S> [Key S] S;

procedure test(n: PolyMap) {
    var m: PolyMap;
    var k: Key PolyMap;
    m := n[makeKey(1) := 3];
    k := makeKey(m);
    m[k] := m;
    m[k] := m;
    assert m[k][k][makeKey(1)] == 3;
}
```

Code 8: Polymorphic maps in Boogie.

```
type IntSeq;
function elems(s:IntSeq) : [int]int;
function length(s:IntSeq) : int;

var m:IntSeq;
var n:IntSeq;

procedure n_reverse_m();
    modifies n;
    ensures length(n)==length(m);
    ensures (forall i: int :: {elems(n)[i]}
        0 <= i && i < length(n) ==>
            elems(n)[i] == elems(m)[length(n) - i - 1]);

procedure test()
    modifies m, n;
{
    call n_reverse_m();
    m := n; call n_reverse_m();
    assert (forall i: int ::
        0 <= i && i < length(n) ==>
            elems(n)[i] == old(elems(m))[i]);
}
```

Code 9: Modeling sequences in Boogie.

```
type Ref; // references
type Field T; // fields
type ObjectHeap = <T> [Ref, Field T] T;

var Heap : ObjectHeap;
const alloc : Field bool;

procedure test(x: Ref)
    requires Heap[x,alloc];
    modifies Heap;
{
    var y: Ref;
    // model allocating a fresh object
    havoc y;
    assume !Heap[y, alloc];
    Heap[y, alloc] := true;

    assert x != y; // succeeds
}
```

Code 10: Modeling a heap in Boogie.

9 Heap Reasoning and Permission Logics

9.1 Implicit Dynamic Frames

9.1.1 Adding heap support: We want to add heap support to our language. We will consider an object-based heap and will not use global variables from now on. There is a wide variety of formal approaches for heap-based reasoning. We will focus on permission-based reasoning with a logic called Implicit Dynamic Frames (IDF).

9.1.2 Access permissions: Each field location in the heap has an associated access permission. For field f of an object x, the associated access permission is denoted `acc(`$x.f$`)`. A field update $x.f$ `:= e` is only permitted when the permission `acc(`$x.f$`)` is held. Access permissions can also be split into fractional permissions. The full permission `acc(`$x.f$`, 1)` is equal to `acc(`$x.f$`)` and is needed to write to the field. For a read, one must only hold some fractional permission. There are no permissions to fields of null, `acc(`$x.f$`)` entails x `!= null`

9.1.3 Pure assertion: An assertion without accessibility predicates is called a pure assertion.

9.1.4 IDF assertions: A, B `::=` e `|` `acc(`$e.f$`,` p`)` `|` $A \wedge B$ `|` $A * B$ `|` $e \Rightarrow A$

- e: is a first-order formula
- `acc(`$e.f$`,` p`)`: where e is of type `Ref` and $e.f$ denotes a heap location, not a value.
- $A \wedge B$: is the normal conjunction.
- $A * B$: is the separating conjunction. $A * B$ is true if A and B are true and the sum of the permissions is held.
- $e \Rightarrow A$: here we cannot have `acc(`$e.f$`,` p`)` on the left-hand side because we can never "negate" permissions in Viper. If we allowed this we could do `acc(`$e.f$`,` p`)` $\Rightarrow \bot$ with which we take away a permission.

9.1.5 State: A state is now a triple (H, P, σ) with heap H which maps (object, field name)-pairs to values, permission mask P which maps (object, field name)-pairs to rationals within $[0, 1]$ and the environment σ which maps variables to values.

9.1.6 Trueness of IDF assertions: An IDF assertion A is true in state (H, P, σ), written $H, P, \sigma \models A$ as defined as follows:

- $H, P, \sigma \models e$ iff $\lceil e \rfloor_H, \sigma = \text{true}$
- $H, P, \sigma \models$ `acc(`$x.f$`)` iff $P[\lceil e \rfloor_H, \sigma,\ f] \geq \lceil p \rfloor_H, \sigma$
- $H, P, \sigma \models A \wedge B$ iff $H, P, \sigma \models A$ and $H, P, \sigma \models B$
- $H, P, \sigma \models A * B$ iff $\exists P_1, P_2 :\ P = P_1 \uplus P_2$ and $H, P, \sigma \models A$ and $H, P, \sigma \models B$
- $H, P, \sigma \models e \Rightarrow A$ iff $(\lceil e \rfloor_H, \sigma = \text{true}) \Rightarrow H, P, \sigma \models A$

9.2 Framing and Permissions

9.2.1 Framing: We define $H, P, \sigma \models_{frm} e$ (read H, P, σ frames e) iff the evaluation of e in the state only depends on the field reads for which permission is held in P. We can generalize this definition to $H, P, \sigma \models_{frm} A$. These are called well-definedness conditions. They are checked for all expressions that are evaluated.

9.2.2 Required permissions: The permissions required by A in state (H, P, σ), written `Perms(`A`)`$_{(H,\sigma)}$ are defined as follows:

- `Perms(`e`)`$_{(H,\sigma)} = \emptyset$
- `Perms(acc(`x`.`f`))`$_{(H,\sigma)} = \emptyset[(\lceil e \rfloor_H, \sigma, f) \to \lceil p \rfloor_H, \sigma]$
- `Perms(`$A \wedge B$`)`$_{(H,\sigma)} = max($`Perms(`A`)`$_{(H,\sigma)}$, `Perms(`B`)`$_{(H,\sigma)})$
- `Perms(`$A * B$`)`$_{(H,\sigma)} =$ `Perms(`A`)`$_{(H,\sigma)} \uplus$ `Perms(`B`)`$_{(H,\sigma)}$
- `Perms(`$e \Rightarrow A$`)`$_{(H,\sigma)} =$ `Perms(`A`)`$_{(H,\sigma)}$ if $\lceil e \rfloor_H, \sigma =$ true or $\emptyset$ if $\lceil e \rfloor_H, \sigma =$ false

9.2.3 Self-framing: An assertion is self-framing (written $\models_{frm} A$) iff $\forall H, P, \sigma((H, P, \sigma \models A) \Rightarrow (H, P, \sigma \models_{frm} A))$

9.3 New Statements

9.3.1 Inhale and exhale: We introduce two new statements:

9.3.1.1 Inhale statement: `inhale` A adds the permissions required by A and assumes pure assertions made in A.

9.3.1.2 Exhale statement: `exhale` A checks that A is true and removes the permissions required by A. It indirectly havocs all locations to which we no longer have any permission.

9.3.2 Label statement: We can use a `label` l in a normal statement position. A labelled `old`-expression `old[`l`](`e`)` evaluates e in the heap as if it was at the label l statement.

9.3.3 Encoding of method calls: We can now encode method calls with `inhale` and `exhale` statements and labels: $\vec{z}$ `:=` $m(\vec{e})$ becomes `label` l`; exhale pre[`$\vec{e}/\vec{x}$`]; havoc` $\vec{z}$`; inhale post[`$\vec{e}/\vec{x}$`][`$\vec{z}/\vec{y}$`][old[`l`]/old]`. The method pre- and postconditions must be self-framing assertions.

9.3.4 Eliminating loops: We can also eliminate loops: `while (`b`) invariant` A_I $\{s\}$ becomes `exhale` A_I`; havoc` $\vec{x}$`; ((inhale` $A_I \wedge b$`;` s`; exhale` A_I`; assume false)[](inhale` $A_I \wedge \neg b$`))`. Loop invariants must be self-framing assertions.

10 Unbounded Heap Data

10.0.1 How to handle unbounded heap data: There are two main techniques for handling unbounded data structures in IDF: recursive definitions for predicates and functions or quantified permissions.

10.1 Recursive Definitions

10.1.1 Predicate: A predicate declaration consists of a name, any number of formal parameters and a body, which is a self-framing assertion.

10.1.2 Recursive predicates: The predicate's body can include instances of the predicate being declared. These predicate definitions are interpreted as least-fixed points. All instances of the predicate have finite unfolding. We treat predicate definitions iso-recursively. Thus a predicate instance is not treated as identical to the corresponding body, but the two can be explicitly exchanged via extra statements.

10.1.3 Extra statements: We add two extra statements to handle the predicate exchange explicitly:

10.1.3.1 `fold` statement: The `fold` statement exchanges a predicate body for a predicate instance.

10.1.3.2 `unfold` statement: The `unfold` statement exchanges a predicate instance for its body.

10.1.3.3 `unfolding` expression: The `unfolding` expression temporarily unfolds a predicate during the evaluation of an expression.

10.1.4 Function: A function declaration consists of a name, declared formal parameters and a return-type, a body which is an expression, a self-framing precondition and optionally a pure postcondition. Functions are side-effect free, so all access in the precondition is still there after the function.

10.1.5 Recursive definitions: Recursive definitions are natural for implementations which perform top-down, recursive traversals. They can easily provide built-in acyclicity and tree-like guarantees. But we need additional statements and they are not useful for random access data or structures with complex sharing.

10.2 Quantified Permissions

10.2.1 Quantified permissions: We allow permissions under a quantifier like `forall` and `exists`. In general, `acc(`e`.`f`)` can occur under a `forall` x`:` T quantifier if the mapping from instantiations of x to instantiations of e is injective. Just as for pure quantifiers, quantified permissions must be appropriately instantiated. Thus selecting a good trigger is important. This approach works well for random-access situations or structures with cycles or sharing.

```
field next : Ref
field val : Int

predicate positive_node(n:Ref) {
    acc(n.val) && acc(n.next) && n.val >= 0
}

predicate list(start: Ref) {
    acc(start.val) && acc(start.next)
        && (start.next != null ==> list(start.next))
}

function elems(start: Ref) : Seq[Int]
    requires list(start)
{
    unfolding list(start) in (
        (start.next == null
            ? Seq(start.val)
            : Seq(start.val) ++ elems(start.next)))
}

method append(l1: Ref, l2: Ref)
    requires list(l1) && list(l2) && l2 != null
    ensures list(l1) && elems(l1) == old(elems(l1) ++ elems(l2))
{
    unfold list(l1)
    if(l1.next == null) {
        l1.next := l2
    } else {
        append(l1.next, l2)
    }
    fold list(l1)
}
```

Code 11: Modeling a linked list in Viper.

```
field val: Int
domain Array {
    function loc(a: Array, i: Int): Ref
    function len(a: Array): Int
    function first(r: Ref): Array
    function second(r: Ref): Int
    axiom injectivity {
        forall a: Array, i: Int :: {loc(a, i)}
            first(loc(a, i)) == a && second(loc(a, i)) == i
    }
    axiom length_nonneg {
        forall a: Array :: len(a) >= 0
    }
}

method incrementAll(a:Array)
    requires forall i: Int :: 0 <= i && i < len(a)
        ==> acc(loc(a,i).val)
    ensures forall i: Int :: 0 <= i && i < len(a)
        ==> acc(loc(a, i).val)
        && loc(a, i).val == old(loc(a, i).val) + 1
{
    var j: Int := 0
    while(j < len(a))
        invariant forall i: Int :: 0 <= i && i < len(a)
            ==> acc(loc(a, i).val) && loc(a, i).val
            == old(loc(a, i).val) + (i < j ? 1 : 0)
        invariant 0 <= j && j <= len(a)
    {
        loc(a, z).val := loc(a, j).val + 1
        j := j + 1
    }
}
```

Code 12: Encoding of arrays in Viper.

11 Advanced Specification and Verification

11.1 Frequent Scenarios

11.1.1 Abstraction: Abstract predicates, functions or methods consist of a name and signature/specification but no body. They are used to abstract over concrete definitions in specifications. This enables information hiding, even for bounded data structures.

11.1.2 Degenerate specification: Writing procedure specifications without implementing them runs the risk that implementations turn out to be awkward or impossible. A procedure might not be implementable simply because the task is impossible, or because e.g. the precondition allows for unintended initial states, so called degenerate cases.

```
field next : Ref
field val : Int

predicate lseg(start: Ref, end: Ref) {
    acc(start.val) && acc(start.next)
        && (start.next != end ==> lseg(start.next, end))
}

function lsegelems(start: Ref, end: Ref) : Seq[Int]
  requires lseg(start, end)
{
  unfolding lseg(start, end) in (
    (start.next == end
      ? Seq(start.val)
      : Seq(start.val) ++ lsegelems(start.next, end)))
}

method addAtEnd(l1: Ref, l2: Ref)
    requires lseg(l1, l2) && acc(l2.val) && acc(l2.next)
    ensures lseg(l1, old(l2.next))
        && lsegelems(l1, old(l2.next))
        == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))

method addAtEndFixed(l1: Ref, l2: Ref)
    requires lseg(l1, l2) && acc(l2.val) && acc(l2.next)
        && acc(l2.next.next, 1/2)
    ensures lseg(l1, old(l2.next))
        && lsegelems(l1, old(l2.next))
        == old(lsegelems(l1, l2)) ++ Seq(old(l2.val))
    ensures acc(old(l2.next).next, 1/2)
```

Code 13: Degenerate case example. Here the precondition of `addAtEnd` permits the possibility that `l1.next == l2.next` and with that initial state the postcondition is unsatisfiable. `addAtEndFixed` fixes this problem by making sure `l1.next` != `l2.next`.

11.1.3 Inconsistency due to recursion: Functions in Viper can potentially introduce inconsistency. Function postconditions can also cause inconsistencies. Predicate definitions do not have the same potential problems

```
function bad(x: Int) : Int {
    1 + bad(x)
}
method test(x: Int)
    requires bad(x) == 42
{
    assert false
    // this works, because bad is infinite
    // and we know nothing of the value
}

function still_bad(x: Int) : Int
    ensures false
{
    still_bad(x)
}
method test_more(x: Int)
    requires bad(x) == 42
{
    assert false
    // this works, because the verifier assumes
    // the postcondition of still_bad
}

predicate ok(i: Int) {
    ok(i+1)
}
predicate still_ok(x: Int) {
    still_ok(x)
}
method test(x: Int)
    requires ok(x) && still_ok(x) // this precondition is false!
{
    unfold ok(x);
    // assert false // fails , inconsistency not detected
    unfold still_ok(x);
    assert false // Carbon succeeds , Silicon doesn t detect it
}
```

Code 14: Bad functions and good predicates.

11.2 Auxiliary State

11.2.1 Auxiliary state: It is common that we need state in the program just for verification purposes. This additional state is called auxiliary state. There are two main variants:

```
method setToArray(vals: Set[Int]) returns (a:Array, map:IntMap)
    ensures len(a) == |vals|
    ensures forall i:Int :: 0 <= i && i < len(a)
        ==> acc(loc(a, i).val)
    ensures forall i:Int :: {i in vals} i in vals
        ==> let k == (select(map,i)) in 0 <= k && k < len(a)
        && loc(a,k).val == i
{
    // model allocating an array of size |vals|
    a := havocArray()
    assume len(a) == |vals|
    inhale forall i: Int :: 0 <= i && i < len(a)
        ==> acc(loc(a,i).val)
    var s  Set[Int] := vals
    var element: Int;
    var j: Int := 0;
    while (|s| > 0)
        invariant forall i: Int :: 0 <= i && i < len(a)
            ==> acc(loc(a,i).val)
        invariant s subset vals && j == |vals setminus s|
        invariant forall i:Int :: {i in vals}
            i in (vals setminus s)
            ==> exists k: Int :: 0 <= k && k < j
            && loc(a,k).val == i
    {
        element := havocInt() // simulate a havoced Int value
        assume element in s // we have *some* element of s
        loc(a, j).val := element
        s := s setminus Set(element)
        j := j + 1
    }
}
```

Code 15: Writing out a set to an array in Viper.

11.2.1.1 Ghost state: Ghost state is auxiliary state which is manually maintained and updated.

11.2.1.2 Model state: Model state is auxiliary state which is automatically maintained and updated by the verifier.

```
field next : Ref
field val : Int

predicate listelems(start : Ref, els: Seq[Int])
{
    acc(start.val) && acc(start.next)
    && |els| > 0 && els[0] == start.val
    && (start.next == null
    ?  els == Seq(start.val)
    : listelems(start.next, els[1..]))
}

method appendelems(l1 : Ref, l2: Ref,
        l1elems : Seq[Int], l2elems : Seq[Int])
    requires listelems(l1, l1elems) && listelems(l2, l2elems)
        && l2 != null
    ensures listelems(l1, l1elems ++ l2elems)
{
    unfold listelems(l1, l1elems)
    if(l1.next == null) {
        l1.next := l2
    } else {
        appendelems(l1.next, l2, l1elems[1..], l2elems)
    }
    assert (l1elems ++ l2elems)[1..] == (l1elems[1..] ++ l2elems)
    fold listelems(l1, l1elems ++ l2elems)
}
```

Code 16: Example code where `l1elems` and `l2elems` are ghost state. For an example on model state, see Code 11, where `elems` is model state.