Molecular Computation Models in ACL2: a Simulation of Lipton's Experiment Solving SAT.*

F.J. Martín-Mateos, J.A. Alonso, M.J. Pérez-Jiménez and F. Sancho-Caparrini
http://www.cs.us.es/{~fmartin,~jalonso,~marper,~fsancho}
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

February 28, 2002

Abstract

In this paper we present an ACL2 formalization of a molecular computing model: Adleman's restricted model [2]. This is a first step to formalize unconventional models of computation in ACL2. As an application of this model, an implementation of Lipton's experiment solving SAT [7] is described, based on the formalization given in [6]. We use ACL2 to make a formal proof of the completeness and soundness properties of the function implementing the experiment.

1 Introduction

At the beginning of the fifties the analogy between some mathematical procedures and biological processes starts to be established. L.M. Adleman [1] proved this relation in 1994, showing that it was possible to use biological processes to solve difficult mathematical problems: he designed a biological experiment based on DNA manipulation to solve instances of the Hamiltonian path problem, a well known NP-complete problem. This experiment can be considered as a first step to build a prototype of a molecular computer.

In 1995, R.J. Lipton [7] solved an instance of the satisfiability problem of propositional logic, using the method of Adleman. Lipton's experiment is interesting because the initial test tube does not depend on the propositional formula, but on the number of its variables. In this way, the experiment is a molecular solution to every instance of the satisfiability problem with a fixed set of variables and hence, it provides a molecular algorithm.

Adleman and Lipton's experiments are the starting point of molecular computation and reveals its huge advantage in parallelism with respect to the conventional electronic computers. In 1995 [2] the first formal models of molecular computing appeared, based on a set of basic biochemical operations. In 1996, D. Beaver [3] proved that these models are computationally complete, in the sense that every computation of a Turing machine can also be achieve by a molecular machine.

In this paper we present an ACL2 formalization of a molecular computing model: Adleman's restricted model. This is a first step to formalize unconventional models of computation in ACL2. The functions implementing the main operations in Adleman's model are disabled after proving its properties. So, the subsequent development is generic in that the specific operations are not important, but the necessary properties of these operations are important.

In [6] a formalization of Lipton's experiment is given as an iterative algorithm based on the elemental operations of Adleman's restricted model. We define recursive functions implementing

*This work has been supported by MCyT: Project TIC2000-1368-CO3-02

this formalization and we prove the completeness and soundness theorems of these functions. A transcription of these proofs is provided in this paper.

Finally we focus on two remarks of our implementation. First, a function providing the initial test tube for Lipton's experiment is provided and second, a final test is incorporated to the Lipton's experiment to decide the satisfiability of a propositional formula. We also show the completeness and soundness theorems in both cases and explain the ideas we have followed to prove them.

2 Adleman's restricted model

In [2] some abstract models for molecular computing are described. The first model proposed works with test tubes with a set of DNA molecules (i.e. a multiset of finite sequences over the alphabet $\{A, C, G, T\}$). Nevertheless, it may be preferable to use molecules other than DNA, using an alphabet Σ which is not necessarily $\{A, C, G, T\}$. Further, though DNA has a natural structure which allows to order the occurrence of elements and hence deal with sequences, this may not be true for other types of molecules. Then, the members of a tube will be multisets of elements from Σ. In the sequel, we consider an alphabet Σ and call *aggregate* a multiset of elements from this alphabet.

The above considerations are the basis of the restricted model of molecular computation. This model works on test tubes with a multiset of aggregates (i.e. a multiset of multisets of elements from Σ). On these tubes, the following operations can be performed:

- $Separate(T, x)$: Given a tube T and an element $x \in \Sigma$, produces two new tubes, $+(T, x)$ and $-(T, x)$, where $+(T, x)$ is the tube consisting of every aggregate of T which contains the element x and $-(T, x)$ is the tube consisting of every aggregate of T which does not contain the element x:

$$+(T, x) = \{\gamma \in T : x \in \gamma\}$$
$$-(T, x) = \{\gamma \in T : x \notin \gamma\}$$

- $Merge(T_1, T_2)$: Given tubes T_1 and T_2, produces the new tube $T_1 \cup T_2$, which is the multiset union of the multisets T_1 and T_2.

- $Detect(T)$: Given a tube T, decides if T contains at least one aggregate; that is, returns "yes" if T contains at least one aggregate and returns "no" if it contains none.

These operations are performed in the laboratory in the following way. If a *Merge* of tubes is required, this is accomplished by pouring the contents of one of the tubes into the other. If a *Separate* or a *Detect* operation is required on a tube then some technical operations (magnetic bead system, polymerase chain reaction, get electrophoresis, ...) are performed on it. This model is called "restricted" in the sense that the molecules themselves do not change in the course of a computation.

To formalize the restricted model in ACL2, we use lists to represent multisets. Then, a test tube is represented as a list of aggregates and an aggregate is represented as a list of elements from Σ. So, the functions associated with the molecular operations work on lists.

We consider two functions, `separate+` and `separate-`, associated with the *Separate* operation. The first one returning the value $+(T, x)$ and the second one the value $-(T, x)$. The *Merge* operation is associated with the function `tube-merge`. Finally, we consider the function `detect` associated with the *Detect* operation.

The definition of these functions is not as much interesting as their properties. The proof of the properties of any algorithm built on the restricted model must be independent of the implementation of the operations. This will ensure us the properties of this algorithm even when it is

evaluated in a molecular laboratory. Therefore, we use an `encapsulate` to assume the properties of these functions, providing the adequate witness functions. The properties and the associated events are the following:

- `member-separate+` : $\gamma \in +(T, x) \leftrightarrow (x \in \gamma) \wedge (\gamma \in T)$

```
(defthm member-separate+
  (iff (member aggr (separate+ tube X))
       (and (member X aggr)
            (member aggr tube))))
```

- `member-separate-` : $\gamma \in -(T, x) \leftrightarrow (x \notin \gamma) \wedge (\gamma \in T)$

```
(defthm member-separate-
  (iff (member aggr (separate- tube X))
       (and (not (member X aggr))
            (member aggr tube))))
```

- `member-tube-merge` : $\gamma \in T_1 \cup T_2 \leftrightarrow (\gamma \in T_1) \vee (\gamma \in T_2)$

```
(defthm member-tube-merge
  (iff (member aggr (tube-merge tube1 tube2))
       (or (member aggr tube1)
           (member aggr tube2))))
```

- `member-detect` : $Detect(T) \leftrightarrow \exists e \in T$

```
(defthm member-detect
  (and (implies (member e tube)
                (detect tube))
       (implies (detect tube)
                (member (car tube) tube))))
```

If we want to test any algorithm build on the restricted model, we must provide functions implementing the basic operations and prove the encapsulated properties for them.

3 Lipton's experiment

Adleman's Experiment [1] solved an instance of the Hamiltonian path problem over a directed graph with two designated vertices, by implementing a brute force procedure in a laboratory of molecular biology. To solve the problem, an initial test tube with DNA molecules encoding all the paths in the graph was built. This tube was subjected to some operations based on DNA manipulation, and every aggregate encoding a path which was not a valid solution of the problem was removed.

Lipton shows in [7] how to solve an instance of the satisfiability problem for Propositional Logic, using the ideas of Adleman. To achieve this, he described every relevant assignment of a propositional formula by means of paths on a directed graph associated with the variable set of the formula. Specifically, given a propositional formula in conjunctive normal form, $F = c_1 \wedge \ldots \wedge c_p$, with $c_i = l_{i,1} \vee \ldots \vee l_{i,r_i}$, and the set of variables $Var(F) = \{x_1, \ldots, x_n\}$, the associated directed graph $G_n = (V_n, E_n)$ is defined as follows:

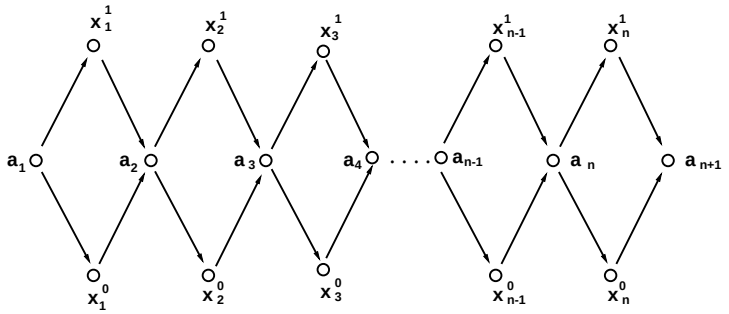

Figure 1: Directed graph associated with a propositional formula with n variables

$$V_n = \{x_i^j : 1 \leq i \leq n, 0 \leq j \leq 1\} \cup \{a_i : 1 \leq i \leq n+1\}$$
$$E_n = \{(a_i, x_i^j), (x_i^j, a_{i+1}) : 1 \leq i \leq n, 0 \leq j \leq 1\}$$

This graph, shown in figure 1, verifies the following properties:

- There are 2^n paths from a_1 to a_{n+1}.
- There exists a natural bijection between the above set of paths and the relevant assignments of F, according to the following criteria: given $\gamma = a_1 x_1^{j_1} a_2 x_2^{j_2} \ldots x_n^{j_n} a_{n+1}$, a path from a_1 to a_{n+1}, then the assignment $\hat{\gamma}$ is associated with it, such as $\hat{\gamma}(x_i) = j_i,\ 1 \leq i \leq n$.

The initial test tube contains DNA molecules codifying the paths from a_1 to a_{n+1}, and so every relevant assignment of F. With the alphabet $\Sigma = \{a_i, x_i^j, a_{n+1} : 1 \leq i \leq n, 0 \leq j \leq 1\}$, the initial test tube considered is the following:

$$T_0 = \{\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, \ldots, x_n^{j_n}, a_{n+1}\} : 1 \leq i \leq n, j_i \in \{0,1\}\}$$

Lipton's experiment can be described as follows: for each clause in the initial formula, every aggregate representing an assignment falsifying this clause is removed. The way to work with clauses is the following: for each literal in the clause, every aggregate representing an assignment in which this literal is true is preserved, and the remaining aggregates are removed. This experiment has been formalized in [6], where it has been expressed as an algorithm based on the elemental operations of Adleman's restricted model:

```
Input: T_0 (as described above)
  For i ← 1 to p do
    T_{i,0} ← ∅;
    T''_{i,0} ← T_{i-1};
    For j ← 1 to r_i do
      T'_{i,j} ← +(T''_{i,j-1}, l^1_{i,j})
      T''_{i,j} ← −(T''_{i,j-1}, l^1_{i,j})
      T_{i,j} ← T_{i,j-1} ∪ T'_{i,j}
    T_i ← T_{i,r_i}
  Detect(T_p)
```

Where, for each literal $l_{i,j}$ in the initial formula:

$$l_{i,j}^1 = \begin{cases} x_m^1 \text{ if } l_{i,j} = x_m \\ x_m^0 \text{ if } l_{i,j} = \neg x_m \end{cases}$$

The number of molecular operations needed to execute this procedure to a formula with k literals is $O(k)$ (k separations, k merges and 1 detect). Also, the number of tubes used is:

$$1 + \sum_{i=1}^{p}(3 + 3 \cdot r_i) = 1 + 3p + 3k \in O(k)$$

The properties we want to prove about this algorithm are the following:

1. Soundness: $\forall \gamma \in T_p, (\hat{\gamma}(F) = 1)$
2. Completeness: $\forall \gamma \in T_0, (\hat{\gamma}(F) = 1 \Rightarrow \gamma \in T_p)$

where $\hat{\gamma}(F)$ is the truth value of F in the assignment $\hat{\gamma}$ (The truth value of a CNF formula is extended as usual).

Next, we present our implementation of Lipton's experiment in ACL2. First of all we must notice that the above algorithm depends on the initial test tube (T_0) and the propositional formula (F), by means of the $l^1_{i,j}$. Then, we have developed a function with two arguments, the formula and the tube. In fact, we have defined two functions, one for each loop. The function dealing with the external loop works recursively on the number of clauses of F, and the other one works recursively on the number of literals of the selected clause.

To formalize the algorithm we need to obtain the values $l^1_{i,j}$ from the literals $l_{i,j}$. Also, to express the properties we want to prove, we need to calculate the truth value of a formula in an assignment. This brings us to formalize some concepts about propositional logic.

3.1 Propositional logic

First, we deal with the syntax. The propositional variables (or simply variables) are any ACL2 `symbolp` except `nil`. A positive literal (`positive-literal-p`) is any propositional variable. A negative literal (`negative-literal-p`) is a list with two elements, the first one is the symbol `-` and the second one is any propositional variable. A literal (`literal-p`) is any positive or negative literal. A clause (`clause-p`) is a list of literals. A CNF formula (`cnf-formula-p`) is a list of clauses.

With respect to the semantic, we use the symbols `0` and `1` as truth values (false and true respectively). The function `opposite-value`, returns the opposite of a given truth value. To represent assignments, we use association lists. In these lists a propositional variable can have associated any value; if this value is `1`, the variable is interpreted as true, if the value is not `1` or the variable is not in the association list, then it is interpreted as false. The function that returns the truth value of a propositional variable in an assignment is `prop-variable-value`.

The truth value of a literal in an assignment (`literal-value`) depends on its sign. The truth value of a positive literal is the value of its variable. The truth value of a negative literal is the opposite of the value of its variable. The truth value of a clause (`clause-value`) is true if some of its literals is true, and false otherwise. The truth value of a CNF formula (`cnf-formula-value`) is false if some of its clauses is false, and true otherwise.

3.2 Implementing the experiment

In this section, we present the functions implementing the loops of Lipton's experiment. This implementation does not use the *Detect* operation, it returns the last tube in the external loop (T_p). Also, the initial test tube must be provided as an argument. In section 4 we show implementations without these limitations and discuss how we have proved their properties.

First of all we need a function that builds the element $l^1_{i,j}$ from the literal $l_{i,j}$. To represent the elements $l^1_{i,j}$ we use pairs: the element x^0_i is represented with the pair `(xi . 0)` and the element x^1_i with the pair `(xi . 1)`.

```
(defun l-element (label L)
  (cond ((negative-literal-p L)
         (cons (second L) (opposite-value label)))
        (t (cons L label))))
```

Next, we define a function implementing the internal loop. Its inputs are a main tube, T_i'', an accumulator tube, T_i, and a clause. The aggregates in the main tube containing the element $l_{i,1}^1$ are merged with the accumulator tube in a new one. The aggregates in the main tube that do not contain the element $l_{i,1}^1$ are poured in a new main tube. The new main and accumulator tubes are used in the recursive call on the rest of the literals:

```
(defun sat-lipton-clause (C tube acc)
  (cond ((endp C) acc)
        (t (let* ((tube+ (separate+ tube (l-element 1 (car C))))
                  (tube- (separate- tube (l-element 1 (car C))))
                  (n-acc (tube-merge acc tube+)))
             (sat-lipton-clause (cdr C) tube- n-acc)))))
```

The main function deals with the external loop. Its inputs are a tube and a CNF formula. This function applies the internal loop on this tube, an initially empty accumulator tube and the first clause of the formula. The result of this process is used as initial tube in the recursive call on the rest of clauses:

```
(defun sat-lipton-cnf-formula (F tube)
  (cond ((endp F) tube)
        (t (let ((n-tube (sat-lipton-clause (car F) tube nil)))
             (sat-lipton-cnf-formula (cdr F) n-tube)))))
```

The figure in the following page shows the behaviour of these functions and its relation with the algorithm associated with Lipton's experiment.

We have proved the following basic but important properties about the algorithm:

- $\forall j = 0, \ldots, r_i (\gamma \in T_{i,j} \Rightarrow \gamma \in T_{i,r_i})$

```
(defthm member-clause-accumulator
  (implies (member aggr acc)
           (member aggr (sat-lipton-clause C tube acc))))
```

- $\forall j = 0, \ldots, r_i (\gamma \in T_{i,r_i} \Rightarrow \gamma \in T_{i,j} \vee \gamma \in T_{i,j}'')$

```
(defthm member-clause
  (implies (member aggr (sat-lipton-clause C tube acc))
           (or (member aggr tube)
               (member aggr acc))))
```

- $\forall i = 1, \ldots, p (\gamma \in T_i \Rightarrow \gamma \in T_{i-1})$

```
(defthm member-clause-accumulator-nil
  (implies (member aggr (sat-lipton-clause C tube nil))
           (member aggr tube)))
```

- $\gamma \in T_p \Rightarrow \gamma \in T_0$

```
(defthm member-cnf-formula
  (implies (member aggr (sat-lipton-cnf-formula F tube))
           (member aggr tube)))
```

```
(defconst *example* '((p q) ((- q) p)))

(sat-lipton *example*)

; (SAT-LIPTON-CNF-FORMULA '((P Q) ((- Q) P))
;  '((((A) . 1) (Q . 0) ((A) . 2) (P . 0) ((A) . 3))          T_0
;    (((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))
;    (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3))
;    (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))))
;
;    => (SAT-LIPTON-CLAUSE '(P Q)
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 0) ((A) . 3))    T''_{1,0}
;          (((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3)))
;        ())                                                  T_{1,0}
;
;    => (SAT-LIPTON-CLAUSE '(Q)
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 0) ((A) . 3))    T''_{1,1}
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3)))
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))    T_{1,1}
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))))
;
;    => (SAT-LIPTON-CLAUSE ()
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 0) ((A) . 3)))   T''_{1,2}
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3))))  T_{1,2}
;
; (SAT-LIPTON-CNF-FORMULA '(((- Q) P))
;  '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))          T_1
;    (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))
;    (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3))))
;
;    => (SAT-LIPTON-CLAUSE '((- Q) P)
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))    T''_{2,0}
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3)))
;        ())                                                  T_{2,0}
;
;    => (SAT-LIPTON-CLAUSE '(P)
;        '((((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))    T''_{2,1}
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3)))
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))))  T_{2,1}
;
;    => (SAT-LIPTON-CLAUSE ()
;        '((((A) . 1) (Q . 1) ((A) . 2) (P . 0) ((A) . 3)))   T''_{2,2}
;        '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))
;          (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))))  T_{2,2}
;
; (SAT-LIPTON-CNF-FORMULA ()
;  '((((A) . 1) (Q . 0) ((A) . 2) (P . 1) ((A) . 3))          T_2
;    (((A) . 1) (Q . 1) ((A) . 2) (P . 1) ((A) . 3))))
```

3.3 Aggregates and properties

The soundness and completeness properties of the algorithm have two hidden assumptions:

1. F is a CNF formula (`cnf-formula-p`).
2. γ is an aggregate with the form: $\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, \ldots, x_n^{j_n}, a_{n+1}\}$ with $1 \leq i \leq n$ and $j \in \{0, 1\}$

We do not have formalized any property about the aggregates. Instead, we have checked that the following property is enough to characterize them: for each variable x_i in the original formula, there must exist one and only one x_i^j in the aggregate. Therefore, the elements a_i in the aggregates are not necessary. Nevertheless, we have to consider them to reflect the original experiment. To formalize this property we must clarify the representation of aggregates. Some aggregates on the variable set {`X`, `Y`, `Z`} are:

```
;   ( ((A) . 1) (X . 1) ((A) . 2) (Y . 0) ((A) . 3) (Z . 0) ((A) . 4) )
;   ( ((A) . 1) (X . 0) ((A) . 2) (Y . 0) ((A) . 3) (Z . 1) ((A) . 4) )
;   ( ((A) . 1) (X . 1) ((A) . 2) (Y . 1) ((A) . 3) (Z . 0) ((A) . 4) )
```

We have chosen the pairs `((A) . i)` to represent the elements a_i. We use `(A)` instead of `A` to avoid problems if the symbol `A` occurs as a variable in the original formula. It is important to notice that the representation chosen can also be used to represent the assignment associated with the aggregate, because `(A)` is not a valid propositional variable in our formalization.

We have defined three functions checking the above property. The first one (`variable--aggregate-p`) checks the property with respect to a propositional variable, the second one (`clause-aggregate-p`) with respect to the variable set of a clause and the third one (`cnf-formula--aggregate-p`) with respect to the variable set of a CNF formula. The functions `literal--variable`, `clause-variables` and `cnf-formula-variables` are used to obtain, respectively, the variable of a literal, the variable set of a clause and the variable set of a CNF formula. In the sequel, when we say that γ is an aggregate w.r.t. a literal, a clause or a formula, we mean that γ is an aggregate with respect to its variable or its variable set.

Finally we establish the relationship between the aggregates and the relevant assignments of the original formula:

- $x^i \in \gamma \leftrightarrow \hat{\gamma}(x) = i$

```
(defthm variable-aggregate-p-valor-member
  (implies (and (prop-variable-p X)
                (variable-aggregate-p aggr X))
           (iff (member (cons X V) aggr)
                (equal (prop-variable-value X aggr) V))))
```

3.4 Completeness property

The completeness property is the following:

$$\forall \gamma \in T_0(\hat{\gamma}(F) = 1 \Rightarrow \gamma \in T_p)$$

We must add conditions establishing that F is a CNF formula and γ is an aggregate w.r.t. F. Next, we show the formalization of this theorem in ACL2, where `aggr` is γ, `tube` is T_0 and `F` is F:

Theorem 3.1 *Completeness of* `sat-lipton-cnf-formula`.

$$\left.\begin{array}{l}\gamma \in T_0 \\ F \text{ is a CNF formula} \\ \gamma \text{ is an aggregate w.r.t. } F \\ \hat{\gamma}(F) = 1\end{array}\right\} \Rightarrow \gamma \in T_p$$

ACL2 theorem:

```
(defthm completeness-sat-lipton-cnf-formula
  (implies (and (member aggr tube)
                (cnf-formula-p F)
                (cnf-formula-aggregate-p aggr F)
                (equal (cnf-formula-value F aggr) 1))
           (member aggr (sat-lipton-cnf-formula F tube)))
```

The function `sat-lipton-cnf-formula` implements the external loop of Lipton's experiment by calling the function `sat-lipton-clause`. So, we must prove a similar result about this last function. Finally, it is also necessary to establish another lemma about the function `separate+` and the literals. These lemmas are necessaries to prove the completeness theorem, so they precede it in the ACL2 book.

Lemma 3.1 *Completeness of* `separate+`.

$$\left.\begin{array}{l}\gamma \in T \\ L \text{ is a literal} \\ \gamma \text{ is an aggregate w.r.t. } L \\ \hat{\gamma}(L) = 1\end{array}\right\} \Rightarrow \gamma \in +(T, L^1)$$

ACL2 theorem:

```
(defthm completeness-separate+
  (implies (and (member aggr tube)
                (literal-p L)
                (variable-aggregate-p aggr (literal-variable L))
                (equal (literal-value L aggr) 1))
           (member aggr (separate+ tube (l-element 1 L)))))
```

Lemma 3.2 *Completeness of* `sat-lipton-clause`.

$$\left.\begin{array}{l}\gamma \in T''_{i,0} \\ C_i \text{ is a clause} \\ \gamma \text{ is an aggregate w.r.t. } C_i \\ \hat{\gamma}(C_i) = 1\end{array}\right\} \Rightarrow \gamma \in T_{i,r_i}$$

ACL2 theorem:

```
(defthm completeness-sat-lipton-clause
  (implies (and (member aggr tube)
                (clause-p C)
                (clause-aggregate-p aggr C)
                (equal (clause-value C aggr) 1))
           (member aggr (sat-lipton-clause C tube acc)))
```

3.5 Soundness property

The soundness property is the following:

$$\forall \gamma \in T_p(\hat{\gamma}(F) = 1)$$

As in the completeness property, we must add conditions establishing that F is a CNF formula and γ is an aggregate w.r.t. F. The formalization in ACL2 is the following, where `aggr` is γ, `tube` is T_0 and `F` is F:

Theorem 3.2 *Soundness of* `sat-lipton-cnf-formula`.

$$\left.\begin{array}{l} F \text{ is a CNF formula} \\ \gamma \text{ is an aggregate w.r.t. } F \\ \gamma \in T_p \end{array}\right\} \Rightarrow \hat{\gamma}(F) = 1$$

ACL2 theorem:

```
(defthm soundness-sat-lipton-cnf-formula
  (implies (and (cnf-formula-p F)
                (cnf-formula-aggregate-p aggr F)
                (member aggr (sat-lipton-cnf-formula F tube)))
           (equal (cnf-formula-value F aggr) 1))
```

Analogously to the completeness case, we also need lemmas showing similar properties about `sat-lipton-clause` with respect to clauses and `separate+` with respect to literals. These lemmas precede the main theorem in the ACL2 book:

Lemma 3.3 *Soundness of* `separate+`*:*

$$\left.\begin{array}{l} L \text{ is a literal} \\ \gamma \text{ is an aggregate w.r.t. } L \\ \gamma \in +(T, L^1) \end{array}\right\} \Rightarrow \hat{\gamma}(L) = 1$$

ACL2 theorem:

```
(defthm soundness-separate+
  (implies (and (literal-p L)
                (variable-aggregate-p aggr (literal-variable L))
                (member aggr (separate+ tube (l-element 1 L))))
           (equal (literal-value L aggr) 1)))
```

Lemma 3.4 *Soundness of* `sat-lipton-clause`*:*

$$\left.\begin{array}{l} C_i \text{ is a clause} \\ \gamma \text{ is an aggregate w.r.t. } C_i \\ \gamma \in T_{i,r_i} \\ \gamma \text{ is not in the accumulator} \end{array}\right\} \Rightarrow \hat{\gamma}(C_i) = 1$$

ACL2 theorem:

```
(defthm soundness-sat-lipton-clause
  (implies (and (clause-p C)
                (clause-aggregate-p aggr C)
                (not (member aggr acc))
                (member aggr (sat-lipton-clause C tube acc)))
           (equal (clause-value C aggr) 1))
```

4 Completing the experiment

The functions developed in the previous section are not completely satisfactory. There are two points we do not have considered yet. The first one is the initial test tube: the function `sat-lipton-cnf-formula` must receive a complete initial test tube to ensure the satisfiability of a formula. The second one is the value returned by `sat-lipton-cnf-formula`: since we do not have used the *Detect* function, the result is not a boolean value. Next, we present our solution to these questions.

4.1 Building an initial test tube

To build the initial test tube we process the list representing the variable set of the CNF formula, $\{x_1, ..., x_n\}$. From this list we build lists representing every aggregate $\{a_1, x_1^{j_1}, a_2, x_2^{j_2}, ..., x_n^{j_n}, a_{n+1}\}$, where $j_i \in \{0, 1\}$. This is done recursively on the initial list by the function `build-tube`, using an index to add the adequate mark a_i. Then, the initial test tube for a CNF formula F is the result of building a tube with the variable set of F, using 1 as an initial index for marks.

```
(defun build-initial-tube (F)
  (build-tube (cnf-formula-variables F) 1))
```

Now, we obtain a satisfiability check function for propositional logic, using the result of the above function as the initial test tube in the function implementing the experiment:

```
(defun sat-lipton (formula-fnc)
  (sat-lipton-cnf-formula F (build-initial-tube F)))
```

Of course, we must prove the appropriate completeness and soundness theorems to ensure that the last function checks the satisfiability of a CNF formula F. We can use the completeness and soundness theorems about `sat-lipton-cnf-formula` to achieve this, previously proving that the elements of the initial test tube are aggregates w.r.t. F. This property about `build-initial-tube` is the following:

```
(defthm build-initial-tube-cnf-formula-aggregates
  (implies (and (cnf-formula-p F)
                (member aggr (build-initial-tube F)))
           (cnf-formula-aggregate-p aggr F)))
```

In this theorem the function `build-initial-tube` is expanded (if it is not disabled) obtaining a too much specific term involving the function `build-tube`. We have proved a similar theorem about `build-tube`, using the term `(build-tube Xs i)` instead of `(build-initial-tube F)`. Of course, we must add new hypothesis about `Xs` with respect to the formula `F`:

H_1) `Xs` contains the variable set of `F`: `(subsetp (cnf-formula-variables F) Xs)`

H_2) `Xs` does not have repeated elements: `(no-duplicatesp Xs)`

H_3) The term `(a)` is not in `Xs` : `(not (member '(a) Xs))`

This more general theorem is:

```
(defthm build-tube-cnf-formula-aggregates
  (implies (and (subsetp (cnf-formula-variables F) Xs)
                (no-duplicatesp Xs)
                (not (member '(a) Xs))
                (cnf-formula-p F)
                (member aggr (build-tube Xs i)))
           (cnf-formula-aggregate-p aggr F)))
```

Finally, an instance of this result provides the proof of the property about `build-initial-tube`. In this instance `i` is replaced by `1` and `Xs` is replaced by `(cnf-formula-variables F)`. To prove the instance we also must prove instances of the new hypothesis:

- Instance of H_1: `(subsetp (cnf-formula-variables F) (cnf-formula-variables F))`

  This property is true because `subsetp` is reflexive.

- Instance of H_2: `(no-duplicatesp (cnf-formula-variables F))`

  This property is easily proved providing similar results about `clause-variables` and `union--equal` (when the arguments do not have repeated elements).

- Instance of H_3: `(not (member '(a) (cnf-formula-variables F)))`

  This holds because `F` is a CNF formula in our formalization of propositional logic in which `(a)` is not a propositional variable.

Finally we prove the completeness and soundness theorems of `sat-lipton`:

```
(defthm completeness-sat-lipton
  (implies (and (cnf-formula-p F)
                (member aggr (build-initial-tube F))
                (equal (cnf-formula-value F aggr) 1))
           (member aggr (sat-lipton F))))

(defthm soundness-sat-lipton
  (implies (and (cnf-formula-p F)
                (member aggr (sat-lipton F)))
           (equal (cnf-formula-value F aggr) 1)))
```

4.2 A satisfiability checker

The other point to consider is the use of *Detect* operation as the last step in Lipton's experiment. Using it we have defined a function that decides if a CNF propositional formula is satisfiable or not:

```
(defun sat-lipton-p (F)
  (detect (sat-lipton F)))
```

On the other hand, it is noticed that the completeness theorems showed before are not satisfactory in the sense that they do not involve an assignment, but an aggregate of the test tube. This differs from the `sat-lipton-p` completeness theorem:

```
(defthm completeness-sat-lipton-p
  (implies (and (cnf-formula-p F)
                (equal (cnf-formula-value F assig) 1))
           (sat-lipton-p F)))
```

To prove this theorem we have used the completeness theorem of `sat-lipton`, building an aggregate `aggr` from the assignment, with the following properties:

- `(member aggr (build-initial-tube F))`
- `(equal (cnf-formula-value F aggr) 1)`

Then, by the completeness theorem, `aggr` belongs to `(sat-lipton F)` and hence this is not empty, therefore `(detect (sat-lipton F))` is true.

Finally the soundness theorem of `sat-lipton-p` is easily proved showing an assignment in which the initial formula is true. The first aggregate in the returned tube of `sat-lipton` can be used for this purpose:

```
(defthm soundness-sat-lipton-p
  (implies (and (cnf-formula-p F)
                (sat-lipton-p F))
           (equal (cnf-formula-value F (car (sat-lipton F))) 1)))
```

5 Conclusions and further work

In this work we have presented a formalization of Adleman's restricted model, one of the first molecular computing models. This formalization has been done in a generic framework in which the concrete implementation of its operations is not important, but only their properties. Using this formalization we have defined functions simulating Lipton's experiment solving SAT. Finally, the completeness and soundness properties of these functions have been proved.

The formalization of unconventional models of computation is a suitable way of working with them when we do not have real models (e.g. we do not have a laboratory implementing molecular computing models). This formalization brings us the possibility of simulate real experiments or develop new ones. With ACL2 we also can prove properties about these experiments. In this way, we actually work on the formalization of membrane computing models [8], [4], [5].

References

[1] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 11, 1994.

[2] Leonard M. Adleman. On constructing a molecular computer. In R.J. Lipton and E.B. Baum, editors, *DNA Based Computers*, number 27 in DIMACS Series, pages 1–21. American Mathematical Society, 1996.

[3] Donald Beaver. A universal molecular computer. In R.J. Lipton and E.B. Baum, editors, *DNA Based Computers*, number 27 in DIMACS Series, pages 29–36. American Mathematical Society, 1996.

[4] M. J. Pérez Jiménez and F. Sancho. A formalization of transition P–systems. *Fundamenta Informaticae*, 49 (1-3):261–272, 2002.

[5] M. J. Pérez Jiménez and F. Sancho. Verifying a P–system generating squares. *Romanian Journal of Information, Science and Technology*, 5 (2-3), 2002, to appear.

[6] M. J. Pérez Jiménez, F. Sancho, C. Graciani, and A. Romero. Soluciones moleculares del problema SAT (*in spanish*). In A. Nepomuceno and all, editors, *Lógica, Lenguaje e Información, JOLL'2000*, pages 243–252. Ed. Kronos, 2000.

[7] Richard J. Lipton. DNA solution of hard computational problems. *Science*, 268:542–545, 28, 1995.

[8] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.