Equality Logic and Uninterpreted Functions

3.1 Introduction

This chapter introduces the theory of equality, also known by the name equality logic. Equality logic can be thought of as propositional logic where the atoms are equalities between variables over some infinite type or between variables and constants. As an example, the formula \((y = z \lor \neg(x = z) \land x = 2)\) is a well-formed equality logic formula, where \(x, y, z \in \mathbb{R}\) (\(\mathbb{R}\) denotes the reals). An example of a satisfying assignment is \(\{x \mapsto 2, y \mapsto 2, z \mapsto 0\}\).

**Definition 3.1 (equality logic).** An equality logic formula is defined by the following grammar:

\[
\begin{align*}
\text{formula} & : \text{formula} \land \text{formula} \mid \neg \text{formula} \mid (\text{formula}) \mid \text{atom} \\
\text{atom} & : \text{term} = \text{term} \\
\text{term} & : \text{identifier} \mid \text{constant}
\end{align*}
\]

where the identifiers are variables defined over a single infinite domain such as the Reals or Integers.\(^1\) Constants are elements from the same domain as the identifiers.

3.1.1 Complexity and Expressiveness

The satisfiability problem for equality logic is NP-complete. We leave the proof of this claim as an exercise (Problem 4.7 in Chap. 4). The fact that both equality logic and propositional logic are NP-complete implies that they can model the same decision problems (with not more than a polynomial difference in the number of variables). Why should we study both, then?

For two main reasons: convenience of modeling, and efficiency. It is more natural and convenient to use equality logic for modeling certain problems

\(^1\) The restriction to a single domain (also called a single type or a single sort) is not essential. It is introduced for the sake of simplicity of the presentation.
than to use propositional logic, and vice versa. As for efficiency, the high-
level structure in the input equality logic formula can potentially be used to
make the decision procedure work faster. This information may be lost if the
problem is modeled directly in propositional logic.

3.1.2 Boolean Variables

Frequently, equality logic formulas are mixed with Boolean variables. Never-
theless, we shall not integrate them into the definition of the theory, in order
to keep the description of the algorithms simple. Boolean variables can easily
be eliminated from the input formula by replacing each such variable with an
equality between two new variables. But this is not a very efficient solution.
As we progress in this chapter, it will be clear that it is easy to handle Boolean
variables directly, with only small modifications to the various decision pro-
cedures. The same observation applies to many of the other theories that we
consider in this book.

3.1.3 Removing the Constants: A Simplification

**Theorem 3.2.** Given an equality logic formula $\varphi^E$, there is an algorithm that generates an equisatisfiable formula (see Definition 1.9) $\varphi^{E'}$ without constants, in polynomial time.

**Algorithm 3.1.1: REMOVE-CONSTANTS**

**Input:** An equality logic formula $\varphi^E$ with constants $c_1, \ldots, c_n$  
**Output:** An equality logic formula $\varphi^{E'}$ such that $\varphi^{E'}$ and $\varphi^E$ are equisatisfiable and $\varphi^{E'}$ has no constants

1. $\varphi^{E'} := \varphi^E$.
2. In $\varphi^{E'}$, replace each constant $c_i$, $1 \leq i \leq n$, with a new variable $C_{c_i}$.
3. For each pair of constants $c_i, c_j$ such that $1 \leq i < j \leq n$, add the constraint $C_{c_i} \neq C_{c_j}$ to $\varphi^{E'}$.

Algorithm 3.1.1 eliminates the constants from a given formula by replacing them with new variables. Problem 3.2, and, later, Problem 4.4, focus on this procedure. Unless otherwise stated, we assume from here on that the input equality formulas do not have constants.

3.2 Uninterpreted Functions

Equality logic is far more useful if combined with uninterpreted functions. Uninterpreted functions are used for abstracting, or generalizing, theorems.
Unlike other function symbols, they should not be interpreted as part of a model of a formula. In the following formula, for example, $F$ and $G$ are uninterpreted, whereas the binary function symbol “+” is interpreted as the usual addition function:

$$F(x) = F(G(y)) \lor x + 1 = y. \quad (3.1)$$

**Definition 3.3 (equality logic with uninterpreted functions (EUF)).** An equality logic formula with uninterpreted functions and uninterpreted predicates\(^2\) is defined by the following grammar:

- **formula**: formula \& formula | \neg formula | (formula) | atom
- **atom**: term = term | predicate-symbol (list of terms)
- **term**: identifier | function-symbol (list of terms)

We generally use capital letters to denote uninterpreted functions, and use the superscript UF to denote EUF formulas.

**Aside: The Logic Perspective**

To explain the meaning of uninterpreted functions from the perspective of logic, we have to go back to the notion of a theory, which was explained in Sect. 1.4. Recall the set of axioms (1.35), and that in this chapter we refer to the quantifier-free fragment.

Only a single additional axiom (an axiom scheme, actually) is necessary in order to extend equality logic to EUF. For each $n$-ary function symbol, $n > 0$,

$$\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n, \quad \land_i t_i = t'_i \implies F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n) \quad \text{(CONGRUENCE)}, \quad (3.2)$$

where $t_1, \ldots, t_n, t'_1, \ldots, t'_n$ should be instantiated with terms that appear as arguments of uninterpreted functions in the formula. A similar axiom can be defined for uninterpreted predicates.

Thus, whereas in theories where the function symbols are interpreted there are axioms to define their semantics – what we want them to mean – in a theory over uninterpreted functions, the only restriction we have over a satisfying interpretation is that imposed by functional consistency, namely the restriction imposed by the (CONGRUENCE) rule.

**3.2.1 How Uninterpreted Functions Are Used**

Replacing functions with uninterpreted functions in a given formula is a common technique for making it easier to reason about (e.g., to prove its validity).

\(^2\) From here on, we refer only to uninterpreted functions. Uninterpreted predicates are treated in a similar way.
At the same time, this process makes the formula *weaker*, which means that it can make a valid formula invalid. This observation is summarized in the following relation, where $\varphi^{\text{UF}}$ is derived from a formula $\varphi$ by replacing some or all of its functions with uninterpreted functions:

$$\models \varphi^{\text{UF}} \implies \models \varphi.$$  \hfill (3.3)

Uninterpreted functions are widely used in calculus and other branches of mathematics, but in the context of reasoning and verification, they are mainly used for simplifying proofs. Under certain conditions, uninterpreted functions let us reason about systems while ignoring the semantics of some or all functions, assuming they are not necessary for the proof. What does it mean to ignore the semantics of a function? (A formal explanation is briefly given in the aside on p. 61.) One way to look at this question is through the axioms that the function can be defined by. Ignoring the semantics of the function means that an interpretation need not satisfy these axioms in order to satisfy the formula. The only thing it needs to satisfy is an axiom stating that the uninterpreted function, like any function, is *consistent*, i.e., given the same inputs, it returns the same outputs. This is the requirement of **functional consistency** (also called **functional congruence**):

**Functional consistency**: Instances of the same function return the same value if given equal arguments.

There are many cases in which the formula of interest is valid regardless of the interpretation of a function. In these cases, uninterpreted functions simplify the proof significantly, especially when it comes to mechanical proofs with the aid of automatic theorem provers.

Assume that we have a method for checking the validity of an EUF formula. Relying on this assumption, the basic scheme for using uninterpreted functions is the following:

1. Let $\varphi$ denote a formula of interest that has interpreted functions. Assume that a validity check of $\varphi$ is too hard (computationally), or even impossible.
2. Assign an uninterpreted function to each interpreted function in $\varphi$. Substitute each function in $\varphi$ with the uninterpreted function to which it is mapped. Denote the new formula by $\varphi^{\text{UF}}$.
3. Check the validity of $\varphi^{\text{UF}}$. If it is valid, return “$\varphi$ is valid” (this is justified by (3.3)). Otherwise, return “don’t know”.

The transformation in step 2 comes at a price, of course, as it loses information. As mentioned earlier, it causes the procedure to be incomplete, even if the original formula belongs to a decidable logic. When there exists a decision procedure for the input formula but it is too computationally hard to solve, one can design a procedure in which uninterpreted functions are gradually substituted back to their interpreted versions. We shall discuss this option further in Sect. 3.4.
3.2.2 An Example: Proving Equivalence of Programs

As a motivating example, consider the problem of proving the equivalence of the two C functions shown in Fig. 3.1. More specifically, the goal is to prove that they return the same value for every possible input `in`.

```c
int power3(int in) {
    int i, out_a;
    out_a = in;
    for (i = 0; i < 2; i++)
        out_a = out_a * in;
    return out_a;
}

int power3_new(int in) {
    int out_b;
    out_b = (in * in) * in;
    return out_b;
}
```

Fig. 3.1. Two C functions. The proof of their equivalence is simplified by replacing the multiplications ("*"') in both programs with uninterpreted functions

In general, proving the equivalence of two programs is undecidable, which means that there is no sound and complete method to prove such an equivalence. In the present case, however, equivalence can be decided. A key observation about these programs is that they have only bounded loops, and therefore it is possible to compute their input/output relations. The derivation of these relations from these two programs can be done as follows:

1. Remove the variable declarations and "return" statements.
2. Unroll the `for` loop.
3. Replace the left-hand side variable in each assignment with a new auxiliary variable.
4. Wherever a variable is read (referred to in an expression), replace it with the auxiliary variable that replaced it in the last place where it was assigned.
5. Conjoin all program statements.

These operations result in the two formulas $\varphi_a$ and $\varphi_b$, which are shown in Fig. 3.2.

It is left to show that these two I/O relations are actually equivalent, that is, to prove the validity of

$$\varphi_a \land \varphi_b \implies out_{2,a} = out_{0,b}.$$  \hfill (3.4)

The undecidability of program verification and program equivalence is caused by unbounded memory usage, which does not occur in this example.

A generalization of this form of translation to programs with "if" branches and other constructs is known as static-single-assignment (SSA). SSA is used in most optimizing compilers and can be applied to the verification of programs with bounded loops in popular programming languages such as C (see [107]). See also Example 1.25.
Uninterpreted functions can help in proving the equivalence of the programs (a) and (b), following the general scheme suggested in Sect. 3.2.1. The motivation in this case is computational: deciding formulas with multiplication over, for example, 32-bit variables is notoriously hard. Replacing the multiplication symbol with uninterpreted functions can solve the problem.

Figure 3.3 presents $\varphi_{a}^{\text{UF}}$ and $\varphi_{b}^{\text{UF}}$, which are $\varphi_{a}$ and $\varphi_{b}$ after the multiplication function has been replaced with a new uninterpreted function $G$. Similarly, if we also had addition, we could replace all of its instances with another uninterpreted function, say $F$. Instead of validating (3.4), we can now attempt to validate

$$\varphi_{a}^{\text{UF}} \land \varphi_{b}^{\text{UF}} \implies \text{out}2_{,a} = \text{out}0_{,b}.$$ \hspace{1cm} (3.5)

Alternative methods to prove the equivalence of these two programs are discussed in the aside on p. 65. Other examples of the use of uninterpreted functions are presented in Sect. 3.5.

### 3.3 From Uninterpreted Functions to Equality Logic

Luckily, we do not need to examine all possible interpretations of an uninterpreted function in a given EUF formula in order to know whether it is valid. Instead, we rely on the strongest property that is common to all functions, namely functional consistency.$^5$ Relying on this property, we can reduce the decision problem of EUF formulas to that of deciding equality logic. We shall

---

$^5$ Note that the term function here refers to the mathematical definition. The situation is more complicated when considering functions in programming languages.
Aside: Alternative Decision Procedures

The procedure in Sect. 3.2.2 is not the only way to automatically prove the equivalence of programs (a) and (b), of course. In this case, substitution is sufficient: by simply substituting \textit{out2\_a} by \textit{out1\_a * in}, \textit{out1\_a} by \textit{out0\_a * in}, and \textit{out0\_a} by \textit{in} in \varphi_a, we can quickly (and automatically) prove (3.4), as we obtain syntactically equal expressions. However, there are many cases where such substitution is not efficient, as it can increase the size of the formula exponentially. It is also possible that substitution alone may be insufficient to prove equivalence. Consider, for example, the two functions \textit{power3\_con} and \textit{power3\_con\_new}:

```c
int power3\_con (int in, int con) {
    int i, out\_a;
    out\_a = in;
    for (i = 0; i < 2; i++)
        out\_a = con?out\_a * in : out\_a;
    return out\_a;
}
```

(a)

```c
int power3\_con\_new (int in, int con) {
    int out\_b;
    out\_b = con?((con?in*in : in) * in) : in;
    return out\_b;
}
```

(b)

After substitution, we obtain two expressions,

\[
\text{out\_a} = \text{con}\?((\text{con}\?\text{in}\*\text{in} : \text{in})\*\text{in}) : (\text{con}\?\text{in}\*\text{in} : \text{in})
\]

(3.6)

and

\[
\text{out\_b} = \text{con}\?((\text{in}\*\text{in})\*\text{in} : \text{in})
\]

(3.7)

corresponding to the two functions. Not only are these two expressions not syntactically equivalent, but also the first expression grows exponentially with the number of iterations.

Another possible way to prove equivalence is to rely on the fact that the loops in the above programs are finite, and that the variables, as in any C program, are of finite type (e.g., integers are typically represented using 32-bit bit vectors – see Chap. 6). Therefore, the set of states reachable by the two programs can be represented and searched. This method can almost never compete, however, with decision procedures for equality logic and uninterpreted functions in terms of efficiency. There is a tradeoff, then, between efficiency and completeness.
see two possible reductions, **Ackermann’s reduction** and **Bryant’s reduction**, both of which enforce functional consistency. The former is somewhat more intuitive to understand, but also imposes certain restrictions on the decision procedures that can be used to solve it, unlike the latter. The implications of the differences between the two methods are explained in Sect. 4.6.

In the discussion that follows, for the sake of simplicity, we make several assumptions regarding the input formula: it has a single uninterpreted function, with a single argument, and no two instances of this function have the same argument. The generalization of the reductions is rather straightforward, as the examples later on demonstrate.

### 3.3.1 Ackermann’s Reduction

Ackermann’s reduction (Algorithm 3.3.1) adds explicit constraints to the formula in order to enforce the functional consistency requirement stated above. The algorithm reads an EUF formula \( \varphi^{UF} \) that we wish to validate, and transforms it to an equality logic formula \( \varphi^E \) of the form

\[
\varphi^E := FC^E \implies flat^E, \tag{3.8}
\]

where \( FC^E \) is a conjunction of functional-consistency constraints, and \( flat^E \) is a flattening of \( \varphi^{UF} \), i.e., a formula in which each unique function instance is replaced with a corresponding new variable.

**Example 3.4.** Consider the formula

\[
(x_1 \neq x_2) \lor (F(x_1) = F(x_2)) \lor (F(x_1) \neq F(x_3)), \tag{3.9}
\]

which we wish to reduce to equality logic using Algorithm 3.3.1.

After assigning indices to the instances of \( F \) (for this example, we assume that this is done from left to right), we compute \( flat^E \) and \( FC^E \) accordingly:

\[
flat^E := (x_1 \neq x_2) \lor (f_1 = f_2) \lor (f_1 \neq f_3), \tag{3.10}
\]

\[
FC^E := (x_1 = x_2 \implies f_1 = f_2) \land
\quad (x_1 = x_3 \implies f_1 = f_3) \land
\quad (x_2 = x_3 \implies f_2 = f_3). \tag{3.11}
\]

Equation (3.9) is valid if and only if the resulting equality formula is valid:

\[
\varphi^E := FC^E \implies flat^E. \tag{3.12}
\]

such as C or JAVA. Functional consistency is guaranteed in that case only if we consider all the data that the function may read (including global variables, static variables, and data read from the environment) as argument of the function, and provided that the program is single-threaded.
Algorithm 3.3.1: Ackermann’s-reduction

Input: An EUF formula $\varphi^{\text{UF}}$ with $m$ instances of an uninterpreted function $F$

Output: An equality logic formula $\varphi^{E}$ such that $\varphi^{E}$ is valid if and only if $\varphi^{\text{UF}}$ is valid

1. Assign indices to the uninterpreted-function instances from subexpressions outwards. Denote by $F_i$ the instance of $F$ that is given the index $i$, and by $\text{arg}(F_i)$ its single argument.

2. Let $\text{flat}^{E} \doteq T(\varphi^{\text{UF}})$, where $T$ is a function that takes an EUF formula (or term) as input and transforms it to an equality formula (or term, respectively) by replacing each uninterpreted-function instance $F_i$ with a new term-variable $f_i$ (in the case of nested functions, only the variable corresponding to the most external instance remains).

3. Let $FC^{E}$ denote the following conjunction of functional consistency constraints:

$$FC^{E} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} (T(\text{arg}(F_i)) = T(\text{arg}(F_j))) \implies f_i = f_j.$$  

4. Let

$$\varphi^{E} := FC^{E} \implies \text{flat}^{E}.$$  

Return $\varphi^{E}$.

In the next example, we go back to our running example for this chapter, and transform it to equality logic.

Example 3.5. Recall our main example. We left it in Fig. 3.3 after adding the uninterpreted-function symbol $G$. Now, using Ackermann’s reduction, we can reduce it into an equality logic formula. This example also demonstrates how to generalize the reduction to functions with several arguments: only if all arguments of a pair of function instances are the same (pairwise), the return value of the function is forced to be the same.

Our example has four instances of the uninterpreted function $G$,

$$G(out0\_a, in),\; G(out1\_a, in),\; G(in, in),\; \text{and } G(G(in, in), in),$$

which we number in this order. On the basis of (3.5), we compute $\text{flat}^{E}$, replacing each uninterpreted-function symbol with the corresponding variable:

$$\text{flat}^{E} := \left( \left( out0\_a = in \wedge out1\_a = g1 \wedge out2\_a = g2 \right) \Rightarrow out0\_b = g4 \right)$$

(3.13)
Aside: Checking the Satisfiability of $\varphi^\text{UF}$

Ackermann’s reduction was defined above for checking the validity of $\varphi^\text{UF}$. It tells us that we need to check for the validity of $\varphi^E := FC^E \implies \text{flat}^E$

or, equivalently, check that $\neg \varphi^E := FC^E \land \neg \text{flat}^E$ is unsatisfiable. This is important in our case, because all the algorithms that we shall see later check for satisfiability of formulas, not for their validity. Thus, as a first step we need to negate $\varphi^E$.

What if we want to check for the satisfiability of $\varphi^\text{UF}$? The short answer is that we need to check for the satisfiability of

$$\varphi^E := FC^E \land \text{flat}^E.$$  

This is interesting. Normally, if we check for the satisfiability or validity of a formula, this corresponds to checking for the satisfiability of the formula or of its negation, respectively. Thus, we could expect that checking the satisfiability of $\varphi^\text{UF}$ is equivalent to checking satisfiability of $(FC^E \implies \text{flat}^E)$. However, this is not the same as the above equation. So what has happened here? The reason for the difference is that we check the satisfiability of $\varphi^\text{UF}$ before the reduction. This means that we can use Ackermann’s reduction to check the validity of $\neg \varphi^\text{UF}$. The functional-consistency constraints $FC^E$ remain unchanged whether we check $\varphi^\text{UF}$ or its negation $\neg \varphi^\text{UF}$. Thus, we need to check the validity of $FC^E \implies \neg \text{flat}^E$, which is the same as checking the satisfiability of $FC^E \land \text{flat}^E$, as stated above.

The functional-consistency constraints are given by

$$FC^E := \begin{align*}
(out0_a = out1_a \land in = in) \implies g_1 = g_2) \land \\
(out0_a = in \land in = in) \implies g_1 = g_3) \land \\
(out1_a = g_3 \land in = in) \implies g_1 = g_4) \land \\
(out1_a = in \land in = in) \implies g_2 = g_3) \land \\
(out1_a = g_3 \land in = in) \implies g_2 = g_4) \land \\
(in = g_3 \land in = in) \implies g_3 = g_4) .
\end{align*}$$  

(3.14)

The resulting equality formula is $FC^E \implies \text{flat}^E$, which we need to validate.

The reader may observe that most of these constraints are in fact redundant. The validity of the formula depends on $G(out0_a, in)$ being equal to $G(in, in)$, and $G(out1_a, in)$ being equal to $G(G(in, in), in)$. Hence, only the second and fifth constraints in (3.14) are necessary. In practice, such observations are important because the quadratic growth in the number of functional-consistency constraints may become a bottleneck. When comparing two systems, as in this case, it is frequently possible to detect in polynomial time large sets of constraints that can be removed without affecting the validity of the formula. More details of this technique can be found in [156].

Finally, we consider the case in which there is more than one function symbol.
Example 3.6. Consider now the following formula, which we wish to validate:

\[
x_1 = x_2 \implies F(F(G(x_1))) = F(F(G(x_2))).
\] (3.15)

We index the function instances from the inside out (from subexpressions outwards) and compute the following:

\[
\text{flat}^E := x_1 = x_2 \implies f_2 = f_4
\] (3.16)

\[
\text{FC}^E := x_1 = x_2 \implies g_1 = g_2 \land
g_1 = f_1 \implies f_1 = f_2 \land
f_1 = g_2 \implies f_1 = f_3 \land
f_1 = f_3 \implies f_1 = f_4 \land
f_1 = g_2 \implies f_2 = f_3 \land
f_1 = f_3 \implies f_2 = f_4 \land
f_2 = f_3 \implies f_3 = f_4.
\] (3.17)

Then, again,

\[
\phi^E := \text{FC}^E \implies \text{flat}^E.
\] (3.18)

From these examples, it is clear how to generalize Algorithm 3.3.1 to multiple uninterpreted functions. We leave this and other extensions as an exercise (Problem 3.3).

3.3.2 Bryant’s Reduction

Bryant’s reduction (Algorithm 3.3.2) has the same goal as Ackermann’s reduction: to transform EUF formulas to equality logic formulas, such that both are equivalent. To check the satisfiability of \(\phi^{\text{UP}}\) rather than the validity, we return \(\text{FC}^E \land \text{flat}^E\) in the last step.

The semantics of the case expression used in step 3 is such that its value is determined by the first condition that is evaluated to \text{true}. Its translation to an equality logic formula, assuming that the argument of \(F_i\) is a variable \(x_i\) for all \(i\), is given by

\[
\bigvee_{j=1}^i (F^*_i = f_j \land (x_j = x_i) \land \bigwedge_{k=1}^{j-1} (x_k \neq x_i)).
\] (3.22)

Example 3.7. Given the case expression

\[
F^*_3 = \begin{cases} 
\text{case } x_1 = x_3 : f_1 \\
\text{case } x_2 = x_3 : f_2 \\
\text{true} : f_3
\end{cases},
\] (3.23)
Algorithm 3.3.2: Bryant’s-reduction

**Input:** An EUF formula \( \varphi^{\text{UP}} \) with \( m \) instances of an uninterpreted function \( F \)

**Output:** An equality logic formula \( \varphi^{\text{E}} \) such that \( \varphi^{\text{E}} \) is valid if and only if \( \varphi^{\text{UP}} \) is valid

1. Assign indices to the uninterpreted-function instances from subexpressions outwards. Denote by \( F_i \) the instance of \( F \) that is given the index \( i \), and by \( \text{arg}(F_i) \) its single argument.
2. Let \( \text{flat}^E = T^*(\varphi^{\text{UP}}) \), where \( T^* \) is a function that takes an EUF formula (or term) as input and transforms it to an equality formula (or term, respectively) by replacing each uninterpreted-function instance \( F_i \) with a new term-variable \( F_i^* \) (in the case of nested functions, only the variable corresponding to the most external instance remains).
3. For \( i \in \{1, \ldots, m\} \), let \( f_i \) be a new variable, and let \( F_i^* \) be defined as follows:
\[
F_i^* := \begin{cases} 
\text{case } T^*(\text{arg}(F_i^*)) = T^*(\text{arg}(F_{i-1}^*)) : f_1 \\
\quad \vdots \\
\quad T^*(\text{arg}(F_{i-1}^*)) = T^*(\text{arg}(F_i^*)) : f_{i-1} \\
\quad \text{true} : f_i 
\end{cases}.
\] (3.19)

Finally, let
\[
FC^E := \bigwedge_{i=1}^m F_i^*.
\] (3.20)

4. Let
\[
\varphi^E := FC^E \implies \text{flat}^E.
\] (3.21)

Return \( \varphi^E \).

Its equivalent equality logic formula is given by
\[
(F_3^* = f_1 \land x_1 = x_3) \lor \\
(F_3^* = f_2 \land x_2 = x_3 \land x_1 \neq x_3) \lor \\
(F_3^* = f_3 \land x_1 \neq x_3 \land x_2 \neq x_3).
\] (3.24)

The differences between the two reduction schemes are:

1. Step 1 in Bryant’s reduction requires a certain order when indices are assigned to function instances. Such an order is not required in Ackermann’s reduction.
2. Step 2 in Bryant’s reduction replaces function instances with \( F^* \) variables rather than with \( f \) variables. The \( F^* \) variables should be thought of simply as macros, or placeholders, which means that they are used only for
simplifying the writing of the formula. We can do without them if we remove $FC^E$ from the formula altogether and substitute them in $flat^E$ with their definitions. The reason that we maintain them is to make the presentation more readable and to maintain a structure similar to Ackermann’s reduction.

3. The definition of $FC^E$, which enforces functional consistency, relies on case expressions rather than on a pairwise enforcing of consistency.

The generalization of Algorithm 3.3.2 to functions with multiple arguments is straightforward, as we shall soon see in the examples.

Example 3.8. Let us return to our main example of this chapter, the problem of proving the equivalence of programs (a) and (b) in Fig. 3.1. We continue from Fig. 3.3, where the logical formulas corresponding to these programs are given, with the use of the uninterpreted function $G$. On the basis of (3.5), we compute $flat^E$, replacing each uninterpreted-function symbol with the corresponding variable:

$$flat^E := \left( \left( out_{0.a} = in \land out_{1.a} = G^*_1 \land out_{2.a} = G^*_2 \right) \land (out_{0.b} = G^*_4) \right) \implies out_{2.a} = out_{0.b}.$$  \hfill (3.25)

Not surprisingly, this looks very similar to (3.13). The only difference is that instead of the $g_i$ variables, we now have the $G^*_i$ macros, for $1 \leq i \leq 4$. Recall their origin: the function instances are $G(out_{0.a}, in)$, $G(out_{1.a}, in)$, $G(in, in)$ and $G(G(in, in), in)$, which we number in this order. The corresponding functional-consistency constraints are

$$FC^E := \left\{ \begin{array}{l}
G^*_1 = g_1 \\
G^*_2 = \left( \begin{array}{l}
\text{case} \quad out_{0.a} = out_{1.a} \land in = in : g_1 \\
\quad \text{TRUE} : g_2
\end{array} \right) \\
G^*_3 = \left( \begin{array}{l}
\text{case} \quad out_{0.a} = in \land in = in : g_1 \\
\quad out_{1.a} = in : g_2 \\
\quad \text{TRUE} : g_3
\end{array} \right) \\
G^*_4 = \left( \begin{array}{l}
\text{case} \quad out_{0.a} = G^*_3 \land in = in : g_1 \\
\quad out_{1.a} = G^*_3 \land in = in : g_2 \\
\quad \text{TRUE} : g_4
\end{array} \right)
\end{array} \right\} \land (3.26)$$

and since we are checking for validity, the formula to be checked is

$$\varphi^E := FC^E \implies flat^E.$$ \hfill (3.27)

Example 3.9. If there are multiple uninterpreted-function symbols, the reduction is applied to each of them separately, as demonstrated in the following example, in which we consider the formula of Example 3.6 again:
72 3 Equality Logic and Uninterpreted Functions

\[ x_1 = x_2 \implies F(F(G(x_1))) = F(F(G(x_2))) \, . \tag{3.28} \]

As before, we number the function instances of each of the uninterpreted-function symbols \( F \) and \( G \) from the inside out (this order is required in Bryant’s reduction). Applying Bryant’s reduction, we obtain

\[ \text{flat}^E := (x_1 = x_2 \implies F_2^* = F_4^*) \, , \tag{3.29} \]

\[ FC^E := F_1^* = f_1 \]

\[ F_2^* \begin{cases} \text{case } G_1^* = F_1^* : f_1 \\ \text{TRUE} : f_2 \end{cases} \]

\[ F_3^* \begin{cases} \text{case } G_1^* = G_2^* : f_1 \\ F_1^* = G_2^* : f_2 \\ \text{TRUE} : f_3 \end{cases} \]

\[ F_4^* \begin{cases} \text{case } G_1^* = F_3^* : f_1 \\ F_1^* = F_3^* : f_2 \\ G_2^* = F_3^* : f_3 \\ \text{TRUE} : f_4 \end{cases} \]

\[ G_1^* = g_1 \]

\[ G_2^* = \begin{cases} \text{case } x_1 = x_2 : g_1 \\ \text{TRUE} : g_2 \end{cases} \]

and

\[ \varphi^E := FC^E \implies \text{flat}^E \, . \tag{3.31} \]

Note that in any satisfying assignment that satisfies \( x_1 = x_2 \) (the premise of (3.28)), \( F_1^* \) and \( F_3^* \) are equal to \( f_1 \), while \( F_2^* \) and \( F_4^* \) are equal to \( f_2 \).

The difference between Ackermann’s and Bryant’s reductions is not just syntactic, as was hinted earlier. It has implications for the decision procedure that one can use when solving the resulting formula. We discuss this point further in Sect. 4.6.

3.4 Functional Consistency Is Not Enough

Functional consistency is not always sufficient for proving correct statements. This is not surprising, as we clearly lose information by replacing concrete, interpreted functions with uninterpreted functions. Consider, for example, the \textit{plus} (\( '+\)’) function. Now suppose that we are given a formula containing the two function instances \( x_1 + y_1 \) and \( x_2 + y_2 \), and, owing to other parts of the formula, it holds that \( x_1 = y_2 \) and \( y_1 = x_2 \). Further, suppose that we
replace “+” with a binary uninterpreted function $F$. Since in Algorithms 3.3.1 and 3.3.2 we only compare arguments pairwise in the order in which they appear, the proof cannot rely on the fact that these two function instances are evaluated to give the same result. In other words, the functional-consistency constraints alone do not capture the commutativity of the “+” function, which may be necessary for the proof. This demonstrates the fact that by using uninterpreted functions we lose completeness (see Definition 1.6).

One may add, of course, additional constraints that capture more information about the original function – commutativity, in the case of the example above. For example, considering Ackermann’s reduction for the above example, let $f_1, f_2$ be the variables that encode the two function instances, respectively. We can then replace the functional-consistency constraint for this pair with the stronger constraint

$$\left( (x_1 = x_2 \land y_1 = y_2) \lor (x_1 = y_2 \land y_1 = x_2) \right) \implies f_1 = f_2 . \quad (3.32)$$

Such constraints can be tailored as needed, to reflect properties of the uninterpreted functions. In other words, by adding these constraints we make them partially interpreted functions, as we model some of their properties. For the multiplication function, for example, we can add a constraint that if one of the arguments is equal to 0, then so is the result. Generally, the more abstract the formula is, the easier it is, computationally, to solve it. On the other hand, the more abstract the formula is, the fewer correct facts about its original version can be proven. The right abstraction level for a given formula can be found by a trial-and-error process. Such a process can even be automated with an abstraction-refinement loop, as can be seen in Algorithm 3.4.1 (this is not so much an algorithm as a framework that needs to be concretized according to the exact problem at hand). In step 2, the algorithm returns “Valid” if the abstract formula is valid. The correctness of this step is implied by (3.3). If, on the other hand, the formula is not valid and the abstract formula $\varphi'$ is identical to the original one, the algorithm returns “Valid” in the next step. The optional step that follows (step 4) is not necessary for the soundness of the algorithm, but only for its performance. This step is worth executing only if it is easier than solving $\varphi$ itself.

Plenty of room for creativity is left when one is implementing such an algorithm: which constraints to add in step 5? When to resort to the original interpreted functions? How to implement step 4? An instance of such a procedure is described, for the case of bit-vector arithmetic, in Sect. 6.3.

---

6 Abstraction-refinement loops [111] are implemented in many model checkers [46] (tools for verifying temporal properties of transition systems) and other automated formal-reasoning tools. The types of abstractions used can be very different than from those presented here, but the basic elements of the iterative process are the same.
Aside: Rewriting systems
Observations such as “a multiplication by 0 is equal to 0” can be formulated with rewriting rules. Such rules are the basis of rewriting systems [64, 99], which are used in several branches of mathematics and mathematical logic. Rewriting systems, in their basic form, define a set of terms and (possibly non-deterministic) rules for transforming them. Theorem provers that are based on rewriting systems (such as ACL2 [104]) use hundreds of such rules. Many of these rules can be used in the context of the partially interpreted functions that were studied in Sect. 3.4, as demonstrated for the “multiply by 0” rule.

Rewriting systems, as a formalism, have the same power as a Turing machine. They are frequently used for defining and implementing inference systems, for simplifying formulas by replacing subexpressions with equal but simpler subexpressions, for computing results of arithmetic expressions, and so forth. Such implementations require the design of a strategy for applying the rules, and a mechanism based on pattern matching for detecting the set of applicable rules at each step.

Algorithm 3.4.1: Abstraction-refinement

Input: A formula $\varphi$ in a logic $L$, such that there is a decision procedure for $L$ with uninterpreted functions

Output: “Valid” if $\varphi$ is valid, “Not valid” otherwise

1. $\varphi' := T(\varphi)$.
2. If $\varphi'$ is valid then return “Valid”.
3. If $\varphi' = \varphi$ then return “Not valid”.
4. (Optional) Let $\alpha'$ be a counterexample to the validity of $\varphi'$. If it is possible to derive a counterexample $\alpha$ to the validity of $\varphi$ (possibly by extending $\alpha'$ to those variables in $\varphi$ that are not in $\varphi'$), return “Not valid”.
5. Refine $\varphi'$ by adding more constraints as discussed in sect. 3.4, or by replacing uninterpreted functions with their original interpreted versions (reaching, in the worst case, the original formula $\varphi$).
6. Return to step 2.

3.5 Two Examples of the Use of Uninterpreted Functions

Uninterpreted functions can be used for property-based verification, that is, proving that a certain property holds for a given model. Occasionally it happens that properties are correct regardless of the semantics of a certain function, and functional consistency is all that is needed for the proof. In such cases, replacing the function with an uninterpreted function can simplify the proof.
The more common use of uninterpreted functions, however, is for proving equivalence between systems. In the chip design industry, proving equivalence between two versions of a hardware circuit is a standard procedure. Another application is translation validation, a process of proving the semantic equivalence of the input and output of a compiler. Indeed, we end this chapter with a detailed description of these two problem domains.

In both applications, it is expected that every function on one side of the equation can be mapped to a similar function on the other side. In such cases, replacing all functions with an uninterpreted version and using one of the reductions that we saw in Sects. 3.3.1 and 3.3.2 is typically sufficient for proving equivalence.

### 3.5.1 Proving Equivalence of Circuits

*Pipelining* is a technique for improving the performance of a circuit such as a microprocessor. The computation is split into phases, called pipeline stages. This allows one to speed up the computation by making use of concurrent computation, as is done in an assembly line in a factory.

The clock frequency of a circuit is limited by the length of the longest path between latches (i.e., memory components), which is, in the case of a pipelined circuit, simply the length of the longest stage. The delay of each path is affected by the gates along that path and the delay that each one of them imposes.

Figure 3.4(a) shows a pipelined circuit. The input, denoted by \( \text{in} \), is processed in the first stage. We model the combinational gates within the stages with uninterpreted functions, denoted by \( C, F, G, H, K, \) and \( D \). For the sake of simplicity, we assume that they each impose the same delay. The circuit applies function \( F \) to the inputs \( \text{in} \), and stores the result in latch \( L_1 \). This can be formalized as follows:

\[
L_1 = F(\text{in}) . \tag{3.33}
\]

The second stage computes values for \( L_2, L_3, \) and \( L_4 \):

\[
\begin{align*}
L_2 &= L_1 , \\
L_3 &= K(G(L_1)) , \\
L_4 &= H(L_1) .
\end{align*}
\tag{3.34}
\]

The third stage contains a *multiplexer*. A multiplexer is a circuit that selects between two inputs according to the value of a Boolean signal. In this case, this selection signal is computed by a function \( C \). The output of the multiplexer is stored in latch \( L_5 \):

\[
L_5 = C(L_2) \ ? \ L_3 \ : \ D(L_4) . \tag{3.35}
\]
Observe that the second stage contains two functions, $G$ and $K$, where the output of $G$ is used as an input for $K$. Suppose that this is the longest path within the circuit. We now aim to transform the circuit in order to make it work faster. This can be done in this case by moving the gates represented by $K$ down into the third stage.

Observe also that only one of the values in $L_3$ and $L_4$ is used, as the multiplexer selects one of them depending on $C$. We can therefore remove one of the latches by introducing a second multiplexer in the second stage. The circuit after these changes is shown in Fig. 3.4(b). It can be formalized as follows:

$$
L'_1 = F(in),
L'_2 = C(L'_1),
L'_3 = C(L'_1) \cdot G(L'_1) : H(L'_1),
L'_5 = L'_2 \cdot K(L'_3) : D(L'_3).
$$

(3.36)

The final result of the computation is stored in $L_5$ in the original circuit, and in $L'_5$ in the modified circuit. We can show that the transformations are correct by proving that for all inputs, the conjunction of the above equalities implies
This proof can be automated by using a decision procedure for equalities and uninterpreted functions.

### 3.5.2 Verifying a Compilation Process with Translation Validation

The next example illustrates a translation validation process that relies on uninterpreted functions and Ackermann’s reduction. Unlike the hardware example, we start from interpreted functions and replace them with uninterpreted functions.

Suppose that a source program contains the statement

\[ z = (x_1 + y_1) \cdot (x_2 + y_2) , \]

which the compiler that we wish to check compiles into the following sequence of three assignments:

\[ u_1 = x_1 + y_1 ; \quad u_2 = x_2 + y_2 ; \quad z = u_1 \cdot u_2 . \]

Note the two new auxiliary variables \( u_1 \) and \( u_2 \) that have been added by the compiler. To verify this translation, we construct the verification condition

\[ u_1 = x_1 + y_1 \land u_2 = x_2 + y_2 \land z = u_1 \cdot u_2 \implies z = (x_1 + y_1) \cdot (x_2 + y_2) , \]

whose validity we wish to check.

We now abstract the concrete functions appearing in the formula, namely addition and multiplication, by the abstract uninterpreted-function symbols \( F \) and \( G \), respectively. The abstracted version of the implication above is

\[ (u_1 = F(x_1, y_1) \land u_2 = F(x_2, y_2) \land z = G(u_1, u_2)) \implies z = G(F(x_1, y_1), F(x_2, y_2)) . \]

Clearly, if the abstracted version is valid, then so is the original concrete one (see (3.3)).

Next, we apply Ackermann’s reduction (Algorithm 3.3.1), replacing each function by a new variable, but adding, for each pair of terms with the same function symbol, an extra antecedent that guarantees the functionality of these terms. Namely, if the two arguments of the original terms are equal, then the terms should be equal.

\[ u_1 = F(x_1, y_1) \land u_2 = F(x_2, y_2) \land z = G(u_1, u_2) \implies z = G(F(x_1, y_1), F(x_2, y_2)) . \]

This verification condition is an implication rather than an equivalence because we are attempting to prove that the values allowed in the target code are also allowed in the source code, but not necessarily the other way. This asymmetry can be relevant when the source code is interpreted as a specification that allows multiple behaviors, only one of which is actually implemented. For the purpose of demonstrating the use of uninterpreted functions, whether we use an implication or an equivalence is immaterial.
Applying Ackermann’s reduction to the abstracted formula, we obtain the following equality formula:

\[
\varphi^E := \left\{ \begin{array}{c}
(x_1 = x_2 \land y_1 = y_2 \implies f_1 = f_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \implies g_1 = g_2) \\
(u_1 = f_1 \land u_2 = f_2 \land z = g_1) \implies z = g_2
\end{array} \right. 
\]

which we can rewrite as

\[
\varphi^E := \left\{ \begin{array}{c}
(x_1 = x_2 \land y_1 = y_2 \implies f_1 = f_2) \land \\
(u_1 = f_1 \land u_2 = f_2 \implies g_1 = g_2) \\
u_1 = f_1 \land u_2 = f_2 \land z = g_1
\end{array} \right. \implies z = g_2.
\]

It is left to prove, then, the validity of this equality logic formula.

The success of such a process depends on how different the two sides are. Suppose that we are attempting to perform translation validation for a compiler that does not perform heavy arithmetic optimizations. In such a case, the scheme above will probably succeed. If, on the other hand, we are comparing two arbitrary source codes, even if they are equivalent, it is unlikely that the same scheme will be sufficient. It is possible, for example, that one side uses the function \(2 \ast x\) while the other uses \(x + x\). Since addition and multiplication are represented by two different uninterpreted functions, they are not associated with each other in any way according to Algorithm 3.3.1, and hence the proof of equivalence is not able to rely on the fact that the two expressions are semantically equal.

3.6 Problems

3.6.1 Warm-up Exercises

**Problem 3.1** (practicing Ackermann’s and Bryant’s reductions). Given the formula

\[
F(F(x_1)) \neq F(x_1) \land \\
F(F(x_1)) \neq F(x_2) \land \\
x_2 = F(x_1),
\]

reduce its validity problem to a validity problem of an equality logic formula through Ackermann’s reduction and Bryant’s reduction.

3.6.2 Problems

**Problem 3.2** (eliminating constants). Prove that given an equality logic formula, Algorithm 3.1.1 returns an equisatisfiable formula without constants.\(^8\)

\(^8\) Further discussion of the constants-elimination problem appears in the next chapter, as part of Problem 4.4.
Problem 3.3 (Ackermann’s reduction). Extend Algorithm 3.3.1 to multiple function symbols and to functions with multiple arguments.

Problem 3.4 (Bryant’s reduction). Suppose that in Algorithm 3.3.2, the definition of $F_i$ is replaced by

$$
F_i^* = \begin{cases}
\text{case } T^*(arg(F_i^*)) = T^*(arg(F_i^*)) : F_i^* \\
\vdots \\
T^*(arg(F_{i-1}^*)) = T^*(arg(F_i^*)) : F_{i-1}^* \\
\text{TRUE} : f_i
\end{cases},
$$

(3.45)

the difference being that the terms on the right refer to the $F_j^*$ variables, $1 \leq j < i$, rather than to the $f_j$ variables. Does this change the value of $F_i^*$? Prove a negative answer or give an example.

Problem 3.5 (abstraction/refinement). Frequently, the functional-consistency constraints become the bottleneck in the verification procedure, as their number is quadratic in the number of function instances. In such cases, even solving the first iteration of Algorithm 3.4.1 is too hard.

Show an abstraction/refinement algorithm that begins with flat$^E$ and gradually adds functional-consistency constraints.

*Hint:* note that given an assignment $\alpha'$ that satisfies a formula with only some of the functional-consistency constraints, checking whether $\alpha'$ respects functional consistency is not trivial. This is because $\alpha'$ does not necessarily refer to all variables (if the formula contains nested functions, some may disappear in the process of abstraction). Hence $\alpha'$ cannot be tested directly against a version of the formula that contains all functional-consistency constraints.

### 3.7 Glossary

The following symbols were used in this chapter:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Refers to …</th>
<th>First used on page …</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi^E$</td>
<td>Equality formula</td>
<td>60</td>
</tr>
<tr>
<td>$C_c$</td>
<td>A variable used for substituting a constant $c$ in the process of removing constants from equality formulas</td>
<td>60</td>
</tr>
<tr>
<td>$\varphi^U^F$</td>
<td>Equality formula + uninterpreted functions (before reduction to equality logic)</td>
<td>62</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Refers to ...</th>
<th>First used on page ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>A function that transforms an input formula or term by replacing each uninterpreted function $F_i$ with a new variable $f_i$</td>
<td>67</td>
</tr>
<tr>
<td>$FC^E$</td>
<td>Functional-consistency constraints</td>
<td>67</td>
</tr>
<tr>
<td>$T^*$</td>
<td>A function similar to $T$, that replaces each uninterpreted function $F_i$ with $F_i^*$</td>
<td>70</td>
</tr>
<tr>
<td>$flat^E$</td>
<td>Equal to $T(\varphi_{UF})$ in Ackermann’s reduction, and to $T^*(\varphi_{UF})$ in Bryant’s reduction</td>
<td>67, 70</td>
</tr>
<tr>
<td>$F_i^*$</td>
<td>In Bryant’s reduction, a macro variable representing the case expression associated with the function instance $F_i()$ that was substituted by $F_i$</td>
<td>70</td>
</tr>
</tbody>
</table>