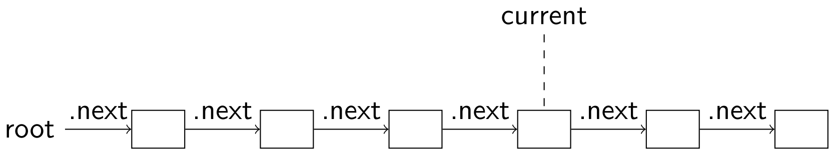
Lecturecise 15: Weak monadic second-order theory of one successor (WS1S)

Reachability in the Heap

Many programs manipulate linked data structures (lists, trees). To express many important properties of such programs we need a logic that supports **transitive closure**.

Example: Consider a linked list



As part of verifying code that manipulates linked lists, we need to say that the object *current* is reachable from the *root* of the linked list. We can write this reachability condition as follows:

$$(\text{node1}, \text{node1}) \in \{(x, y) \mid \; x \neq \mathit{null} \; \land \; \mathit{next}(x) = y\}^*$$

Transitive closure of a relation

Reachability condition:

$$(\mathrm{node1}, \mathrm{node1}) \in \{(x, y) \mid \; x \neq \mathit{null} \;\wedge\; \mathit{next}(x) = y\}^*$$

Define the shorthand

$$\mathrm{Closed}(S, F) = (\forall x, y. x \wedge y \wedge x \in S \wedge F(x, y) \rightarrow y \in S)$$

Define reflexive transitive closure $(u, v) \in \{(x, y) \mid F(x, y)\}^*$ by

$$\forall S. u \in S \wedge \mathrm{Closed}(S, F) \rightarrow v \in S$$

More Data Structure Correctness Properties

- Data structure has no cycles
- The set of objects in the data structure does not change when we balance a binary search tree
- When we insert an object into a data structure, the set of reachable objects is increased by the given object
- Correctness of removal from doubly linked list

A Related Use: Non-Interference

Prove that two parts of program will touch disjoint heap locations

- ensure that properties of an object on heap hold after a procedure call
- prove that the object is not reachable from procedure parameters and globals
- prove that if an object is reachable in one procedure, it is not reachable in another

Application in e.g. automated parallelization, Scala actor programs:

- to ensure absence of data races, actors should not access same region of state
- again need to show that if an object is reachable in one actor, it is not reachable in another actor

WS1S as a Logic for Reachability

Questions we aim to answer:

- What is the syntax and semantics of logics that we can use to express such conditions?
- How do we prove verification conditions in such logic?

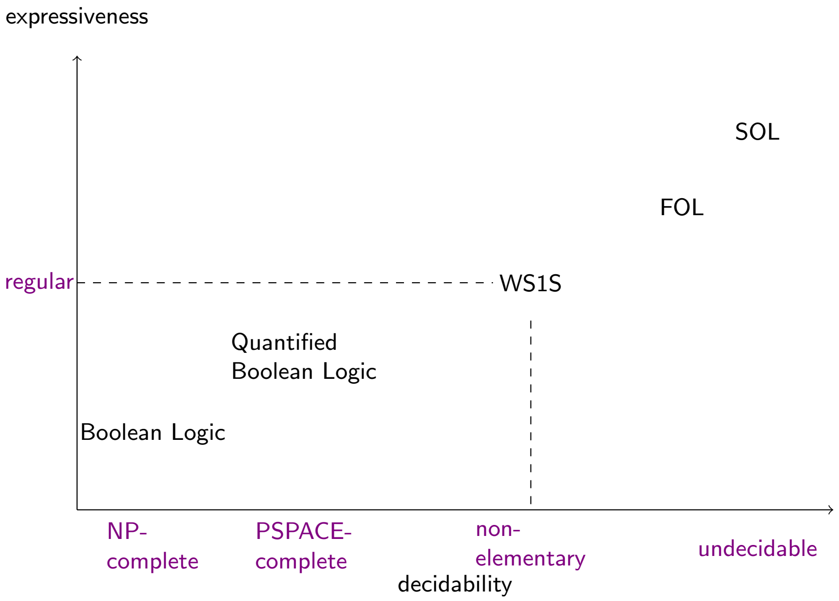
We look at one such logic: Weak Monadic Second-Order Logic of One Successor (WS1S). We show its decidability by using automata to describe the set of models of WS1S formulas.

Weak Monadic Logic of One Successor

Explanation of the name:

- second-order logic: we can quantify not only over elements but also over **sets** and relations
- monadic: we cannot quantify over relations of arity two or more, just over unary relations (**sets**)
- weak: the sets we quantify over are **finite**
- of one successor: the domain is an infinite chain, where each element has one successor (we only have successor and equality)

Overview


expressiveness
SOL
FOL
regular
WS1S
Quantified Boolean Logic
Boolean Logic
NP-complete
PSPACE-complete
non-elementary
undecidable
decidability

Syntax of $WS1S_0$

Minimalistic syntax

$$F ::= v \subseteq v \mid succ(v, v) \mid F \vee F \mid \neg F \mid \exists v.F$$

- only sets, no first-order variables
- quantification only over finite sets

Semantics

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ denote non-negative integers and D the set of all *finite subsets* of $\mathbb{N}$ (i.e. $2^{\mathbb{N}}$).

We consider the set of interpretations (D, α) where

- for each variable v we have $\alpha(v) \in D$
- $\subseteq$ is the subset relation

$$\alpha(\subseteq) = \{(S_1, S_2) \mid S_1 \subseteq S_2\}$$

- the relation $succ(v_1, v_2)$ is the successor relation on integers lifted to singleton sets:

$$\alpha(succ) = \{(\{k\}, \{k+1\}) \mid k \in \mathbb{N}\}$$

The meaning of formulas is given by standard First-order logic semantics.

Set operations (as an exercise)

Quantification over sets with $\subseteq$ gives us the full Boolean algebra of sets.

- Two sets are equal: $(S_1 = S_2) = (S_1 \subseteq S_2) \land (S_2 \subseteq S_1)$
- Strict subset: $(S_1 \subset S_2) = (S_1 \subseteq S_2) \land \neg(S_2 \subseteq S_1)$
- Set is empty: $(S = \emptyset) = \forall S_1 . S \subseteq S_1$
- Set S is singleton (has exactly 1 element): $One(S) =$
- Set membership, treating elements as singletons: $(x \in S) =$
- Intersection: $(A = B \cap C) =$
- Union: $(A = B \cup C) =$
- Set difference: $(A = B \setminus C) =$
- If k is a fixed constant, properties $\mathrm{card}(A) \geq k$, $\mathrm{card}(A) \leq k$, $\mathrm{card}(A) = k$

Set operations

The ideas is that quantification over sets with $\subseteq$ gives us the full Boolean algebra of sets.

- Two sets are equal: $(S_1 = S_2) = (S_1 \subseteq S_2) \wedge (S_2 \subseteq S_1)$
- Strict subset: $(S_1 \subset S_2) = (S_1 \subseteq S_2) \wedge \neg(S_2 \subseteq S_1)$
- Set is empty: $(S = \emptyset) = \forall S_1 . S \subseteq S_1$
- Set S is singleton (has exactly 1 element):
  $One(S) = (\neg(S = \emptyset)) \wedge (\forall S_1 . S_1 \subset S \rightarrow S_1 = \emptyset)$
- Set membership, treating elements as singletons: $(x \in S) = (x \subseteq S)$
- Intersection:
  $(A = B \cap C) = (A \subseteq B \wedge A \subseteq C) \wedge (\forall A_1 . A_1 \subseteq B \wedge A_1 \subseteq C \rightarrow A_1 \subseteq A)$
- Union:
  $(A = B \cup C) = (B \subseteq A \wedge C \subseteq A) \wedge (\forall A_1 . B \subseteq A_1 \wedge C \subseteq A_1 \rightarrow A \subseteq A_1)$
- Set difference: $(A = B \setminus C) = (A \cup (B \cap C) = B \wedge A \cap C = \emptyset)$ (or just use element-wise definitions with singletons)
- If k is a fixed constant, properties card$(A) \geq k$, card$(A) \leq k$, card$(A) = k$

Transitive closure of a relation

The relation Closed becomes

$$\text{Closed}(S, F) = (\forall x, y. One(x) \land One(y) \land x \in S \land F(x, y) \rightarrow y \in S)$$

If $F(x, y)$ is a formula on singletons, we can define reflexive transitive closure, $(u, v) \in \{(x, y) \mid F(x, y)\}^*$ as before by

$$\forall S. u \in S \land \text{Closed}(S, F) \rightarrow v \in S$$

Thus, we can express

$$(current, root) \in \{(y, x) \mid \; x \neq null \; \land \; next(x) = y\}^*$$

in WS1S.
(To express structures with multiple acyclic lists and 'null', we introduce the set 'Nulls' denoting natural numbers that are considered null.)

Using transitive closure and successors

- Constant zero: $(x = 0) = One(x) \wedge \neg(\exists y. One(y) \wedge succ(y, x))$

- Addition by constant: $(x = y + c) =$
  $(\exists y_1, \ldots, y_{c-1}. succ(y, y_1) \wedge succ(y_1, y_2) \wedge \ldots \wedge succ(y_{c-1}, x))$

- Ordering on positions in the string:
  $(u \leq v) = ((u, v) \in \{(x, y) \mid succ(x, y))\}^*$

- Reachability in k-increments, that is, $\exists k \geq 0. y = x + c \cdot k$:
  $\text{Reach}_c(u, v) = ((u, v) \in \{(x, y) \mid y = x + c\}^*)$

- Congruence modulo c:
  $(x \equiv y)(\text{mod } c) = \text{Reach}_c(x, y) \vee \text{Reach}_c(y, x)$

Representing integers

- although we interpret elements as sets of integers, we cannot even talk about addition of two arbitrary integers $x = y + z$, only addition with a constant
- although we can say $\mathrm{Reach}_c(x, y)$ we cannot say in how many steps we reach y from x.

However, if we view sets of integers digits of a binary representation of another integer, then we can express much more. If S is a finite set, let $N(S)$ represent the number whose digits are S, that is:

$$N(S) = \sum_{i \in S} 2^i$$

Representing integers

$$N(S) = \sum_{i \in S} 2^i$$

Then we can define addition $N(Z) = N(X) + N(Y)$ by saying that there exists a set of carry bits C such that the rules for binary addition hold:

$$\begin{array}{rrcl} \exists C.\ 0 \notin C\ \wedge \forall i.\ \big(& ((i \in Z) & \leftrightarrow & ((i \in X) \oplus (i \in Y) \oplus (i \in C)) \wedge \\ & ((i+1 \in C) & \leftrightarrow & \mathrm{atLeastTwo}(i \in X, i \in Y, i \in C)\big) \end{array}$$

where

$$\begin{array}{rcl} x \oplus y & = & (x \vee y) \wedge \neg(x \wedge y) \\ \mathrm{atLeastTwo}(x, y, z) & = & (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \end{array}$$

This way we can represent entire Presburger arithmetic in MSOL over strings.

Expressive power

- We can represent entire Presburger arithmetic in MSOL over strings

  actually, more expressive power because $X \subseteq Y$ means that the one bits of $N(X)$ are included in the bits of $N(Y)$, that is, the bitwise or of $N(X)$ and $N(Y)$ is equal to $N(Y)$

  In fact, if we add the relation of bit inclusion into Presburger arithmetic, we obtain precisely the expressive power of MSOL when sets are treated as binary representations of integers. Indeed, taking the minimal syntax of MSOL from the beginning, the bit inclusion gives us the subset, whereas the successor relation $s(x, y)$ is expressible using $y = x + 1$.

Definable Relations

We can define relations on $\mathbb{N}$ in two different ways.
Relations on singleton sets:

$$r_F^s = \{(p, q) \mid F(\{p\}, \{q\})\}$$

Relations on binary representations:

$$r_F^b = \{(p, q) \mid F(N(p), N(q))\}$$

Addition is not definable as some r_F^s, but it is definable as r_F^b.

Decision procedure

Theorem (Büchi theorem, [Büchi, 1960] and [Elgot, 1961])
A language $\mathcal{L} \subseteq \Sigma^*$ is regular $\Leftrightarrow$ it is expressible in weak monadic second-order logic on words.

Decision procedure

Theorem (Büchi theorem, [Büchi, 1960] and [Elgot, 1961])
A language $\mathcal{L} \subseteq \Sigma^*$ is regular $\Leftrightarrow$ it is expressible in weak monadic second-order logic on words.

Given WS1S sentence ϕ, check for satisfiability:

- convert ϕ to $WS1S_0$ formula ϕ'
- construct automaton $\mathcal{A}_{\phi'}$ such that $\mathcal{L}(A_{\phi'}) = \mathcal{L}(\phi') = \mathcal{L}(\phi)$
- check whether $\mathcal{L}(\mathcal{A}_{\phi'}) = \emptyset$
- if not, ϕ is satisfiable, else, ϕ is unsatisfiable.

MSOL on words

Given a formula $\phi(X_1, \ldots, X_n)$, with free variables $X_1, \ldots, X_n$, an interpretation α is a finite string in $(\{0,1\})^*$:

- X_1 is given on the first row, X_2 on the second, ...
- for a string $c_0, \ldots, c_p, ..., c_n$ in row i

$$p \in X_i \Leftrightarrow c_i = 1$$

Example:

$$\phi(X, Y) : \forall P.One(P) \rightarrow (P \subseteq X \leftrightarrow (\exists Q.One(Q) \land succ(P, Q) \land Q \subseteq Y))$$

$\alpha(X) = \{0, 3, 4, 5\}$ and $\alpha(Y) = \{1, 4, 5, 6\}$ is given by

$$w = \left| \begin{array}{ccccccc} 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array} \right|$$

Using Automata to Decide WS1S

Consider a formula F of WS1S. Let V be a finite set of all variables in F. We construct an automaton $\mathcal{A}(F)$ in the finite alphabet

$$\Sigma = \Sigma_1^V$$

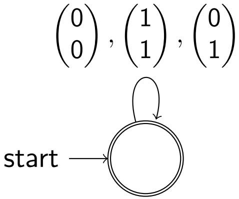
for $\Sigma_1 = \{0, 1\}$ such that the following property holds:
for every $w \in \Sigma^*$,

$$w \in \mathcal{L}(\mathcal{A}(F)) \quad \Longleftrightarrow \quad w \models F$$

Automaton construction

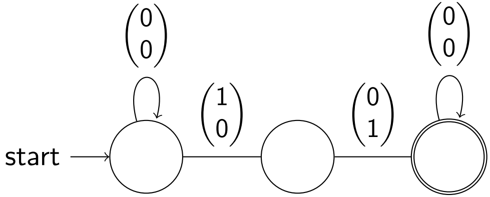
We define the automaton by recursion on the structure of formula.

Case $\mathcal{A}(x \subseteq y)$: Automaton for regular expression $[x \to y]^*$



Automaton construction cont.

Case $\mathcal{A}(succ(x, y))$: Automaton for regular expression
$[\neg x \land \neg y]^*[x \land \neg y][\neg x \land y][\neg x \land \neg y]^*$



Automaton construction cont.

Case $\mathcal{A}(F_1 \vee F_2)$: Automaton for union of regular languages, $\mathcal{A}(F_1) \vee \mathcal{A}(F_2)$

Case $\mathcal{A}(\neg F)$: Complement $\neg\mathcal{A}(F)$ (by flipping accepting and non-accepting states).

Existential Quantification $\mathcal{A}(\exists x.F)$

$$w \models \exists x.F \text{ iff } \exists b.patch(w, x, b) \models F$$

Let $w = w_1 \dots w_n$ where $w_i \in \Sigma$ and $b = b_1 \dots b_m$ where $b_j \in \Sigma_1$.
Let $N = \max(n, m)$. Define $patch(w, x, b) = p_1 \dots p_N$ where $p_i \in \Sigma$ such that

$$p_i(v) = \begin{cases} w_i(v), & \text{if } v \neq x \wedge i \leq n \\ 0, & \text{if } v \neq x \wedge i > n \\ b_i, & \text{if } v = x \wedge i \leq m \\ 0, & \text{if } v = x \wedge i > m \end{cases}$$

```
.......
.......    0
.......
bbbbbbbbbbbbbb
.......
.......    0
.......
-------
   w
--------------
 patch(w,x,b)
```

Existential Quantification $\mathcal{A}(\exists x.F)$

We need that for every word,

$$w \in \mathcal{L}(\mathcal{A}(\exists x.F))$$

iff

$$\exists s \in \Sigma_1^*.\ patch(w, x, s) \in \mathcal{L}(\mathcal{A}(F))$$

Given a deterministic automaton $\mathcal{A}(F)$, we can construct a deterministic automaton accepting $\{w \mid \exists s \in \Sigma_1^*.\ patch(w, x, s) \in \mathcal{L}(\mathcal{A}(F))\}$ in two steps:

- take the same initial state
- for each transition $\delta(q, w)$ introduce transitions $\delta(q, w[x := b])$ for all $b \in \Sigma_1$
- initially set final states as in the original automaton
- if q is a final state and $zero_x \in \Sigma$ is such that $zero_x(v) = 0$ for all $x \neq v$, and if $\delta(q', zero_x) = q$, then set q' also to be final

Examples

Implementation in the MONA tool: `http://www.brics.dk/mona/`

Example 1
Compute automaton for formula $\exists X . \neg(X \subseteq Y)$.
MONA syntax:

```
var2 Y;
ex2 X: ~(X sub Y);
```

Example 2
Use the rules to compute (and minimize) the automaton for

$$\neg((\neg(X \subseteq Y)) \wedge (\neg(Y \subseteq X)))$$

Complexity

The construction determinizes the automaton whenever it needs to perform negation. Moreover, existential quantifier forces the automaton to be non-deterministic. Therefore, with every alternation between $\exists$ and $\forall$ we obtain an exponential blowup. For formula with n alternations we have $2^{2^{\cdots 2^{n}}}$ complexity with a stack of exponentials of height n. Is there a better algorithm?

Lower Bound The following paper shows that, in the worst case such behavior cannot be avoided, because of such high expressive power of MSOL over strings.
Cosmological Lower Bound on the Circuit Complexity of a Small Problem in Logic, L.Stockmeyer and A.R. Meyer, 2002. (See the introduction and the conclusion sections)
`http://theory.csail.mit.edu/~meyer/stock-circuit-jacm.pdf`

Example

Compute the automaton for the formula $\exists Y.(X < Y)$ where $<$ is interpreted treating X, Y as digits of natural numbers. Also compute the automaton for the formula $\exists X.(X < Y)$.
Define the less-than relation in MONA and encode this example.