purposes: first, it provides a fast mechanism for comparing values using physical equality or hash equality. Second, it is easy to use hash-consing to build fast map structures using hash-consed values as keys. Finally, using such maps it is possible to implement memoization.

This assessment led us, in collaboration with Braibant and Monniaux [BJM13, BJM14], to the study of several methods to implement maximal sharing (i.e., *hash-consing*) and memoization in formally verified Coq programs. We used the case study of binary decision diagrams (BDDs), which are one of the well known uses of the hash-consing technique. We tried different approaches and compared them, as reported in the following sections. These ideas are not currently implemented in Verasco, but we believe some of them (especially the SMART and SMART+UID approaches described in Section 9.4) could be adapted to many of its data structures.

9.1. Safe Physical Equality in Coq: the PHYSEQ Approach

The obvious way of introducing physical equality in Coq is to declare it as an axiom in the development, state that physical equality implies Leibniz equality, and ask the extraction mechanism to extract it to OCaml's physical equality:

```
Parameter physEq: ∀ A:Type, A -> A -> bool.
Axiom physEq_correct: ∀ (A:Type) (x y:A), physEq x y = true -> x = y.
Extract Constant physEq => "(==)".
```

However, this appears to be unsound. Let `a` and `b` be two physically different copies of the same value. Then we have `physEq a a = true` and `a = b`, using Coq's Leibniz equality. Thus, we deduce, in Coq's logic, that `physEq a b = true`, which is wrong.

This unsoundness is of a particular kind: in fact, the axioms we postulate are not inconsistent: they can be easily instantiated by posing `physEq x y = false`. However, the OCaml term `(==)` is not a valid extraction for `physEq`, and using it would make it possible to prove properties on programs that will become false after extraction.

In order to circumvent this problem, we propose to avoid having physical equality in the language. Instead, we provide a term that enables us to use physical equality, just like a church Boolean would enable us to use a Boolean without manipulating a term of type `bool`: that is, we expose a function, taking as parameter the values of each branch of the test of physical equality. In order to make sure no use of physical equality is harmful, we add the condition, as an additional dependent parameter in `Prop`, that both branches are actually *equal* in the case of physical equality. Thus, the type of `physEq` becomes:

```
physEq: ∀ {A B:Type} (x y:A) (eq neq:B) (H:x = y -> eq = neq), B.
```

Note that, as is, `physEq` is not useful, because both branches are always executed. There is a simple solution: delaying the evaluation of branches by hiding them behind a function taking the `unit` value as parameter:

```
physEq: ∀ {A B:Type} (x y:A) (eq neq:unit -> B)
          (H:x = y -> eq tt = neq tt), B.
```

This declaration is extracted to a simple OCaml term:

```
Extract Constant physEq =>
  "(fun x y eq neq -> if x == y then eq () else neq ())".
```