Automata and Formal Languages — Homework 8

Due **Friday** 4th December 2015 (TA: Christopher Broadbent)

Exercise 8.1

In the lectures you saw an algorithm *kinter* that can be used to compute the intersection of two DDs. It takes as input states recognising $\langle L_1 \rangle$ and $\langle L_2 \rangle$ and outputs a state recognising $\langle L_1 \rangle \cap \langle L_2 \rangle$.

(i) Design an algorithm *kunion* that instead returns a state recognising $\langle L_1 \rangle \cup \langle L_2 \rangle$ and that thus can be used to compute the union of two DDs.

(ii) Design an algorithm *knot* for complementing a DD state

Exercise 8.2

The typical use case for BDDs is to represent (the meanings of) propositional formulae. Question 7.3 on the previous sheet already introduced you to this idea using finite-length DFA. However, as demonstrated in the lectures, BDDs can provide a more efficient representation since they avoid representing decisions that are irrelevant to the outcome.

Suppose that we have four Boolean variables $x_1, x_2, \ldots, x_4$. We write t for true and f for false. A *valuation* for the four variables is a word of length four belonging to $\{ \mathsf{t}, \mathsf{f} \}^4$. A valuation $b_1\, b_2\, b_3\, b_4$ sets the variable x_i to be b_i.

A *propositional formula* ϕ over $x_1, \ldots, x_4$ is formed by combining (possibly multiple copies of) $x_1, \ldots, x_4$ with the Boolean connectives $\wedge$, $\vee$ and $\neg$. We write

$$b_1 \cdots b_4 \vDash \phi$$

to mean that the formula ϕ is true when the variables are given the values defined by the valuation $b_1 \cdots b_4$. Let us define

$$\mathcal{L}(\phi) := \{ b_1 \cdots b_4 \mid b_1 \cdots b_4 \vDash \phi \}$$

For example,

$$\mathcal{L}(x_1 \wedge x_2) = \{ \mathsf{tt} b_3 b_4 \mid b_3, b_4 \in \{ \mathsf{t}, \mathsf{f} \} \}$$

A 'BDD for a formula ϕ' is a BDD recognising $\mathcal{L}(\phi)$.

(i) Give a BDD for the formula

$$(x_1 \quad \wedge \quad ((\neg x_2 \wedge \neg x_3) \ \vee \ (x_2 \wedge x_3)) \qquad \vee \qquad (\neg x_1 \quad \wedge \quad ((\neg x_2 \wedge x_4) \ \vee \ (x_2 \wedge \neg x_4))$$

[HINT: You may find it helpful to remember that a BDD is indeed a decision diagram and that you are making a decision on the value of the formula depending on the value of the variables!]

(ii) Give a BDD for the formula

$$(x_4 \quad \wedge \quad ((\neg x_3 \wedge \neg x_2) \ \vee \ (x_3 \wedge x_2)) \qquad \vee \qquad (\neg x_4 \quad \wedge \quad ((\neg x_3 \wedge x_1) \ \vee \ (x_3 \wedge \neg x_1))$$

The answers to (i) and (ii) should look quite different with one being bigger than the other. This is despite the fact that they both represent the same Boolean function given by the formula

$$(p \quad \wedge \quad ((\neg q \wedge \neg r) \ \vee \ (q \wedge r)) \qquad \vee \qquad (\neg p \quad \wedge \quad ((\neg q \wedge s) \ \vee \ (q \wedge \neg s))$$

where the variables p, q, r, s are given the respective names x_1, x_2, x_3, x_4 in the first case and the respective names x_4, x_3, x_2, x_1 in the second case. The way in which numbers are assigned to variables in a formula is called the *variable ordering* since it specifies the order in which the BDD considers each variable.

For (i) the variable ordering is $p < q < r < s$ (e.g. p is considered first because it is given the name x_1). For (ii) it is $s < r < q < p$.

The variable ordering can have a big impact on the size of a BDD. We say that a variable ordering is *optimal* for a formula ϕ if it is an ordering that yields a minimal BDD for ϕ.

(iii) Give an optimal variable ordering for the formula

$$((p \wedge s) \quad \vee \quad (\neg p \wedge \neg q)) \qquad \wedge \qquad ((q \wedge \neg r) \quad \vee \quad (\neg q \wedge r))$$

(iv) p, q, r, s can encode two two-bit integers $p\,q$ and $r\,s$. (You are free to choose whether to use a least-significant-bit-first or a most-significant-bit-first encoding).

Give a propositional formula expressing that $p\,q \geq r\,s$ (with respect to the integers that they encode). What is an optimal variable ordering for its BDD?

Exercise 8.3

Suppose that we have n propositional variables $x_1, \ldots, x_n$. A valuation $b_1 \cdots b_n \in \{ \mathsf{true}, \mathsf{false} \}^n$ over these n-variables assigns b_i to x_i. We define

$$\mathcal{L}_n(\phi) := \{ b_1 \cdots b_n \mid b_1 \cdots b_n \vDash_n \phi \}$$

where $b_1 \cdots b_n \vDash_n \phi$ means that ϕ is true when the variables are set as defined by the valuation $b_1 \cdots b_n$.

Let Φ be a finite set of propositional formulae over $x_1, \ldots, x_n$ (for some n). Let us say that a Φ-BDD is a quadruple of the form (m, Φ, f, B), where $m \geq n$, and B is a BDD with state set Q and $f : \Phi \to Q$ is a map such that $\mathcal{L}_m(\phi) = \mathcal{L}_n(f(\phi))$ for all $\phi \in \Phi$.

(i) Give an algorithm that takes a Φ-BDD (n, Φ, f, B) and an integer $i \in \mathbb{N}$ as input and returns a $(\Phi \cup \{ x_i \})$-BDD. (Remember that a variable x_i is just an atomic formula).

(ii) Describe algorithms that take a Φ-BDD (n, Φ, f, B) as input and return a $(\Phi \cup \{ \phi \})$-BDD for each of $\phi = \phi_1 \wedge \phi_2$, $\phi = \phi_1 \vee \phi_2$ and $\phi = \neg \phi_1$ for $\phi_1, \phi_2 \in \Phi$. You should make use of algorithms that you have already seen as subprocedures (either from the lectures or from exercise 8.1).

(iii) What is the significance of this question?

Exercise 8.4

Give sentences of first-order logic that define each of the following languages.

(i) The language of words over $\Sigma := \{ a, b \}$ that contain at least three as and at most two bs.

(ii) The language $(ab)^\star + (ba)^\star$

(iii) The language $aa(ab)^\star + bb(ba)^\star bb$

Now consider the language

$$L = \{ w \in (\{ 0, 1 \}^3)^* \mid \pi_1(w) \in \mathit{lsbf}(m_1), \pi_2(w) \in \mathit{lsbf}(m_2), \pi_3(w) \in \mathit{lsbf}(n) \text{ and } n = m_1 + m_2 \text{ (with } m_1, m_2, n \in \mathbb{N}) \}$$

This language L turns out *not* to be definable in first-order logic. Part (iv) thus gives you a bit of choice in the language that you define:

(iv) Give a first-oder sentence ϕ that defines a language $\widehat{L} \subseteq (\{ 0, 1 \}^4)^*$ such that

$$L = \{ (a_1, b_1, c_1) \cdots (a_k, b_k, c_k) \mid (a_1, b_1, c_1, d_1) \cdots (a_k, b_k, c_k, d_k) \in \widehat{L} \text{ for some } d_1, \ldots d_k \in \{ 0, 1 \} \}$$

[Hint : You have a free choice for what the fourth component contains. You might like to use it to store the carry bits.]

Solution 8.1

kunion(q_1, q_2)
Input: states q_1, q_2 recognizing $\langle L_1 \rangle, \langle L_2 \rangle$
Output: state recognizing $\langle L_1 \cup L_2 \rangle$

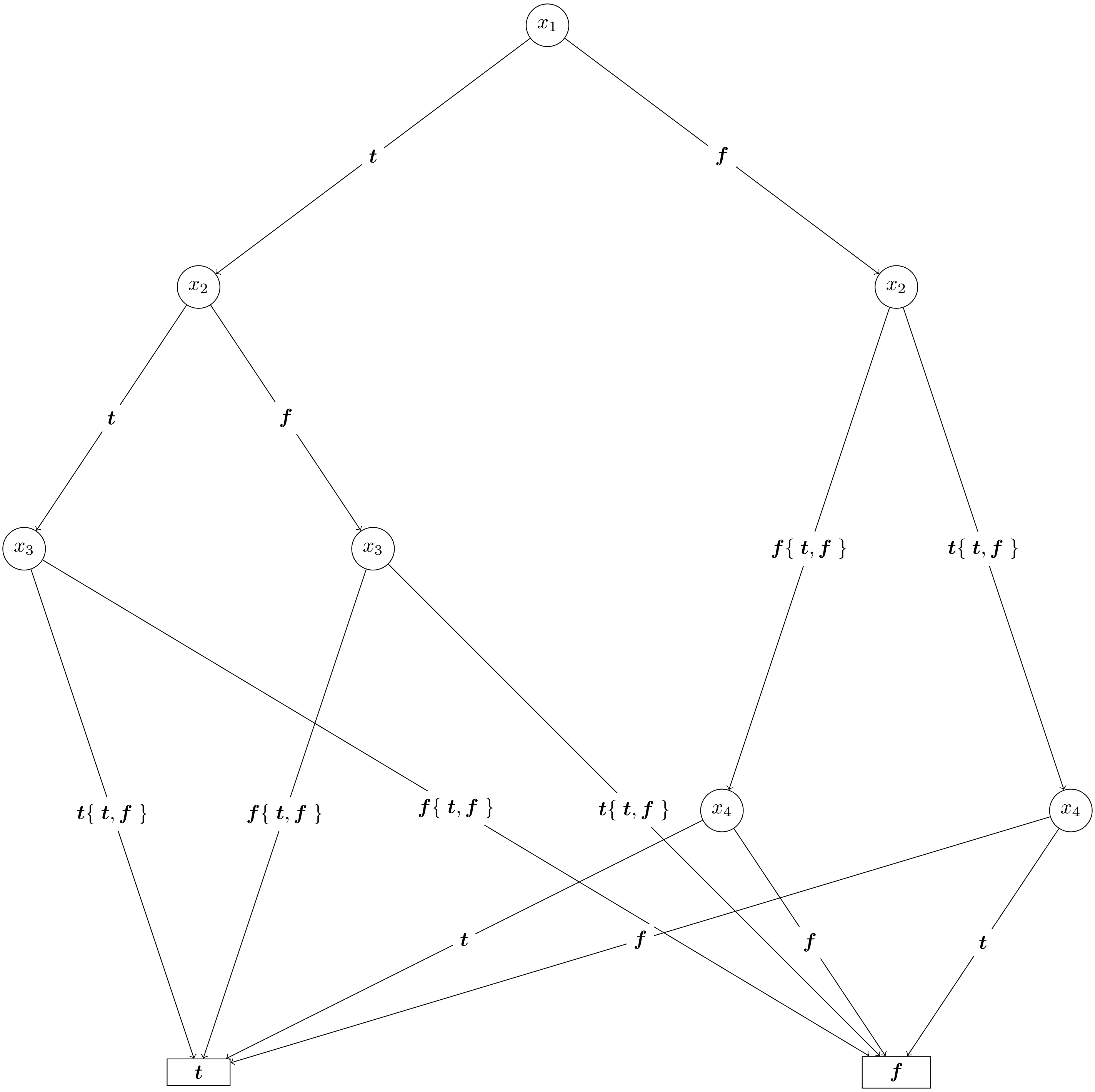
if $G(q_1, q_2)$ is not empty **then return** $G(q_1, q_2)$
if $q_1 = q_\emptyset$ **and** $q_2 = q_\emptyset$ **then return** $q_\emptyset$
if $q_1 \neq q_\emptyset$ **and** $q_2 \neq q_\emptyset$ **then**
  if $l_1 < l_2$ /* lengths of the kernodes for q_1, q_2 */ **then**
    for all $i = 1, \ldots, m$ **do** $r_i \leftarrow kunion(q_1, q_2^{a_i})$
    $G(q_1, q_2) \leftarrow \texttt{kmake}(l_2, r_1, \ldots, r_m)$
  else if $l_2 < l_1$ **then**
    for all $i = 1, \ldots, m$ **do** $r_i \leftarrow kunion(q_1^{a_i}, q_2)$
    $G(q_1, q_2) \leftarrow \texttt{kmake}(l_1, r_1, \ldots, r_m)$
  else /* $l_1 = l_2$ */
    for all $i = 1, \ldots, m$ **do** $r_i \leftarrow kunion(q_1^{a_i}, q_2^{a_i})$
    $G(q_1, q_2) \leftarrow \texttt{kmake}(l_1, r_1, \ldots, r_m)$
else if $q_1 \neq q_\emptyset$ **and** $q_2 = q_\emptyset$ **then**
  $G(q_1, q_2) \leftarrow q_1$
else /* $q_1 = q_\emptyset$ **and** $q_2 \neq q_\emptyset$ */
  $G(q_1, q_2) \leftarrow q_2$
return $G(q_1, q_2)$

knot(q)
Input: state q recognizing a kernel K
Output: state recognizing $\widehat{K}$

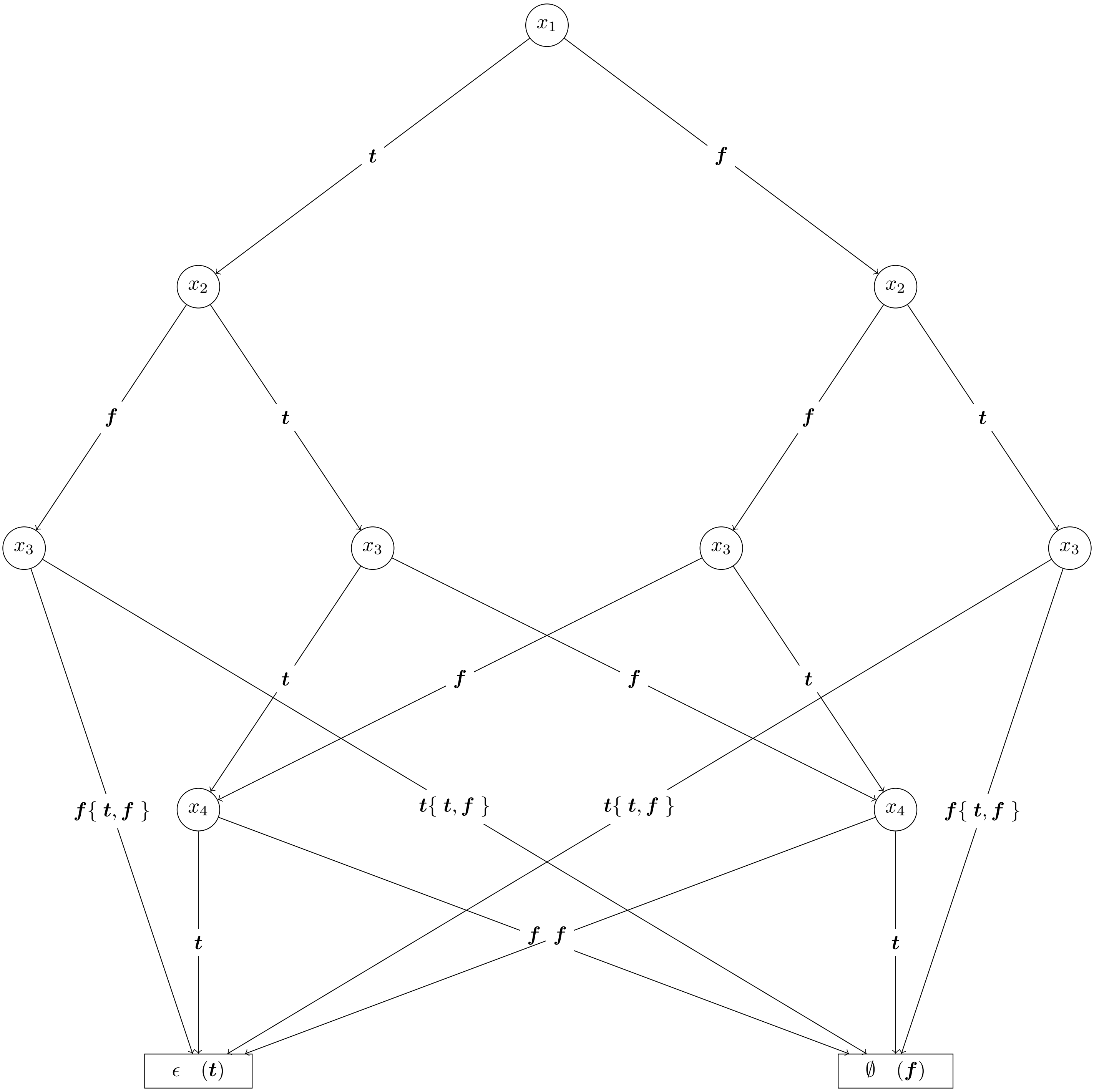
if $G(q)$ is not empty **then return** $G(q)$
if $q = q_\emptyset$ **then return** q_ϵ
else if $q = q_\epsilon$ **then return** $q_\emptyset$
else
  for all $i = 1, \ldots, m$ **do** $r_i \leftarrow knot(q^{a_i})$
  $G(q) \leftarrow \texttt{kmake}(r_1, \ldots, r_m)$
  return $G(q)$

Solution 8.2

(i)


x_1
$\boldsymbol{t}$
$\boldsymbol{f}$
x_2
x_2
$\boldsymbol{t}$
$\boldsymbol{f}$
$\boldsymbol{f}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
$\boldsymbol{t}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
x_3
x_3
$\boldsymbol{t}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
$\boldsymbol{f}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
$\boldsymbol{f}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
$\boldsymbol{t}\{\ \boldsymbol{t}, \boldsymbol{f}\ \}$
x_4
x_4
$\boldsymbol{t}$
$\boldsymbol{f}$
$\boldsymbol{f}$
$\boldsymbol{t}$
$\boldsymbol{t}$
$\boldsymbol{f}$

(ii)



(iii) $r < q < p < s$

To see this intuitively think about what one 'needs to remember' after checking the value of each variable and when one has discovered enough information to be able to ignore certain variables. To illustrate this, let us write how we decide the truth of the formula by checking the variables according to each ordering. The orderings are chosen so as to minimise what has to be 'remembered' between each step.

[1 BDD state at first level]:

- Check r and remember the value. [2 BDD states at second level]
- Check q and compare to r. If they are equal, we know immediately that the formula is false. If they are different, then we know the right-hand conjunct is true and just need to check the left-hand conjunct.

  The constraint on p is different depending on whether q is true or false, thus the third level must contain two states for each possible value of q. (Observe that the value of q would still need to be remembered even if we had chosen s to come before p). [2 BDD states at third level]

– Check p. If q was false and p is false, then we can immediately conclude the formula is true. If q was true and p is false, then we can immediately conclude the formula is false. Thus the only time when we still need to check s is when p is true (and then we can forget about q). [1 BDD state at fourth level]

– Check s. If it is true then the whole formula is true, otherwise it is false.

Thus the BDD will have 6 states (plus the two terminating states).

(iv) It will always be necessary to compare the most-significant bits. We might, however, get lucky and have different msbs, allowing us to ignore the least-significant bit. If the msbs are equal, the lsbs need to be checked.

Thus using LSBF encoding, an optimal variable ordering for $pq \geq rs$ will be $q < s < p < r$. (There are other orderings that are just as good, e.g. $s < q < p < r$).

Solution 8.3

Let us take the first (left) child of a node to correspond to $\boldsymbol{t}$ and the second (right) to correspond to $\boldsymbol{f}$.

(i) Simply return (n', Φ', f', B') where $n' = n, \Phi' = \Phi, f' = f, B' = B$ if $x_i \in \Phi$, otherwise return $n' := \max(n, i)$, $\Phi := \Phi \cup \{ x_i \}$, $f' = f \cup \{ x_i \mapsto q \}$, where q is the state returned by $\texttt{kmake}(i, q_\epsilon, q_\emptyset)$ (and B' is the resulting BDD).

(ii) If $\phi \in \Phi$, then we just return the input. Otherwise we return $(n, \Phi \cup \{ \phi \}, f \cup \{ q \mapsto \phi \}, B')$ where q is the state returned by, and B' is the resulting BDD produced by, running the appropriate algorithm on the nodes $f(\phi_1)$ (and $f(\phi_2)$). The appropriate algorithm is `kunion` for $\vee$, `kintersect` for $\wedge$ and `kcomp` for $\neg$.

(iii) This shows how BDDs provide a *compositional* representation of Boolean formulae. One can reuse the work done to construct BDDs representing simpler formulae to construct BDDs representing a more complex formula made up from such simpler components.

Solution 8.4

(i) Given a formula $\phi(x)$ with a single free variable together with $n \in \mathbb{N}$, we can define the formulae of the form

$$\exists_{\geq n} x.\phi(x) \quad \text{and} \quad \forall_{\leq n} x.\phi(x)$$

asserting respectively that there are *at least* n positions (in a word) satisfying ϕ, and that there are *at most* n positions (in a word) satisfying ϕ.

$$\exists_{\geq n} x.\phi(x) := \exists x_1. \cdots \exists x_n. \left(\bigwedge_{1 \leq i < j \leq n} \neg x_i = x_j \wedge \bigwedge_{1 \leq i \leq n} \phi(x_i) \right)$$

$$\forall_{\leq n} x.\phi(x) := \forall x_1. \cdots \forall x_n. \forall y. \left(\left(\bigwedge_{1 \leq i \leq n} \phi(x_i) \wedge \phi(y) \right) \rightarrow \left(\bigvee_{1 \leq i \leq n} x_i = y \right) \right)$$

Strictly speaking you have not been given equality in the lectures. However equality is definable from the $<$ primitive: $x = y$ can be expressed by $\neg x < y \;\wedge\; \neg y < x$.

The following formula then answers the present question:

$$\exists_{\geq 3} Q_a(x) \quad \wedge \quad \forall_{\leq 2} Q_b(x)$$

(ii) You have seen in the lectures that we can define $x = y + k$ in the logic, where k is any positive integer. You have also seen that *First* and *Last* predicates are definable, defining the last and first position in the word.

It is also helpful to remember that the domain of the empty-word is, well, empty. This means that the empty word satisfies *all* formulae of the form $\forall x.\phi$ irrespective of what the formula ϕ may be (ϕ could even be unsatisfiable).

The language $(ab)^*$ can thus be defined by the sentence:

$$\phi_{(ab)^*} := \forall x.\forall y.(y = x{+}1 \rightarrow ((Q_a(x) \rightarrow Q_b(x)) \wedge (Q_b(x) \rightarrow Q_a(x)) \quad \wedge \quad \forall x.(\mathit{First}(x) \rightarrow Q_a(x)) \wedge (\mathit{Last}(x) \rightarrow Q_b(x)))$$

Likewise we can define

$$\phi_{(ba)^*} := \forall x.\forall y.(y = x{+}1 \rightarrow ((Q_a(x) \rightarrow Q_b(x)) \wedge (Q_b(x) \rightarrow Q_a(x)) \quad \wedge \quad \forall x.((\mathit{First}(x) \rightarrow Q_b(x)) \wedge (\mathit{Last}(x) \rightarrow Q_a(x)))$$

So the required sentence is $\phi_{(ab)^*} \vee \phi_{(ba)^*}$.

(iii) For each formula ϕ that does not contain x free, and for each formula $\psi(x)$ containing a free variable x, let us define the formula $\phi[x : \psi(x)]$ by the following

$$Q_a(y)[x : \psi(x)] := Q_a(y) \quad y_1 < y_2[x : \psi(x)] := y_1 < y_2 \quad (\phi_1 \wedge \phi_2)[x : \psi(x)] := (\phi_1[x : \psi(x)] \wedge \phi_2[: \psi(x)])$$

$$(\neg\phi)[x : \psi(x)] := \neg(\phi[x : \psi(x)]) \qquad \exists y.\phi[x : \psi(x)] := \exists y.(\psi(y) \wedge \phi)$$

($\forall$ is defined in terms of $\exists$, but if we were to treat it as primitive we would have $\forall y.\phi[x : \psi(x)] := \forall y.(\psi(y) \rightarrow \phi)$.)

The formula $\phi[x : \psi(x)]$ is the formula ϕ with *quantifiers restricted to positions* satisfying $\psi(x)$. So given a word w we can construct a new word from w' made up of positions i in w such that $\psi(i)$. Then w satisfies $\phi[x : \psi(x)]$ iff w' satisfies ϕ.

Thus the language $aa(ab)^*$ is defined by the following sentence

$$\phi_{aa(ab)^*} := \exists x.\exists y.(\mathit{First}(x) \wedge y = x + 1 \wedge Q_a(x) \wedge Q_a(y) \wedge \phi_{(ab)^*}[z : y < z]$$

and the language $bb(ba)^*bb$ is defined by

$$\phi_{bb(ba)^*bb} := \exists x.\exists y.\exists x'.\exists y'.(\mathit{First}(x) \ \wedge \ y = x + 1 \ \wedge \ \mathit{Last}(y') \ \wedge \ y' = x' + 1 \ \wedge \ Q_b(x) \ \wedge \ Q_b(y) \ \wedge \ Q_b(x') \ \wedge \ Q_b(y') \\ \wedge \ \phi_{(ba)^*}[z : y < z \wedge z < y'])$$

We can then take the required sentence to be $\phi_{aa(ab)^*} \vee \phi_{bb(ba)^*bb}$.

(iv) We follow the hint and use the fourth position to represent the carry-bit of an adder.

We define eight auxiliary predicates A_b, B_b, C_b and S_b for each $b \in \mathbb{B} = \{ 0, 1 \}$. The A predicates are used to describe the bits making up the binary representation of the number m_1 and B for m_2. The C predicates will be used to define the bits making up the *fourth number* (C stands for 'carry'), and the S predicates will be used to define the bits making up the *sum* $m_1 + m_2$ stored in the third row (adding the A and B bits together with the carry C bit).

Explicitly we can define:

$$A_b(x) := \bigvee_{b_1,b_2,b_3 \in \mathbb{B}} Q_{(b,b_1,b_2,b_3)}(x) \qquad B_b(x) := \bigvee_{b_1,b_2,b_3 \in \mathbb{B}} Q_{(b_1,b,b_2,b_3)}(x)$$

$$S_b(x) := \bigvee_{b_1,b_2,b_3 \in \mathbb{B}} Q_{(b_1,b_2,b,b_3)}(x) \qquad C_b(x) := \bigvee_{b_1,b_2,b_3 \in \mathbb{B}} Q_{(b_1,b_2,b_3,b)}(x)$$

It will also be useful to have the following formulae:

$$\mathit{CarryOne}(x) := \exists y.(y = x + 1 \wedge C_1(y))$$

and

$$\mathit{NoCarry}(x) := \forall y.(y = x + 1 \rightarrow C_0(y))$$

The first formula asserts that a 1 should be carried to the position following x whilst the second formula asserts that nothing should be carried. (Note that in the first case we must assert the existence of the next position, since a next position is a prerequisite for carrying to the next position. In the second case we can 'avoid carrying' either by having a 0 carry bit in the next position, or else the next position not existing at all).

Let us consider how we would describe a 'half-adder' in the logic, which sums a single bit from the first two numbers together with the carry bit to produce a result together with a fresh carry bit.

$$\phi_{\mathit{half}}(x) := \bigwedge \left(\begin{array}{lcl} \bigvee_{[\text{ at least two of } b_1,b_2,b_3 \in \mathbb{B} \text{ are } 1]} (A_{b_1}(x) \wedge B_{b_2}(x) \wedge C_{b_3}(x)) & \rightarrow & \mathit{CarryOne}(x) \\ \bigvee_{[\text{ at most one of } b_1,b_2,b_3 \in \mathbb{B} \text{ are } 1]} (A_{b_1}(x) \wedge B_{b_2}(x) \wedge C_{b_3}(x)) & \rightarrow & \mathit{CarryZero}(x) \\ \bigvee_{[\text{an even number of } b_1,b_2,b_3 \in \mathbb{B} \text{ are } 1]} (A_{b_1}(x) \wedge B_{b_2}(x) \wedge C_{b_3}(x)) & \rightarrow & S_0(x) \\ \bigvee_{[\text{an odd number of } b_1,b_2,b_3 \in \mathbb{B} \text{ are } 1]} (A_{b_1}(x) \wedge B_{b_2}(x) \wedge C_{b_3}(x)) & \rightarrow & S_1(x) \end{array} \right)$$

We can turn our half-adder into a 'full adder' by asserting that (i) every position in the word conforms to the half-adder, (ii) if there is a first-postion in the word (i.e. if the word is non-empty) then the carry-bit there is 0, and (iii) there is no 'dangling carry bit' (every position with a 1 carry-bit has a successor).

$$\forall x.\phi_{\mathit{half}}(x) \quad \wedge \quad \forall x.(\mathit{First}(x) \rightarrow C_0(x)) \quad \wedge \quad \forall x.(C_1(x) \rightarrow \exists y.(y = x + 1))$$