We would like to use automata on \( \omega \)-words for model checking. Currently, the only type of machine we have available is a Büchi automaton. It is not hard to show that Büchi automata behave well with respect to union and intersection.

Needless to say, we also need complements, so it is natural to try to determinize Büchi automata.

Proposition
Let \( L = \{ x \in \{a,b\}^\omega \mid \#_b x < \infty \} \). Then \( L \) is recognizable but cannot be accepted by any deterministic Büchi automaton.

Proof. To see this, suppose there is some deterministic Büchi automaton that accepts \( L \).

Hence, for some \( n_1 \), \( \delta(q_0, ba^{n_1}) \in F \). Moreover, for some \( n_2 \), \( \delta(q_0, ba^{n_1}ba^{n_2}) \in F \). By induction we produce an infinite word

\[
ba^{n_1}ba^{n_2}ba^{n_3} \ldots
\]

accepted by the automaton. Contradiction.

This shows that a deterministic transition system together with a Büchi type acceptance condition \( \text{rec}(\pi) \cap F \neq \emptyset \) is not going to work: not only do we have to construct a deterministic transition system, we also have to modify our acceptance conditions. Alas, it is far from clear how one should do this.

So there are recognizable \( \omega \)-languages that cannot be accepted by any deterministic Büchi automaton. Alas, without determinization it is unclear how we could deal with complements, which we need to handle negation.

So, we need to find alternative machine models that allow for deterministic descriptions of recognizable languages.

As before, we will not change the transition system, just the acceptance condition.

**Key Idea:** Try to pin down \( \text{rec}(\pi) \) more carefully.

**Definition**
A Muller automaton consists of a deterministic transition system \( \langle Q, \Sigma, \tau \rangle \) and an acceptance condition \( q_0 \in Q \) and \( F \subseteq \mathcal{P}(Q) \).

\( \mathcal{A} \) accepts an infinite word \( x \in \Sigma^\omega \) if there is a run \( \pi \) of \( \mathcal{A} \) on \( x \) that starts at \( q_0 \) and such that \( \text{rec}(\pi) \in F \).

\( F \) is often referred to as the table of the Muller automaton. Note that the table may have size exponential in the size of the transition system.

But, complementation is easy (just as it was easy for DFAs): make sure the machine is complete, then replace the old table \( F \) by \( \mathcal{P}(Q) - F \).

But note that this simple operation might produce an exponential blow-up if done in a ham-fisted way. Lots of opportunity for clever hacks.
Example: Muller

The at-least-one-but-finitely-many-\(b\)'s language from above is accepted by the following Muller automaton.

The table has the form \(F = \{(2), \{3\}\} \).

![Muller automaton](image)

WTF? This automaton distinguishes between an even and odd number of \(b\)'s. This is a bit scary since the distinction is by no means obvious from the original Büchi automaton.

Muller and Closure

As we have seen, the family of languages accepted by Muller automata is closed under complementation. It is in fact an effective Boolean algebra.

Lemma

Given two Muller automata \(A_1\) and \(A_2\) one can construct a new Muller automaton \(A\) such that \(L(A) = L(A_1) \cap L(A_2)\).

Proof

As usual, we use a product machine \(A = A_1 \times A_2\).

The table of \(A\) has the form

\[
\{ F_1 \times F_2 \mid F_1 \in F_1, F_2 \in F_2 \}
\]

Since the machines are deterministic it is easy to see that this works. \(\square\)

The Complement

We can complement the table to get a machine for the complement of the language.

![Muller automaton complement](image)

The complement table contains several useless entries \(F\) (other than \(\emptyset\)):

<table>
<thead>
<tr>
<th>(F)</th>
<th>1</th>
<th>1,2</th>
<th>1,3</th>
<th>2,3</th>
<th>1,2,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L)</td>
<td>(a^*)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(a^{(ba^*)^2})</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

However, the two non-empty entries duly produce no \(b\)'s or infinitely many \(b\)'s, exactly the complement of the language.

Muller versus Deterministic Büchi

 Lemma

A language \(L \subseteq \Sigma^\omega\) is recognizable by a Muller automaton if, and only if, it is of the form

\[
L = \bigcup_{i \leq n} U_i - V_i
\]

where \(U_i, V_i \subseteq \Sigma^\omega\) are recognizable by a deterministic Büchi automaton. In other words, \(L\) must lie in the Boolean algebra generated by deterministic Büchi languages.

Proof

Suppose \(L\) is recognized by \(A\) with table \(F\). Since \(L = \bigcup_{F \in F} L(A(F))\) we only need to deal with tables of size \(1\). But

\[
L(A(F)) = \bigcap_{p \in F} L(A(p)) \cap \bigcup_{q \in F} L(A(q))
\]

where the automata on the right hand side are deterministic Büchi. Done by the closure properties of deterministic Büchi languages.

Proof, contd.

For the opposite direction assume \(U\) is accepted by a deterministic Büchi automaton \(A\). Define

\[
F = \{ P \subseteq Q \mid P \cap F \neq \emptyset \}
\]

Then the corresponding Muller automaton \(A(F)\) accepts \(U\).

But we know that Muller languages form a Boolean algebra, so we can get a Muller automaton for any Boolean combination \(\bigcup_{i \leq n} U_i - V_i\).

\(\square\)

Rabin Automata

Another possibility to modify acceptance conditions is to augment the positive condition of Büchi automata by a negative condition: a successful run must ultimately avoid a certain set of states.

Definition

A Rabin automaton consists of a deterministic transition system \((Q, \Sigma, \tau)\) and an acceptance condition \(q_0 \in Q\) and \(R \subseteq \mathcal{P}(Q) \times \mathcal{P}(Q)\).

\(A\) accepts an infinite word \(x \in \Sigma^\omega\) if there is a run \(\pi\) of \(A\) on \(x\) that starts at \(q_0\) and such that for some \((L, R) \in R\): \(\text{rec}(\pi) \cap L \neq \emptyset\) and \(\text{rec}(\pi) \cap R \neq \emptyset\).

The pairs \((L, R)\) are called Rabin pairs: \(L\) is the negative condition and \(R\) the positive condition.

In the special case where \(R = \{ (\emptyset, F) \}\) we are dealing with a deterministic Büchi automaton.
**Example: Rabin**

The Muller automaton from above can also be turned into a Rabin automaton with Rabin pairs

\[ \mathcal{R} = \{(1, 2, 3), (1, 3, 2)\} \]

The excluded sets force a tail end of the run to look like $2\omega$ or $3\omega$.

---

**Büchi, Muller, Rabin**

The acceptance condition for all three has the form

- initial states $I$ (a singleton for Muller and Rabin)
- a family $\mathcal{F} \subseteq \wp(Q)$ of permissible values for the recurrent state set of a run.

Note that we may safely assume that $\mathcal{F}$ contains only strongly connected sets.

For Büchi automata the family is trivial: $\mathcal{F} = \{\emptyset\}$ and thus a data structure of size $O(n)$.

For Muller automata it is explicitly specified and potentially large.

For Rabin automata the specification is implicit: all $X \subseteq Q$ such that $\exists (L,R) \in \mathcal{R} (X \cap L = \emptyset, X \cap R \neq \emptyset)$. Each Rabin pair is $O(n)$, but there may be exponentially many.

---

**Muller to/from Rabin**

**Lemma**

For every Rabin automaton there exists an equivalent Muller automaton, and conversely.

**Proof.**

Consider a Rabin automaton $\langle Q, \Sigma, \delta, g_0, \mathcal{R} \rangle$. We will use the same transition system and define a table

\[ \mathcal{F} = \{ X \subseteq Q \mid \exists (L,R) \in \mathcal{R} \ (X \cap L = \emptyset, X \cap R \neq \emptyset) \} \]

It is easy to see that $\langle Q, \Sigma, \tau, \mathcal{F} \rangle$ is an equivalent Muller automaton.

---

**Determinization**

The reason these equivalences are so important is the following theorem, see below for a (messy) proof.

**Theorem (Safra 1988)**

There is an algorithm to convert a Büchi automaton into an equivalent Rabin (or Muller) automaton.

The algorithm has running time $2^{O(n \log n)}$.

Unfortunately, this is optimal: there are examples where the deterministic automata are that large.

---

**Equivalence, contd.**

The opposite direction is harder.

Let $\langle Q, \Sigma, \delta, g_0, \mathcal{F} \rangle$ be a Muller automaton, say, $\mathcal{F} = \{F_1, \ldots, F_k\}$. Consider a new transition system on state set

\[ Q' = Q \times \Psi(F_1) \times \cdots \times \Psi(F_k) \]

and transitions

\[ (p, U_1, \ldots, U_k) \rightarrow (\delta(p, a), U'_1, \ldots, U'_k) \]

where $U'_i = \emptyset$ if $U_i = F_i$ and $F_i \cap (U_i \cup \{\delta(p, a)\})$ otherwise. The Rabin pairs are defined by

\[ L_i = \{ (p, U_1, \ldots, U_k) \mid p \notin F_i \} \quad R_i = \{ (p, U_1, \ldots, U_k) \mid U_i = F_i \} \]

One can verify that the new machine is equivalent to the given Muller automaton. \(\square\)
We now close the ring of equivalences by showing that every Büchi automaton has an equivalent Rabin automaton. Note the trade-off: the Rabin automaton must be deterministic, but it has a more flexible acceptance condition.

Theorem
For every Büchi automaton there exists an equivalent Rabin automaton. Hence the recognizable \( \omega \)-languages are closed under complementation.

One might suspect that this theorem is similar to the old Rabin/Scott result (powerautomaton construction). Alas, things don’t work out: we are not keeping track of individual computations (a tree of unbounded width), only of reachable sets of states. That introduces spurious computations in the infinite case.

For example, for the “infinitely many b’s” example from above we get

There is no way the power automaton on the right can be made to accept the right language, no matter whether we deal with Rabin or Muller automata: The second state 1,2 must be recurrent on any accepting run, but then there is a run accepting \( b^\omega \).

It seems we need a bigger hammer than Rabin-Scott.
The best way to organize the computation of the states of $B$ is to use ordered labeled trees, so-called Safra trees. Each node in a Safra tree carries three pieces of information. Assume that the Büchi automaton has $n$ states:

- **Name:** $v \in V = \{1, 2, \ldots, 2n\}$.
- **Label:** $\emptyset \neq \lambda(v) \subseteq Q$.
- **Mark:** a bit.

The names of all nodes in a tree are always distinct; $2n$ is a magic number that will be explained later. The root is always named 1. Only leaves can be marked. Since names are unique, we will occasionally confuse them with nodes.

In a sense, we will run the power automaton construction on all the nodes of the tree.
Proposition
A Safra tree has at most \( n \) nodes.

Proof. For every node \( v \) define
\[
f(v) \in \lambda(v) - \bigcup \lambda(u) \neq \emptyset
\]
Thus \( f(v) \) is a state that is present at \( v \), but missing from the subtrees of \( v \).
It is easy to see that \( f \) is injective. \( \square \)

Of course, the number of these trees is still wildly exponential: the only obvious bound is
\[
2^{O(n \log n)}
\]
This is uncomfortably large, but at least it’s finite: we can use Safra trees as states in the deterministic machine.

It remains to explain how to compute the transition function
\[
\delta(T, a) = T'
\]
where \( T \) and \( T' \) are Safra trees and \( a \in \Sigma \).

Batten down the hatches.

Suppose
\[
B = (Q, \Sigma, \tau; I, F)
\]
is an arbitrary Büchi automaton on \( n \) states.

We want to construct a (deterministic) Rabin automaton \( A \) whose states will be Safra trees over \( B \).

For each Safra tree \( T \) and letter \( a \in \Sigma \), we will explain in a moment how to construct a new Safra tree \( \delta_a(T) \).

Unmark
Unmark all the nodes in the tree.

Update
Replace \( \lambda(v) \) by \( \tau(\lambda(v), a) \) everywhere.

Create
If \( \lambda(v) \cap F \neq \emptyset \), attach a new rightmost child \( u \) to \( v \).
Set \( \lambda(u) = \lambda(v) \cap F \) and mark \( u \).

Horizontal Merge
Remove all states in \( \lambda(u) \) that appear in nodes \( v \) to the left of \( u \).

Kill Empty
Remove all nodes with empty label set.

Vertical Merge
Mark all states \( v \) such that \( \lambda(v) = \bigcup_{u \text{par} v} \lambda(u) \) and remove all descendants.
Let $T$ be an ordered tree (children are ordered left-to-right). We can introduce a partial order on the nodes as follows.

A node $v$ in $T$ is to the left of node $u$ if there is a subtree $T'$ of $T$ such that $T'$ has root $r$ and children $r_1, r_2, \ldots, r_k$ and there is $1 \leq i < j \leq k$ such that $v$ is in the subtree with root $r_i$ and $u$ is in the subtree with root $r_j$.

**Exercise**

Figure out a fast way of performing the Horizontal Merge in a Safra tree.

---

### Details

In the Create step, new names must be chosen from $V - \text{current nodes}$. In practice, the choice is always

\[ \text{new} = \min(V - \text{current nodes}) \]

This works fine since there can be at most $n$ nodes before step 3 and names are chosen in $V = [2n]$.

Also, we will traverse the tree in top-down, left-to-right order.

**Warning:** The node names are critical, we are not just dealing with trees of a certain shape. For example, the tree $(1 : P, 2 : R)$ is not the same as $(1 : P, 3 : R)$. The construction breaks without this distinction.

---

### More Details

Similarly, in Horizontal Merge, we arbitrarily have adopted the convention to move from left to right (top-down is not an issue here).

**Exercise**

Figure out what would happen in the following examples if we changed any of these conventions.

---

### The Whole Rabin Machine

The 6-step procedure defines (somewhat complicated) functions

\[ \delta_a : \text{Safra trees} \longrightarrow \text{Safra trees} \]

for each $a \in \Sigma$.

The Rabin machine $A$ is now simply defined as follows:

Run the vanilla closure algorithm starting at tree $T_0$ and with operations $\delta_a, a \in \Sigma$.

This produces a finite collection of Safra trees as state set $Q$ of $A$, plus the transition function of $A$ (the usual Cayley graph argument).

Of course, $T_0 \in Q$ is the initial state.

---

### Rabin Pairs

It remains to determine the Rabin pairs of $A$.

The pairs are $(L, R)$ where $v$ is (the name of) some node and

\[ L = \{ T \in Q \mid v \notin T \} \quad R = \{ T \in Q \mid v \in T, \text{marked} \} \]

Of course, we only need consider nodes $v \in V$ that appear marked in at least one tree.

That’s all.
Suppose the Büchi automaton has $Q = F$.

Then Safra’s algorithm degenerates into the ordinary Rabin-Scott powerset construction: all the trees have exactly one node, the root.

This is reassuring, since any infinite run is accepting in this case and the existence of an infinite run (without any additional conditions) can be tested by the power automaton.

Exercise

Make sure you understand how and why this works.

Lame Example 43

Example I 44

Let’s return to the old workhorse example

$L = \{ x \in \{ a, b \}^\omega | 1 \leq \#_b x < \infty \}$

of words containing at least one but only finitely many $b$’s. A Büchi automaton $B$ for $L$ looks like so:

The Rabin automaton $A$ has initial state $(1 : 1)$.

Computing All States 45

The Rabin pairs are

$\{ (1,2; 3), (1,3; 2) \}$

since 2 and 3 are the only marked nodes.

Some Steps 46

The diagram should look familiar:

The diagram should look familiar:

This is the machine we already saw previously.

In this particular case, we have already verified that the machine behaves properly.

Example II 48

Here is another Büchi automaton $B$ on alphabet $\{ a, b, c \}$.

This one is slightly more complicated.

The language is $a ((b + c)^*a + b)^\omega$
Example II, contd.

Rabin pairs \{ (0; 1, 4, 5), (1, 3, 4, 5; 2) \}

Comments

The Safra trees corresponding to the 5 states are

1. (1 : 1!)
2. (1 : 1, 2; 2 : 1!)
3. (1 : 2)
4. (1 : 1, 2!)
5. (1 : 2!)

The second Rabin pair is useless: there is no run that conforms to (1, 3, 4, 5; 2). Hence we really have built a deterministic Büchi automaton.

Alas, the last machine is too big: by “visual inspection” one finds that we could merge states 3 and 5, as well as 2 and 4.

After Merging

As a Büchi automaton, \( F = \{1, 2\} \).

Why Should This Work?

The key property of the construction is the following lemma (which says, in essence, our original plan has been duly implemented).

Lemma

Suppose \( T \) is a Safra tree in \( A \) that contains a marked node \( v \). Let \( x = x_1 x_2 \ldots x_i \) be a finite word such that \( v \) is an unmarked node in \( \delta(T, x_1 \ldots x_i) \) for \( i < k \) and a marked node in \( \delta(T, x_1 \ldots x_k) \). Let \( P_i \) be the label sets associated with node \( v \) in these trees.

Then \( P_i \subseteq \delta(P_0, x_1 \ldots x_i) \) and for all \( p \in P_k \) there is a run in the Büchi automaton starting at some \( q \in P_0 \) that touches a final state.

Proof.

We forgo the opportunity to inflict significant cognitive pain on the student body and do not prove the general case: we will only deal with the case where \( v \) is the root.

Proof, contd.

Since the root has no siblings to the left we have \( P_i = \delta(P_0, x_1 \ldots x_i) \).

Since \( P_k \) is marked, at time \( k - 1 \) we must have had a tree where the root had children; for simplicity let’s assume there are only 2 children. Hence there are times \( 0 < i < j < k \) where the children were introduced:

\[
P_0 \quad P_i \quad P_j \quad P_{k-1} \quad P_k
\]

But then \( P_k \) was obtained by a Vertical Merge, and any run from \( P_0 \) to \( P_k \) passes through a final state: \( R_i = P_i \cap F \) and \( S_j \subseteq P_j \cap F \).

Recall: König’s Lemma

Theorem

Any infinite, finitely-branching tree must have an infinite branch.

Proof.

Start with \( r_0 \), the root. Since the tree is finitely-branching, one of the children of the root must span an infinite subtree. Let \( r_1 \) be one of these fat children.

Done by induction.

We can think of the lemma as a weak choice principle. This is more powerful than plain Peano arithmetic (which suffices for ordinary finite state machines).

Note that the construction is non-constructive: if the tree were, say, computable, we would not know how to actually determine \( r_1 \).
**Correctness**

**Theorem**

Let $A$ be the Rabin automaton obtained by applying Safra’s algorithm to a Büchi automaton $B$. Then $L(A) = L(B)$.

**Proof.**

First assume $A$ accepts $x \in \Sigma^\omega$. Then there is a node named $v$ that appears infinitely often marked in the run of $A$ on $x$. Moreover, after some initial segment, all trees in the run contain $v$. Since $v$ is marked infinitely often, there is a chain of state sets $P_i$, $i \in \mathbb{N}$ and $i < i_{i+1}$, that appear as labels of $v$ when the node is marked.

By the lemma, every state in $P_{i+1}$ can be traced back to a state in $P_i$. By induction, there is a partial (meaning finite) run starting at $I$ to every state in $P_i$ for all $i$.

Think of these runs as defining nodes in a tree, the tree of all finite initial segments of computations of the Büchi automaton. Clearly, the tree is finitely branching and is infinite. By König’s lemma it must contain an infinite branch – which branch corresponds to an accepting computation of $B$ on $x$.

**Complementation**

**Lemma**

Recognizable languages of $\Sigma^\omega$ are closed under complementation.

**Proof.** To see this, first construct a Rabin automaton for the language. Then convert the Rabin automaton into an equivalent Muller automaton (which is still deterministic). We know how to complement the Muller automaton and convert back to Büchi.

Hence recognizable $\omega$-languages again form a Boolean algebra, and the operations are effective: we can compute the corresponding machines.

**Exercise**

Modify Safra’s algorithm so that it produces directly a Muller automaton.

**Complexity**

If we measure complexity in terms of the minimal number of states in any Büchi automaton recognizing the language, we have

- union: $O(n_1 + n_2)$
- intersection: $O(n_1 n_2)$
- complement: $2^{O(n \log n)}$

The horrendous upper bound for complementation is not just theoretical: one can construct artificial languages $L_n$ accepted by a Büchi automaton on $n + 2$ states such that any Büchi automaton for the complement of $L_n$ has at least $n!$ states.

Thus any algorithm using repeated complementation may well blow up and fail in any practical sense.

**Running Time**

The only general bound on the size of the Rabin automaton is $2^{O(n \log n)}$.

Unfortunately, this is asymptotically optimal: there are Büchi automata that exhibit this type of blow-up during determinization.

Safra’s algorithm has the vexing property that even though one may believe one understands it completely, things often go wrong when one actually tries to implement it. One problem is that there are several reasonable versions that differ slightly in their behavior.

**Exercise**

Implement Safra’s algorithm in the language of your choice.

**Proof, contd.**

For the opposite direction, let $B$ accept $x$ and let $\pi$ be a corresponding run; say, $p$ is a final state that appears infinitely often in $\pi$. Then the corresponding states $p_i$ appear in the root of the Safra trees in the unique run of $A$ on $x$. If the root is marked infinitely often, $A$ accepts and we are done.

Otherwise, since $p$ appears infinitely often in $\pi$, it must appear in some child of the root. After a while, it will settle down in the leftmost position. If the corresponding node is marked infinitely often, we are done. Otherwise, by the same argument, we consider a node at level 2.

Since the trees have bounded depth, we must ultimately reach a level where the node is marked infinitely often, and $A$ accepts $x$. 

**Hacking Challenge**

The standard choice of name for a new node during the computation of $T' = \delta_a(T)$ is the least available one:

$$\text{new} = \min(V - \text{current nodes}).$$

But one could also assign a symbolic name and then, after the tree has been constructed, try to assign actual names in such a way that $T'$ has already been encountered earlier on.

**Exercise**

Implement this algorithm so that it beats the standard one, at least on occasion.
We now have all the pieces in place: we can use ω-automata to construct a decision procedure for Presburger arithmetic.

Perhaps surprisingly, we will use weak monadic second-order logic for this purpose.

Exercise
What is the complexity of Büchi Emptiness? Explore Emptiness tests for Muller and Rabin automata.

We have the counterpart of “NFA Emptiness” for infinite words.

Problem: Büchi Emptiness
Instance: A Büchi automaton A.
Question: Is Lω(A) empty?

This is easily decidable: there has to be a path from an initial state to a final state p such that p lies in a non-trivial strongly connected component of the diagram of A.

Exercise
What is the complexity of Büchi Emptiness? Explore Emptiness tests for Muller and Rabin automata.

The last automaton accepts the language Σ*aΣ*bΣω.

Weak monadic second order logic is define like MSO, except that we quantify over finite subsets of the domain. For example,

∀X (∃x X(x) ⇒ ϕ(X))

means that, for any non-empty finite set of positions P, ϕ(P) holds. So if ϕ(X) is

∃x (X(x) ∧ ∀y (X(y) ⇒ y ≤ x))

we get a valid formula (which is invalid in full second-order).

As already mentioned, Büchi automata can easily deal with the additional finiteness condition for the second-order tracks. Even better, the construction is essentially the same as in the full second-order case.

It follows that weak monadic second order logic (with <) is also decidable, using essentially the same algorithm.
One might wonder why Büchi’s theorem is important outside of pure theory.

One-way infinite words arise naturally in the study of non-terminating programs (such as operating systems) or certain protocols, so it is important to have some tools available to deal with infinite words.

Another application is perhaps more surprising: logic on infinite words can be used to express assertions in arithmetic – which, in turn, are important for program verification.

Ordinary arithmetic is the study of the structure
\[ \mathbb{N} = \langle \mathbb{N}, +, \times, 0, 1; < \rangle \]

Alas, even \( \Sigma_1 \) statements of the form
\[ \exists x_1, \ldots, x_n \varphi(x_1, \ldots, x_n) \]
are already undecidable in general over \( \mathbb{N} \) (where \( \varphi \) has only bounded quantifiers): we can express Diophantine equations this way.

And truth of all of first-order logic over \( \mathbb{N} \) is highly undecidable.

How about weaker structures that have fewer operations?

Realistically, the only useful choice is to drop multiplication. This yields \textbf{Presburger arithmetic}:
\[ \mathbb{N}_+ = \langle \mathbb{N}, +, 0, < \rangle \]

Since multiplication is missing, one cannot describe polynomials in this setting, only linear combinations.

So the problem of Diophantine equations disappears and there is some hope that a decision algorithm might exist.

Without multiplication, arithmetic is much less complicated.

\begin{itemize}
  \item In 1929, Presburger showed that Peano arithmetic without multiplication (Presburger arithmetic) is decidable.
  \item In 1930, Skolem proved that Peano arithmetic without addition (Skolem arithmetic) is decidable.
  \item In 1931, Gödel showed that full Peano arithmetic (actually: any arithmetic) is incomplete and hence necessarily undecidable.
  \item In 1948, Tarski showed that real arithmetic is decidable.
  \item In 1970, Matiyasevic showed that Diophantine equations are undecidable.
\end{itemize}
WMSO[$<<$] can be used to give a decision procedure for Presburger arithmetic that seems to work reasonably well in practice (though, in principle, the use of determinization could cause blow-up).

One might think that natural numbers would be represented by first order variables ranging over positions in a word: after all, in an infinite word these positions are just $\mathbb{N}$.

Alas, that won’t work: we need to be able to check addition. We would need a finite state machine that accepts three track binary words of the form

$$0^i10^j : 0^i10^j : 0^{i+j}10^r$$

Impossible by the Pumping Lemma.

The trick is to represent natural numbers by second order variables, finite sets $X \subseteq \mathbb{N}$:

$$\text{val}(X) = \sum_{i \in X} 2^i$$

Thus, $X$ is essentially just the standard reverse binary expansion.

Now an automaton can check $\text{val}(X) + \text{val}(Y) = \text{val}(Z)$:

$$\begin{array}{c|cccccccccccc}
X & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots \\
Y & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\
Z & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & \ldots \\
\end{array}$$

This is really the same argument that shows that addition in reverse binary is synchronous.

Similarly we can check $\text{val}(X) < \text{val}(Y)$ and so forth.

In programs that are not too terribly complex, index arithmetic can often be described in terms of Presburger arithmetic.

Being able to check the validity of Presburger formulae is thus directly relevant in program verification.

This is used for example in Microsoft’s Spec# system, an extension of C# that includes specifications and tools to verify these specifications.