4

Equality Logic and Uninterpreted Functions

4.1 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 $(y = z \vee (\neg(x = z) \wedge x = 2))$ 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 4.1 (equality logic). *An equality logic formula is defined by the following grammar:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg\mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{term}\ =\ \mathit{term} \\ \mathit{term} &: \mathit{identifier} \mid \mathit{constant} \end{aligned}$$

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.*

From an algorithmic perspective, we restrict our attention to the conjunctive fragment (i.e., conjunction is the only propositional operator allowed), since the more general Boolean structure is handled in the DPLL(T) framework, as introduced in the previous chapter.

4.1.1 Complexity and Expressiveness

The satisfiability problem for equality logic, as defined in Definition 4.1, is NP-complete. We leave the proof of this claim as an exercise (Problem 4.6).

[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.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_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 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.

4.1.2 Boolean Variables

Frequently, equality logic formulas are mixed with Boolean variables. Nevertheless, 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 procedures. The same observation applies to many of the other theories that we consider in this book.

4.1.3 Removing the Constants: a Simplification

$\boxed{\varphi^{\mathrm{E}}}$

Theorem 4.2. *Given an equality logic formula φ^{E}, there is an algorithm that generates an equisatisfiable formula (see Definition 1.9) $\varphi^{\mathrm{E}\prime}$ without constants, in polynomial time.*

Algorithm 4.1.1: Remove-Constants

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

$\boxed{C_{c_i}}$

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

Algorithm 4.1.1 eliminates the constants from a given formula by replacing them with new variables. Problems 4.1 and 4.2 focus on this procedure. Unless otherwise stated, we assume from here on that the input equality formulas do not have constants.

4.2 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:

$$F(x) = F(G(y)) \lor x + 1 = y \; . \tag{4.1}$$

Definition 4.3 (equality logic with uninterpreted functions (EUF)). *An equality logic formula with* uninterpreted functions *and* uninterpreted predicates[2] *is defined by the following grammar:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \land \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{term} \ = \ \mathit{term} \ \mid \mathit{predicate\text{-}symbol}\ (\mathit{list\ of\ terms}) \\ \mathit{term} &: \mathit{identifier} \mid \mathit{function\text{-}symbol}\ (\mathit{list\ of\ terms}) \end{aligned}$$

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, $n > 0$,

$$\begin{aligned} &\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n . \\ &\quad \textstyle\bigwedge_i t_i = t'_i \implies F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n) \quad (\textsc{Congruence}) \; , \end{aligned} \tag{4.2}$$

where $t_1, \ldots, t_n, t'_1, \ldots, t'_n$ are new variables. 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.

[2] From here on, we refer only to uninterpreted functions. Uninterpreted predicates are treated in a similar way.

4.2.1 How Uninterpreted Functions Are Used

$\boxed{\varphi^{\text{UF}}}$

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). 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 φ^{UF} is derived from a formula φ by replacing some or all of its functions with uninterpreted functions:

$$\models \varphi^{\text{UF}} \implies \models \varphi \,. \tag{4.3}$$

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. 79.) 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.[3] 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 φ denote a formula of interest that has interpreted functions. Assume that a validity check of φ is too hard (computationally), or even impossible.

[3] Note that the term *function* here refers to the mathematical definition. 'Functions' in programming languages such as C or JAVA are not necessarily mathematical functions, e.g., they do not necessarily terminate or return a value. Assuming they do, they are functionally consistent with respect to all the data that they read and write (including, e.g., global variables, the heap, data read from the environment). If the function operates in a multi-threaded program or it has nondeterminism, e.g., because of uninitialized local variables, then the definition of consistency changes—see a discussion in [66].

2. Assign an uninterpreted function to each interpreted function in φ. Substitute each function in φ with the uninterpreted function to which it is mapped. Denote the new formula by φ^{UF}.
3. Check the validity of φ^{UF}. If it is valid, return "φ is valid" (this is justified by (4.3)). 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. 4.4.

4.2.2 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. 4.1. More specifically, the goal is to prove that they return the same value for every possible input "in".

```
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;
}
```

(a)

```
int power3_new(int in)
{
  int out_b;

  out_b = (in * in) * in;

  return out_b;
}
```

(b)

Fig. 4.1. 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.[4] 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.

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

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 φ_a and φ_b, which are shown in Fig. 4.2.[5]

$$\begin{array}{rcll} out0\_a &=& in0_a & \wedge \\ out1\_a &=& out0\_a * in0_a & \wedge \\ out2\_a &=& out1\_a * in0_a & \end{array} \qquad (\varphi_a)$$

$$out0\_b = (in0_b * in0_b) * in0_b; \qquad (\varphi_b)$$

Fig. 4.2. Two formulas corresponding to the programs (a) and (b) in Fig. 4.1. The variables are defined over finite-width integers (i.e., bit vectors)

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

$$in0\_a = in0\_b \wedge \varphi_a \wedge \varphi_b \implies out2\_a = out0\_b \, . \tag{4.4}$$

Uninterpreted functions can help in proving the equivalence of the programs (a) and (b), following the general scheme suggested in Sect. 4.2.1. The motivation in this case is computational: deciding formulas with multiplication over, for example, 64-bit variables is notoriously hard. Replacing the multiplication symbol with uninterpreted functions can solve the problem.

$$\begin{array}{rcll} out0\_a &=& in0\_a & \wedge \\ out1\_a &=& G(out0\_a, in0\_a) & \wedge \\ out2\_a &=& G(out1\_a, in0\_a) & \end{array} \qquad (\varphi_a^{\mathrm{UF}})$$

$$out0\_b = G(G(in0\_b, in0\_b), in0\_b) \qquad (\varphi_b^{\mathrm{UF}})$$

Fig. 4.3. After replacing "$*$" with the uninterpreted function G

Figure 4.3 presents φ_a^{UF} and φ_b^{UF}, which are φ_a and φ_b after the multiplication function has been replaced with a new uninterpreted function G.

[5] 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 [170]. See also Example 1.25.

Similarly, if we also had addition, we could replace all of its instances with another uninterpreted function, say F. Instead of validating (4.4), we can now attempt to validate

$$\varphi_a^{\text{UF}} \wedge \varphi_b^{\text{UF}} \implies out2\_a = out0\_b\ . \tag{4.5}$$

Let us make the example more challenging. Consider the two programs in Fig. 4.4. Now the input "in" to both programs is a pointer to a linked list, which, we assume, is in both programs a structure of the following form:

```
struct list {
    struct list *n; // pointer to next element
    int data;
};
```

Simply enforcing the inputs to be the same, as we did in (4.4), is not sufficient and is in fact meaningless since it is not the absolute addresses that affect the outputs of the two programs, it is the data *at* these addresses that matter. Hence we need to enforce the data rooted at "in" at the time of entry to the functions, which is read by the two programs, to be the same at isomorphic locations. For example, the value of in −> n −> data is read by both programs and hence should be the same on both sides. We use uninterpreted functions to enforce this condition. In this case we need two such functions which we call list_n and list_data, corresponding to the two fields in list. See the formulation in Fig. 4.5. It gets a little more complicated when the recursive data structure also gets written to—see Problem 4.7.

```
int mul3(struct list *in)
{
 int i, out_a;
 struct list *a;
 a = in;
 out_a = in -> data;
 for (i = 0; i < 2; i++) {
   a = a -> n;
   out_a= out_a * a -> data;
 }
 return out_a;
}
```

(a)

```
int mul3_new(struct list *in)
{
  int out_b;

  out_b =
    in -> data *
    in -> n -> data *
    in -> n -> n -> data;

  return out_b;
}
```

(b)

Fig. 4.4. The difference between these programs and those in Fig. 4.1 is that here the input is a pointer to a list. Since now the input is an arbitrary address, the challenge is to enforce the inputs to be the same in the verification condition

$$\begin{array}{ll}
a0\_a = in0\_a & \wedge \\
out0\_a = list\_data(in0\_a) & \wedge \\
a1\_a = list\_n(a0\_a) & \wedge \\
out1\_a = G(out0\_a, list\_data(a1\_a)) & \wedge \\
a2\_a = list\_n(a1\_a) & \wedge \\
out2\_a = G(out1\_a, list\_data(a2\_a)) &
\end{array}$$

$$(\varphi_a^{\mathrm{UF}})$$

$$\begin{array}{l}
out0\_b = G(G(list\_data(in0\_b), \\
list\_data(list\_n(in0\_b)), \\
list\_data(list\_n(list\_n(in0\_b)))))
\end{array}$$

$$(\varphi_b^{\mathrm{UF}})$$

Fig. 4.5. After replacing "$*$" with the uninterpreted function G, and the fields n and data with the uninterpreted function *list_n* and *list_data*, respectively

It is sufficient now to prove (4.4) in order to establish the equivalence of these two programs.

As a side note, we should mention that that there are alternative methods to prove the equivalence of these two programs. In this case, *substitution* is sufficient: by simply substituting $out2\_a$ by $out1\_a * in$, $out1\_a$ by $out0\_a * in$, and $out0\_a$ by in in φ_a, we can automatically prove (4.4), 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 power3_con and power3_con_new:

```
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)

```
int power3_con_new
   (int in, int con) {
  int out_b;


  out_b = con?(in*in)*in
               :in;
  return out_b;
}
```

(b)

After substitution, we obtain two expressions,

$$out\_a = con?\ ((con?\ in * in : in) * in) : (con?\ in * in : in) \tag{4.6}$$

and

$$out\_b = con?\ (in * in) * in : in\ , \tag{4.7}$$

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.

Other examples of the use of uninterpreted functions are presented in Sect. 4.5.

4.3 Deciding a Conjunction of Equalities and Uninterpreted Functions with Congruence Closure

We now show a method for solving a conjunction of equalities and uninterpreted functions, introduced in 1978 by Shostak [258]. As is the case for most of the theories that we consider in this book, the satisfiability problem for conjunctions of predicates can be solved in polynomial time. Recall that we are solving the satisfiability problem for formulas without constants, as those can be removed with, for example, Algorithm 4.1.1.

Starting from a conjunction φ^{UF} of equalities and disequalities over variables and uninterpreted functions, Shostak's algorithm proceeds in two stages (see Algorithm 4.3.1) and is based on computing equivalence classes. The version of the algorithm that is presented here assumes that the uninterpreted functions have a single argument. The extension to the general case is left as an exercise (Problem 4.5).

Algorithm 4.3.1: CONGRUENCE-CLOSURE

Input: A conjunction φ^{UF} of equality predicates over variables and uninterpreted functions

Output: "Satisfiable" if φ^{UF} is satisfiable, and "Unsatisfiable" otherwise

1. Build congruence-closed equivalence classes.
   (a) Initially, put two terms t_1, t_2 (either variables or uninterpreted-function instances) in their own equivalence class if $(t_1 = t_2)$ is a predicate in φ^{UF}. All other variables form singleton equivalence classes.
   (b) Given two equivalence classes with a shared term, merge them. Repeat until there are no more classes to be merged.
   (c) Compute the *congruence closure*: given two terms t_i, t_j that are in the same class and that $F(t_i)$ and $F(t_j)$ are terms in φ^{UF} for some uninterpreted function F, merge the classes of $F(t_i)$ and $F(t_j)$. Repeat until there are no more such instances.
2. If there exists a disequality $t_i \neq t_j$ in φ^{UF} such that t_i and t_j are in the same equivalence class, return "Unsatisfiable". Otherwise return "Satisfiable".

Example 4.4. Consider the conjunction

$$\varphi^{\text{UF}} := \; x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1 \wedge F(x_1) \neq F(x_3) \; . \tag{4.8}$$

Initially, the equivalence classes are

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\} \; . \tag{4.9}$$

Step 1(b) of Algorithm 4.3.1 merges the first two classes:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\} . \tag{4.10}$$

The next step also merges the classes containing $F(x_1)$ and $F(x_3)$, because x_1 and x_3 are in the same class:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\} . \tag{4.11}$$

In step 2, we note that $F(x_1) \neq F(x_3)$ is a predicate in φ^{UF}, but that $F(x_1)$ and $F(x_3)$ are in the same class. Hence, φ^{UF} is unsatisfiable. ▟

Variants of Algorithm 4.3.1 can be implemented efficiently with a **union–find** data structure, which results in a time complexity of $O(n \log n)$ (see, for example, [210]).

We ultimately aim at solving the general case of formulas with an arbitrary Boolean structure. In the original presentation of his method, Shostak implemented support for disjunctions by means of case-splitting, which is the bottleneck in this method. For example, given the formula

$$\varphi^{\text{UF}} := \ x_1 = x_2 \lor (x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1 \land F(x_1) \neq F(x_3)) \ , \tag{4.12}$$

he considered separately the two cases corresponding to the left and right parts of the disjunction. This can work well as long as there are not too many cases to consider.

The general problem of how to deal with propositional structure arises with all the theories that we study in this book. There are two main approaches. As discussed in Chap. 3, a highly efficient method is to combine a DPLL-based SAT solver with an algorithm for deciding a conjunction of literals from a particular theory. The former searches for a satisfying assignment to the propositional part of the formula, and the latter is used to check whether a particular propositional assignment corresponds to a satisfying assignment to the equality predicates.

An alternative approach is based on a full reduction of the formula to propositional logic, and is the subject of Chap. 11.

4.4 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 *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 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, functional consistency alone does 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 constraints that capture more information about the original function—commutativity, in the case of the example above. For the above example, we may add

$$(x_1 = y_2 \land x_2 = y_1) \implies F(x_1, x_2) = F(y_1, y_2) \; . \tag{4.13}$$

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**,[6] as can be seen in Algorithm 4.4.1 (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 (4.3). If, on the other hand, the formula is not valid and the abstract formula φ' is identical to the original one, the algorithm returns "Not 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 φ 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.

4.5 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

[6] Abstraction–refinement loops [173] are implemented in many automated formal-reasoning tools. The types of abstractions used can be very different 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* [100, 151], which are used in several branches of mathematics and mathematical logic. Rewriting systems, in their basic form, define a set of terms and (possibly nondeterministic) rules for transforming them. Theorem provers that are based on rewriting systems (such as ACL2 [162]) 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. 4.4, 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 4.4.1: 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, and “Not valid” otherwise

1. $\varphi' := \mathcal{T}(\varphi)$.        ▷ $\mathcal{T}$ is an abstraction function.
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. 4.4, or by replacing
   uninterpreted functions with their original interpreted versions
   (reaching, in the worst case, the original formula $\varphi$).
6. Return to step 2.
```

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 is typically sufficient for proving equivalence.

4.5.1 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 4.6(a) shows a pipelined circuit. The input, denoted by 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 in, and stores the result in latch L_1. This can be formalized as follows:

$$L_1 = F(in) \ . \tag{4.14}$$

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

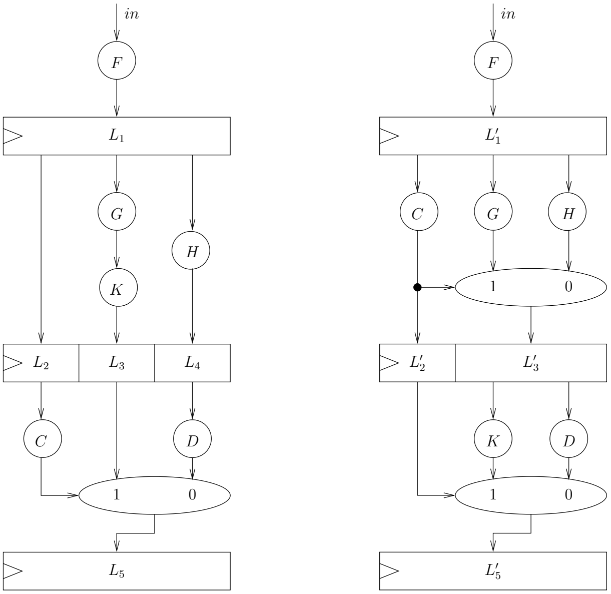
$$\begin{aligned} L_2 &= L_1 \ , \\ L_3 &= K(G(L_1)) \ , \\ L_4 &= H(L_1) \ . \end{aligned} \tag{4.15}$$

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:

$$L_5 = C(L_2) \, ? \, L_3 \ : \ D(L_4) \ . \tag{4.16}$$

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



(a) Original circuit (b) After transformation

Fig. 4.6. Showing the correctness of a transformation of a pipelined circuit using uninterpreted functions. After the transformation, the circuit has a shorter longest path between stages, and thus can be operated at a higher clock frequency

of the latches by introducing a second multiplexer in the second stage. The circuit after these changes is shown in Fig. 4.6(b). It can be formalized as follows:

$$\begin{aligned} L'_1 &= F(in)\ , \\ L'_2 &= C(L'_1)\ , \\ L'_3 &= C(L'_1)\,?\,G(L'_1)\ :\ H(L'_1)\ , \\ L'_5 &= L'_2\,?\,K(L'_3)\ :\ D(L'_3)\ . \end{aligned} \tag{4.17}$$

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

$$L_5 = L'_5\ . \tag{4.18}$$

This proof can be automated by using a decision procedure for equalities and uninterpreted functions.

4.5.2 Verifying a Compilation Process with Translation Validation

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

Suppose that a source program contains the statement

$$z = (x_1 + y_1) * (x_2 + y_2) \ , \tag{4.19}$$

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

$$u_1 = x_1 + y_1; \ u_2 = x_2 + y_2; \ z = u_1 * u_2 \ . \tag{4.20}$$

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

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 * u_2 \implies z = (x_1 + y_1) * (x_2 + y_2) \ , \tag{4.21}$$

whose validity we wish to check.[7]

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

$$\begin{array}{l} (u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2)) \\ \implies \ z = G(F(x_1, y_1), F(x_2, y_2)) \ . \end{array} \tag{4.22}$$

Clearly, if the abstracted version is valid, then so is the original concrete one (see (4.3)).

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 $2 * x$ 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 the requirement of functional consistency, and hence the proof of equivalence is not able to rely on the fact that the two expressions are semantically equal.

[7] 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.

4.6 Problems

Problem 4.1 (eliminating constants). Prove that, given an equality logic formula, Algorithm 4.1.1 returns an equisatisfiable formula without constants.

Problem 4.2 (a better way to eliminate constants?). Is the following theorem correct?

Theorem 4.5. *An equality formula φ^{E} is satisfiable if and only if the formula $\varphi^{\mathrm{E}'}$ generated by Algorithm 4.6.1* (Remove-Constants-Optimized) *is satisfiable.*

Prove the theorem or give a counterexample. You may use the result of Problem 4.1 in your proof.

Algorithm 4.6.1: Remove-Constants-Optimized

```
Input:  An equality logic formula $\varphi^{\mathrm{E}}$
Output: An equality logic formula $\varphi^{\mathrm{E}'}$ such that $\varphi^{\mathrm{E}'}$ contains
        no constants and $\varphi^{\mathrm{E}'}$ is satisfiable if and only if $\varphi^{\mathrm{E}}$ is
        satisfiable

1. $\varphi^{\mathrm{E}'} := \varphi^{\mathrm{E}}$.
2. Replace each constant $c$ in $\varphi^{\mathrm{E}'}$ with a new variable $C_c$.
3. For each pair of constants $c_i, c_j$ with an equality path between them
   ($c_i =^* c_j$) not through any other constant, add the constraint $C_{c_i} \neq$
   $C_{c_j}$ to $\varphi^{\mathrm{E}'}$. (Recall that the equality path is defined over $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$,
   where $\varphi^{\mathrm{E}}$ is given in NNF.)
```

Problem 4.3 (deciding a conjunction of equality predicates with a graph analysis). Show a graph-based algorithm for deciding whether a given conjunction of equality predicates is satisfiable, while relying on the notion of contradictory cycles. What is the complexity of your algorithm?

Problem 4.4 (deciding a conjunction of equalities with equivalence classes).

1. Consider Algorithm 4.6.2. Present details of an efficient implementation of this algorithm, including a data structure. What is the complexity of your implementation?

```
Algorithm 4.6.2: Conj-of-Equalities-with-Equiv-Classes

Input:  A conjunction $\varphi^{\mathrm{E}}$ of equality predicates
Output: "Satisfiable" if $\varphi^{\mathrm{E}}$ is satisfiable, and "Unsatisfiable" otherwise

(a) Define an equivalence class for each variable. For each equality $x = y$
    in $\varphi^{\mathrm{E}}$, unite the equivalence classes of $x$ and $y$.
(b) For each disequality $u \neq v$ in $\varphi^{\mathrm{E}}$, if $u$ is in the same equivalence class
    as $v$, return "Unsatisfiable".
(c) Return "Satisfiable".
```

2. Apply your algorithm to the following formula, and determine if it is satisfiable:

$$x = f(f(f(f(f(x))))) \wedge x = f(f(f(x))) \wedge x \neq f(x)\ .$$

Problem 4.5 (a generalization of the Congruence-Closure algorithm). Generalize Algorithm 4.3.1 to the case in which the input formula includes uninterpreted functions with multiple arguments.

Problem 4.6 (complexity of deciding equality logic). Prove that deciding equality logic is NP-complete.

Note that, to show membership in NP, it is not enough to say that every solution can be checked in P-time, because the solution itself can be arbitrarily large, and hence even reading it is not necessarily a P-time operation.

Problem 4.7 (using uninterpreted functions to encode fields of a recursive data structure). Recall the example at the second part of Sect. 4.2.2, involving pointers. The method as presented does not work if the data structure is also written to. For example, in the figure below, the code on the left results in the SSA equation on the right, which is contradictory.

```
a -> data = 1;
x = a -> data;
a -> data = 2;
x = a -> data.
```

$$\begin{array}{l} \mathit{data}(a) = 1 \quad \wedge \\ x = \mathit{data}(a) \quad \wedge \\ \mathit{data}(a) = 2 \quad \wedge \\ x_1 = \mathit{data}(a); \end{array}$$

Generalize the method so it also works in the presence of updates.

4.7 Bibliographic Notes

The treatment of equalities and uninterpreted functions can be divided into several eras. Solving the conjunctive fragment, for example as described in

Sect. 4.3, coupled with the DPLL(T) framework that was described in the previous chapter, is the latest of those.

In the first era, before the emergence of the first effective theorem provers in the 1970s, this logic was considered only from the point of view of mathematical logic, most notably by Ackermann [1]. In the same book, he also offered what we now call Ackermann's reduction, a procedure that we will describe in Sect. 11.2.1. Equalities were typically handled with rewriting rules, for example, substituting x with y given that $x = y$.

The second era started in the mid-1970s with the work of Downey, Sethi, and Tarjan [106], who showed that the decision problem was a variation on the common-subexpression problem; the work of Nelson and Oppen [205], who applied the union–find algorithm to compute the congruence closure and implemented it in the Stanford Pascal Verifier; and then the work of Shostak, who suggested in [258] the congruence closure method that was briefly presented in Sect. 4.3. All of this work was based on computing the congruence closure, and indicated a shift from the previous era, as it offered complete and relatively efficient methods for deciding equalities and uninterpreted functions. In its original presentation, Shostak's method relied on syntactic case-splitting (see Sect. 1.3), which is the source of the inefficiency of that algorithm. In Shostak's words, "it was found that most examples four or five lines long could be handled in just a few seconds". Even factoring in the fact that this was done on a 1978 computer (a DEC-10 computer), this statement still shows how much progress has been made since then, as nowadays many formulas with tens of thousands of variables are solved in a few seconds. Several variants on Shostak's method exist, and have been compared and described in a single theoretical framework called **abstract congruence closure** in [10]. Shostak's method and its variants are still used in theorem provers, although several improvements have been suggested to combat the practical complexity of case-splitting, namely *lazy case-splitting*, in which the formula is split only when it is necessary for the proof, and other similar techniques.

The third era will be described in Chap. 11 (see also the bibliographic notes in Sect. 11.9). It is based on the *small-model property*, namely reducing the problem to one in which only a finite set of values needs to be checked in order to determine satisfiability (this is not trivial, given that the original domain of the variables is infinite). The fourth and current era, as mentioned above, is based on solving the conjunctive fragment as part of the DPLL(T) framework.

4.8 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
φ^{E}	Equality formula	78
C_c	A variable used for substituting a constant c in the process of removing constants from equality formulas	78
φ^{UF}	Equality formula + uninterpreted functions	80