From Boolean Equalities to Constraints*

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. Although functional as well as logic languages use equality to discriminate between logically different cases, the operational meaning of equality is different in such languages. Functional languages reduce equational expressions to their Boolean values, `True` or `False`, logic languages use unification to check the validity only and fail otherwise. Consequently, the language Curry, which amalgamates functional and logic programming features, offers two kinds of equational expressions so that the programmer has to distinguish between these uses. We show that this distinction can be avoided by providing an analysis and transformation method that automatically selects the appropriate operation. Without this distinction in source programs, the language design can be simplified and the execution of programs can be optimized. As a consequence, we show that one kind of equational expressions is sufficient and unification is nothing else than an optimization of Boolean equality.

1 Motivation

Functional as well as logic programming languages are based on the common idea to specify computational problems in a high-level and descriptive manner. However, the computational entities and, thus, the programming styles are different. This can be seen in a prominent feature of such languages: the discrimination between logically different cases of a given problem. Functional (as well as imperative) languages use Boolean equations for this purpose, i.e., an equational expression is reduced to either `True` or `False` and, depending on the computed result, a different computation path is selected. A typical example is the factorial function where the base case is distinguished from the recursive case by comparing the argument with `0`:[3]

```
fac n = if n==0 then 1
                else n * fac (n-1)
```

On the other hand, logic languages, like Prolog, use separate rules for different cases where (equational) constraints restrict the applicability of the rules. For

* This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

[3] We use the syntax of Haskell [22] for functional programs.

instance, the following Prolog program defines the concatenation relation between three lists (where we do not use patterns in left-hand sides to make the equational constraints explicit):

```
append(X,Y,Z) :- X=[], Y=Z.
append(X,Y,Z) :- X=[E|T], Z=[E|U], append(T,Y,U).
```

The equality symbol "`=`" used in this program is different from the Boolean equality "`==`" above. For instance, in the first rule it is not intended to evaluate `X=[]` to `True` or `False`, but this equality must hold to proceed with this rule, i.e., it is a constraint for subsequent evaluation steps. As a consequence, it is not necessary to fully evaluate equational expressions but one can continue a computation even with partial knowledge, as long as the constraint is ensured to hold. For instance, if we want to ensure that a list `L` ends with the element `0`, we can write

```
append(_,[0],L)
```

which is solvable even if the values of the list elements are not known. Thus, if `L=[A,B,C]` is a list of three variables, then the literal above is solved by binding `C` to `0` but leaving all other list elements unspecified. Operationally, this is done by unification [25] instead of evaluation to Boolean values.

Functional logic languages attempt to combine the most important features of functional and logic programming in a single language (see [5, 16] for recent surveys). In particular, the functional logic language Curry [19] extends Haskell by common features of logic programming, i.e., non-determinism, free variables, and equational constraints. Due to its roots in functional *and* logic programming, Curry provides two kinds of equalities: Boolean equality ("`==`") as in functional programming and equational constraints ("`=:=`") as in logic programming. The motivation for this decision is to support nested case distinctions, like in functional programming, as well as rule-oriented programming with partial information, like in logic programming. Although one might argue that it is always possible to guess values for unknowns, so that one kind of equality is sufficient, an important insight of logic programming is that unification can restrict the search space by binding variables instead of guessing values [25]. For instance, if `X` and `Y` are Boolean variables, the equational constraint "`X=Y`" can be solved by simply binding `X` to `Y` instead of enumerating appropriate values for `X` and `Y`.

Although the distinction between these two kinds of equalities is present in Curry from its early design [15], it also causes some complications. A programmer might not always easily understand which equality should be chosen in a particular situation. Moreover, the distinction between solving and evaluating equalities is also present in the type system, i.e., "`==`" has the result type `Bool` whereas "`=:=`" has the result type `Success` (indicating the type of constraints). As a consequence, various standard (combinator) functions on Booleans need also be duplicated for the type `Success`.

In order to improve this situation, we argue in this paper that one kind of equality, namely Boolean equality, is sufficient for the programmer. This will be justified by an automatic method to transform Boolean equalities into constraint

equalities, if it is appropriate. Hence, we automatically obtain the nice features of unification, i.e., reduction of the search space. For this purpose, we present a program analysis and transformation method that automatically selects the appropriate kind of equality. This leads to a simpler language design without sacrificing program efficiency.

2 Functional Logic Programming and Curry

We briefly review those elements of functional logic languages and Curry which are necessary to understand the contents of this paper. More details can be found in recent surveys on functional logic programming [5, 16] and in the language report [19].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming (concurrency is irrelevant as our work goes, hence it is ignored in this paper). The syntax of Curry is close to Haskell [22], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. Functional types are "curried," i.e., $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β, and the application of an operation f to an argument e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows *free* (*logic*) *variables* in conditions and right-hand sides of rules and expressions evaluated by an interpreter.

A *Curry program* consists in the definition of *functions* or *operations* and the *data types* on which the functions operate. Functions are defined by (conditional) equations and are evaluated lazily. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of demanded arguments which corresponds to narrowing [26, 23]. Curry narrows with possibly non-most-general unifiers to ensure the optimality of computations [4].

Example 1. We present the above feature in a program chosen for its simplicity and brevity, rather than its power. The program defines the data type of Boolean values and polymorphic lists and operations to concatenate two lists and compute the last element of a list:[4]

```
data Bool   = True | False
data List a = []   | a : List a

(++) :: [a]  →  [a]  →  [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a]  →  a
last xs | _ ++ [x] =:= xs  = x
```

The `data` type declarations define `True` and `False` as Boolean values and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists

[4] Note that Curry requires the explicit declaration of free variables, as `x` in the rule of `last`, to ensure checkable redundancy, but we omit them in this paper for the sake of simplicity.

(`a` is a type variable ranging over all types and the type "`List a`" is written as `[a]` for conformity with Haskell). The (optional) type declaration ("`::`") of the operation "`++`" specifies that "`++`" takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type. Since "`++`" can be called with free variables in arguments, the equation "`_ ++ [x] =:= xs`" in the condition of `last` is solved by instantiating the anonymous free variable `_` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

The (optional) condition of a program rule is typically a conjunction of constraints. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms.

In order to use equations to discriminate between different cases, as in the definition of the factorial function `fac` shown in Section 1, Curry also offers a Boolean equality operator "`==`" which evaluates to `True` if both arguments can be evaluated to identical data terms, and to `False` if the arguments evaluate to different data terms. Conceptually, "`==`" can be considered as defined by rules comparing constructors of the same type, i.e., by the following rules ("`&&`" is the Boolean conjunction):

```
True  == True  = True       []     == []     = True
False == False = True       (x:xs) == (y:ys) = x==y && xs==ys
True  == False = False      []     == (y:ys) = False
False == True  = False      (x:xs) == []     = False
```

As already discussed in [6], the presence of the types `Success` and `Bool` together with two equality operators, rooted in the history of Curry, might cause confusions and should be avoided in order to obtain a simpler definition of Curry. Hence, [6] proposes to omit the type `Success` and the operator "`=:=`" from the definition of Curry, and we follow this proposal in our paper. Note that one can also solve equations by narrowing with the above rules. For instance, `[x,x]==[True,y]` is solved by instantiating `x` and `y` to `True` while evaluating "`==`". However, solving equations by narrowing with "`==`" rules has also a drawback compared to logic programming. If there is an equation between two variables, narrowing enumerates all values for these variables whereas unification (deterministically!) binds one variable to the other. Hence, the expression "`xs == ys && xs++ys == [True]`" has an infinite search space with solely `False` results.

This was the motivation for the inclusion of the operator "`=:=`" in Curry. Conceptually, it can be considered as defined by "positive" rules:

```
True  =:= True  = True     []     =:= []     = True
False =:= False = True     (x:xs) =:= (y:ys) = x=:=y && xs=:=ys
```

Thus, "`=:=`" yields `True` for identical data terms or fails.[5] Operationally, these rules are not applied by narrowing but combined with the unification principle

[5] Note that we omit the type `Success`, as proposed in [6]. Hence, equational constraints as well as rule conditions are of type `Bool` rather than `Success`, in contrast to the current definition of Curry [19].

[25], i.e., if one argument is a free variable, it is bound to the evaluated data term of the other side (if the variable is not contained in this term, see [19] for details). Therefore, the expression "`xs =:= ys`" evaluates to `True` by binding `xs` to `ys` and the expression "`xs =:= ys && xs++ys =:= [True]`" has a finite search space without any result.

It would be desirable to automatically replace occurrences of "`==`" by "`=:=`" whenever it can be done without losing solutions (see the next section). This would free the programmer from having to select the "right" equality and simplify the language: programmers always use "`==`" so that the operator "`=:=`" is just an optimization of "`==`". This is the motivation for our current work.

Since Curry with all its syntactic sugar (we have only presented a small fragment of it) is a quite rich source language, a simpler intermediate representation of Curry programs has been shown to be useful to describe the operational semantics [1], compile programs [10, 17], or implement analyzers [18] and similar tools. Programs of this intermediate language, called FlatCurry, contain a single rule for each function where the pattern matching strategy is represented by case expressions. The basic structure of FlatCurry is defined as follows (where x_i denotes variables, f defined functions, C constructors, and $\overline{o_k}$ a sequence of objects $o_1 \ldots o_k$):

P	$::= D_1 \ldots D_m$	(program)
D	$::= f\ \overline{x_n} = e$	(function definition)
p	$::= C\ \overline{x_n}$	(flat pattern)
e	$::= x$	(variable)
	$\mid\ C\ \overline{e_n}$	(constructor application)
	$\mid\ f\ \overline{e_n}$	(function application)
	$\mid\ \mathit{case}\ e_0\ \mathit{of}\ \{\overline{p_k \to e_k}\}$	(case distinction)
	$\mid\ e_1\ \texttt{?}\ e_2$	(non-deterministic choice)

A program P (we omit data type declarations) consists of a sequence of function definitions D with pairwise different variables in the left-hand sides. The right-hand sides are expressions e composed by variables, constructor and function calls, case expressions, and disjunctions. A case expression[6] has the form $\mathit{case}\ e\ \mathit{of}\ \{C_1\ \overline{x_{n_1}} \to e_1, \ldots, C_k\ \overline{x_{n_k}} \to e_k\}$, where e is an expression, $C_1, \ldots, C_k$ are different constructors of the type of e, and $e_1, \ldots, e_k$ are expressions. The *pattern variables* $\overline{x_{n_i}}$ are local variables which occur only in the corresponding subexpression e_i

By fixing a strategy to match arguments, one can translate Curry programs into FlatCurry programs. The higher-order constructs of Curry are translated into FlatCurry by defunctionalization [24]. Thus, lambda abstractions are transformed into top-level functions and there is a predefined operation *apply* to apply an expression of functional type to an argument (see [16, 27] for more details).

[6] Since we do not discuss residuation and concurrent computations, we also omit the difference between rigid and flexible case expressions [16].

Conditional rules are not present in FlatCurry since, as shown in [3], they can be transformed into unconditional ones by introducing a "conditional" operator `cond` defined by

```
cond True x = x
```

For instance, the rule defining `last` as shown above can be transformed into

```
last xs = cond (_++[x] =:= xs) x
```

The evaluation strategy of Curry is by-need. Hence, the second argument of *cond* is evaluated only if the first argument is `True`.

3 Transforming Equalities

In this section we discuss an automatic method to replace occurrences of Boolean equalities of the form e_1`==`e_2 by an equational constraint e_1`=:=`e_2. Obviously, such a replacement is not always correct. For instance, consider the following contrived example:

```
isEmpty xs = if xs==[] then True else False
```

If we evaluate the expression "`isEmpty xs`", where `xs` is a free variable, we obtain the following two results (e.g., with the Curry system KiCS2 [10]):

```
{xs = []} True
{xs = (_x1:_x2)} False
```

These two results are computed by narrowing the equation `xs==[]` w.r.t. the rules defining "`==`" shown in the previous section. However, if we replace the Boolean equality by an equational constraint, as in

```
isEmpty' xs = if xs=:=[] then True else False
```

and evaluate the expression "`isEmpty' xs`", then we obtain only the single result

```
{xs = []} True
```

since the constraint "`xs=:=[]`" can only be satisfied, i.e., delivers the value `True` only.

Thus, in order to avoid losing solutions, a Boolean equation e_1`==`e_2 can be replaced by the equational constraint e_1`=:=`e_2 if it is ensured that only the value `True` is required as the result of this equation. In general, this depends on the context of the equation. Fortunately, there are many situations in functional logic programs where this requirement can be deduced. For instance, consider the following definition of `last`:

```
last xs | xs == _++[x]  = x
```

As discussed above, this rule is transformed into the unconditional rule

```
last xs = cond (xs == _++[x]) x
```

Since the definition of `cond` requires that the first argument must have the value `True` in order to evaluate a `cond` expression, the condition can be replaced by an equational constraint:

```
last' xs = cond (xs =:= _++[x]) x
```

Hence, if we evaluate `last' [x,42]`, where `x` is a free variable, we obtain the single result

```
{x = _x1} 42
```

On the other hand, we obtain infinitely many answers for the expression `last [x,42]` (where in each answer `x` is bound to a different integer value). Similarly, we can replace the occurrences of "`==`" by "`=:=`" in the rule

```
f xs ys | xs == _++[x] && ys == _++[x]++_ = x
```

However, in the rule

```
g xs ys | xs == _++[x] && not (ys == _++[x]++_) = x
```

only the first occurrence of "`==`" can be replaced by "`=:=`", since the second occurrence is required to be evaluated to `False` in order to apply the rule.[7]

These examples show that a careful analysis of the kind of values required for a successful evaluation is necessary in order to perform our proposed transformation. Note that such an analysis is different from a strictness analysis in purely functional programming [21]. A strictness analysis provide information about the necessary demand of computation in order to compute any value, whereas we need information about possible values in order to compute other values. For instance, in order to transform the definition of `f` above, it is necessary to know that both arguments of the conjunction operator "`&&`" need to be `True` in order to obtain the overall value `True`. For this purpose, we define in the next section an appropriate analysis for "required" values.

4 Analysis of Required Values

Our goal is to develop a program analysis to infer which kind of values are required at some position in a program in order to compute a result, i.e., some value. To obtain a manageable analysis, we consider only top-level constructors in the analysis so that a *value* is some constructor-rooted expression. In principle, this could be extended to any depth bound k (as used in the abstract diagnosis of functional programs [2] or in the abstraction of term rewriting systems [8, 9]), but in practice only a depth $k = 1$ (i.e., top-level constructors) is useful due to the quickly growing size of the abstract domain for $k > 1$. For instance, for lists we distinguish the values `[]` (empty list) and "`:`" (non-empty lists) and for Booleans we distinguish the values `True` and `False`.

Following the framework of abstract interpretation [13], we define for each type τ an *abstract domain* τ^α, i.e., a set of *abstract values*, as follows. If $\mathcal{C}_\tau = \{C_1, \ldots, C_k\}$ denotes the set of all constructors of type τ, then $\tau^\alpha = 2^{\mathcal{C}_\tau} \cup \{Any\}$, i.e., an abstract value of τ^α is either a subset of the constructors of type τ or the specific constant *Any* denoting any expression. For instance, the abstract domain for Boolean values is

$\texttt{Bool}^\alpha$ `= { ∅, {True}, {False}, {True,False},` *Any* `}`

[7] The latter equality could also be improved if disequality constraints [7, 20] are available in the target language, but since this is not the case for standard implementations of Curry, we do not consider them in this paper.

Abstract values are ordered by: for all τ_1 and τ_2, $\tau_1 \sqsubseteq \mathit{Any}$, and $\tau_1 \sqsubseteq \tau_2$ if $\tau_1 \subseteq \tau_2$ and both are not Any. Thus, the least upper bound of two abstract values $\tau_1 \neq \mathit{Any} \neq \tau_2$ is their set union, i.e., $\tau_1 \sqcup \tau_2 = \tau_1 \cup \tau_2$.

The meaning of an abstract value a, i.e., the concretization $[\![a]\!]$ of a, is the set of all expressions, if $a = \mathit{Any}$, or, if $a \neq \mathit{Any}$, the set of all values rooted by some constructor of a (where $\mathit{root}(e)$ denotes the symbol at the root of the expression e): $[\![a]\!] = \{e \mid \mathit{root}(e) \in a\}$. We call two abstract values $a, a' \in \tau^\alpha$ *compatible* if $[\![a]\!] \cap [\![a']\!] \neq \varnothing$, i.e., if they have some element in common.

As discussed above, we are interested to deduce required argument values from required result values. For instance, if `True` is the required value of a conjunction e_1 `&&` e_2, then `True` is also the required value of both e_1 and e_2. We denote this property by `(&&)` $::^\alpha$ {`True`},{`True`}→{`True`}.

We can read this type as: in order to compute the result `True`, the argument values are required to be `True`. Or: unless both arguments are evaluated to `True`, the result cannot be `True`.

Definition 1. *A typing $f ::^\alpha a_1, \ldots, a_n \to a$ of a function f is* correct *if, for all $e = f\ e_1 \ldots e_n$, the following implication holds: if e evaluates to some value (constructor-rooted term) $t \in [\![a]\!]$, then, for $i = 1, \ldots, n$, e_i evaluates to some $t'_i \in [\![a_i]\!]$.*

The above notion of correctness establishes a condition on the values of the arguments of a function application to produce a certain value as the result of the application. For each function f of (concrete) type $\tau_1, \ldots, \tau_n \to \tau$, the typing $f ::^\alpha \mathit{Any}, \ldots, \mathit{Any} \to \mathcal{C}_\tau$ (with appropriate numbers of arguments) is correct since any expression is an element of $[\![\mathit{Any}]\!]$. Clearly, defined functions can have more than one correct typing. For instance, the negation operator `not` has the types

`not` $::^\alpha$ {`True`}→{`False`}
`not` $::^\alpha$ {`False`}→{`True`}

and the conjunction operator `(&&)` has the types

`(&&)` $::^\alpha$ {`True`},{`True`}→{`True`}
`(&&)` $::^\alpha$ *Any*,*Any* →{`False`}

These abstract types can be used as follows. If the condition of a program rule has the form e_1 `&&` e_2, the value `True` is required as the result of this conjunction. By the first type of "`&&`", we can deduce that `True` is also required as the result of both expressions e_1 and e_2, otherwise the conjunction cannot be evaluated to `True`. However, if a condition has the form `not (`e_1 `&&` e_2`)`, we cannot deduce a single value required for e_1 or e_2 (by the second type of "`&&`"), since this condition yields `True` if e_1 has the value `False` or if e_1 has the value `True` and e_2 has the value `False`. Note that

`(&&)` $::^\alpha$ {`False`},*Any* →{`False`}

is not a correct typing: `True` $\notin [\![$`{False}`$]\!]$ but `True && False` $\in [\![$`{False}`$]\!]$. This is intended: we cannot deduce from the required result value `False` that the first argument is required to be `False`.

$$\textit{Var}\quad \frac{}{F \vdash x ::^\alpha a \mid \{x ::^\alpha a\}} \quad \text{if } x \text{ is a variable}$$

$$\textit{Con}\quad \frac{}{F \vdash C\ e_1 \ldots e_n ::^\alpha a \mid \varnothing} \quad \text{if } \{C\} \text{ and } a \text{ are compatible}$$

$$\textit{Fun}\quad \frac{F \vdash e_1 ::^\alpha a_1 \mid E_1 \ldots F \vdash e_n ::^\alpha a_n \mid E_n}{F \vdash f\ e_1 \ldots e_n ::^\alpha a \mid \bigsqcap\{E_i \mid a_i \neq \mathit{Any}\}} \quad \text{if } f ::^\alpha a_1, \ldots, a_n \to a \in F$$

$$\textit{Or}\quad \frac{F \vdash e_1 ::^\alpha a \mid E_1 \quad F \vdash e_2 ::^\alpha a \mid E_2}{F \vdash e_1\,?\,e_2 ::^\alpha a \mid E_1 \sqcup E_2}$$

$$\textit{Case}\quad \frac{\begin{array}{c} F \vdash e_0 ::^\alpha a' \mid E_0 \quad F \vdash e_1 ::^\alpha a \mid E_1 \ldots F \vdash e_j ::^\alpha a \mid E_j \\ F \vdash e_{j+1} ::^\alpha a_{j+1} \mid E_{j+1} \ldots F \vdash e_n ::^\alpha a_n \mid E_n \end{array}}{F \vdash \mathit{case}\ e_0\ \mathit{of}\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\} ::^\alpha a \mid E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j)}$$

if $C_1, \ldots, C_j \in a'$ and, for $i = j+1, \ldots, n$, a_i and a are not compatible

Fig. 1. Abstract typing rules for FlatCurry expressions

In order to define well-typed programs, we assume a *type environment* F (for a given program) which contains for each n-ary function symbol f occurring in the program at least one element of the form $f ::^\alpha a_1, \ldots, a_n \to a$. Since we want to know required values of arguments in order to compute some value of an expression, our type analysis also returns a *variable type environment* E containing variable types $x ::^\alpha a$ for variables x occurring in the expression. The least upper bound $E_1 \sqcup E_2$ of two variable type environments E_1 and E_2 is the element-wise least upper bound of the associated types (where absent type information is interpreted as *Any*), e.g., $\{x ::^\alpha \{\texttt{True}\}, y ::^\alpha \{\texttt{True}\}\} \sqcup \{x ::^\alpha \{\texttt{False}\}\} = \{x ::^\alpha \{\texttt{True}, \texttt{False}\}, y ::^\alpha \mathit{Any}\}$. Observe that $y ::^\alpha \mathit{Any}$ is in the upper bound because the second environment places no restrictions on y. Similarly, $E_1 \sqcap E_2$ denotes the greatest lower bound of E_1 and E_2.

The (abstract) typing rules are shown in Fig. 1. The notation $F \vdash e ::^\alpha a \mid E$ should be read as: "if e is evaluated to some value of type a w.r.t. type environment F, then E are the required values of variables occurring in e." Rule *Var* requires the type of a variable as the type of the expression. Rule *Con* does not put requirements on variables since the term is already a value. Rule *Fun* requires well-typed arguments and an appropriate function typing to apply a function, but joins only the requirements of arguments where a value is required, since other arguments might not be evaluated. Rule *Or* requires that both alternatives of a choice expression must have the same type where the variable type environments are unified from both alternatives. Finally, rule *Case* requires that the constructors of the patterns in the various branches must be contained in the type of the discriminating expression. However, branches with a type that is not compatible with the overall result type are ignored. By this refinement, we can obtain more precise information about required arguments.

Definition 2. *A program P is* well typed w.r.t. a type environment F *for P if, for each rule $f\ x_1 \ldots x_n = e \in P$ and each $f :: ^\alpha a_1, \ldots, a_n \to a \in F$, $F \vdash e :: ^\alpha a \mid E$ is derivable by the rules in Fig. 1, for some variable type environment E, and, for $i = 1, \ldots, n$, $a'_i \subseteq a_i$ if $x_i :: ^\alpha a'_i \in E$, otherwise $a_i = Any$, i.e., the deduced required value is more specific or does not occur.*

We show the usage of this type system by a few examples that are relevant for the application intended with this paper. In these examples, we write `T` and `F` for the abstract types {`True`} and {`False`}, respectively. The first example is the operator `cond` introduced in Sect. 2 to transform conditional equations. In FlatCurry, this operator is defined by the rule

```
cond x y = case x of { True  →  y }
```

This rule is well-typed w.r.t. $\texttt{cond} ::^\alpha \texttt{T}, Any \to Any$ so that we can deduce that the first argument is required to be `True` in order to compute any value. Note that this rule is also well typed w.r.t. $\texttt{cond} ::^\alpha Any, Any \to Any$, but this typing provides less precise information about required arguments.

The second example is the negation operator `not` defined by

```
not x = case x of { True   →  False
                  ; False  →  True }
```

It is easy to check that $\texttt{not} ::^\alpha \texttt{T} \to \texttt{F}$ is a well-typing of `not` since the following derivation is valid w.r.t. $F = \{\texttt{not} ::^\alpha \texttt{T} \to \texttt{F}\}$:

$$\dfrac{\dfrac{}{F \vdash \texttt{x} ::^\alpha \texttt{T} \mid \{\texttt{x} ::^\alpha \texttt{T}\}}\ Var \quad \dfrac{}{F \vdash \texttt{False} ::^\alpha \texttt{F} \mid \varnothing}\ Con \quad \dfrac{}{F \vdash \texttt{True} ::^\alpha \texttt{T} \mid \varnothing}\ Con}{F \vdash \texttt{case x of } \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\} ::^\alpha \texttt{F} \mid \{\texttt{x} ::^\alpha \texttt{T}\}}\ Case$$

Note that the second case branch is ignored in the application of the *Case* rule since its result type `T` is not compatible with the overall result type `F`. Similarly, the following types (among others) can be derived to be well typed:

$\texttt{not} ::^\alpha \texttt{F} \to \texttt{T}$
$\texttt{not} ::^\alpha \{\texttt{False}, \texttt{True}\} \to Any$

Finally, we consider the conjunction operator `(&&)` defined by

```
x && y = case x of { True   →  y
                   ; False  →  False }
```

$\texttt{(\&\&)} ::^\alpha \texttt{T}, \texttt{T} \to \texttt{T}$ is a well-typing since the following derivation holds for the type environment $F = \{\texttt{(\&\&)} ::^\alpha \texttt{T}, \texttt{T} \to \texttt{T}\}$:

$$\dfrac{\dfrac{}{F \vdash \texttt{x} ::^\alpha \texttt{T} \mid \{\texttt{x} ::^\alpha \texttt{T}\}}\ Var \quad \dfrac{}{F \vdash \texttt{y} ::^\alpha \texttt{T} \mid \{\texttt{y} ::^\alpha \texttt{T}\}}\ Var \quad \dfrac{}{F \vdash \texttt{False} ::^\alpha \texttt{F} \mid \varnothing}\ Con}{F \vdash \texttt{case x of } \{\texttt{True} \to \texttt{y}; \texttt{False} \to \texttt{False}\} ::^\alpha \texttt{T} \mid \{\texttt{x} ::^\alpha \texttt{T}, \texttt{y} ::^\alpha \texttt{T}\}}\ Case$$

The correctness of our type analysis can be stated by the following theorem:

Theorem 1. *If a program P is well typed w.r.t. a type environment F for P, then each $f :: ^\alpha a_1, \ldots, a_n \to a \in F$ is correct.*

We have seen in various examples that there does not exist a meaningful most general type for each function. Although we could type each function f by

$f ::^{\alpha} Any, \ldots, Any \to Any$, this type does not provide any useful information about required arguments. Thus, the inference of types is more complex than in classical type inference systems [14].

Instead, we use the idea to compute types by a fixpoint analysis [12]. The analysis is started with no information about each function (e.g., $f ::^{\alpha} Any, \ldots, Any \to Any$) and uses the rules in Fig.1 to compute values for required arguments. If the analysis computes some more precise information about the result of a function, i.e., a result type like $\{C\}$, then the analysis is started again with all constructors (of the corresponding concrete data type): if $C_1, \ldots, C_k$ are all constructors of the data type to which C belongs, we restart the analysis with the environment containing $f ::^{\alpha} Any, \ldots, Any \to C_i$ (for $i = 1, \ldots, k$). In this way we obtain more meaningful results without testing all constructors from the beginning, which seems a good compromise between efficiency and precision of the analysis.

5 Implementation

The analysis of required values is a prerequisite to implement the transformation of equalities as discussed in Sect. 3. To implement the analysis, we used the Curry analysis system CASS [18]. CASS is a generic program analysis system which provides an infrastructure to implement new bottom-up analyses. CASS requires only the definition of the abstract domain and the abstract operations to compute the abstract values for each function based on given abstract values for the operations on which the operation to be analyzed depend. Then the reading, parsing, and analysis of modules in their import order and the fixpoint computations are managed by CASS.

The results of the analysis are used to transform Boolean equations as follows. For each function f, we apply the rules in Fig. 1 in order to compute the required values at an occurrence of an expression of the form e_1`==`e_2 in the right-hand side of the rule of f. If the abstract type is always $\{$`True`$\}$, we replace this expression by e_1`=:=`e_2. This is justified by the fact that the result `False` is never required when this function must be evaluated.

Hence, our implementation automatically transforms the occurrences of "`==`" shown in Sect. 3. Since this transformation is performed on FlatCurry programs, it can be easily integrated into the compilation chain for Curry programs. In fact, the transformation is fully integrated into the current releases of the Curry systems PAKCS [17] and KiCS2 [10].

In order to evaluate the usefulness of our transformation, we tested it on some benchmarks. As discussed in Sect. 2, our transformation can reduce infinite search spaces into finite ones. For instance, the expression

```
cond (xs == ys && xs++ys == [True]) True
```

has an infinite search space, whereas the transformed expression

```
cond (xs =:= ys && xs++ys =:= [True]) True
```

has a finite search space. Even in the case of finite search spaces, replacing Boolean equations by equational constraints often has a good impact on the run

Expression	==	=:=
`last 10`	0.01	0.00
`last 15`	0.41	0.00
`last 20`	13.12	0.00
`fromPeano (half (toPeano 10000))`	31.09	12.98
`grep`	0.54	0.37
`simplify`	22.41	16.68
`varInExp`	0.95	0.42

Fig. 2. Benchmarks: comparing Boolean equations and equational constraints

time since non-deterministic search is transformed to deterministic bindings, as demonstrated by some benchmarks.

The benchmarks were performed with the Curry implementation KiCS2 [10] which evaluates the Boolean equality operator by narrowing with the "`==`" rules shown in Sect. 2 and the equational constraints by managing variable bindings [11]. The benchmarks were executed on a Linux machine (Debian 8.0) with an Intel Core i7-4790 (3.60Ghz) processor and 8GiB of memory. KiCS2 (Version 0.4.0) has been used with the Glasgow Haskell Compiler (GHC 7.6.3, option -O2) as its backend. The timings were performed with the time command measuring the execution time to compute all solutions (in seconds) of a compiled executable for each benchmark as a mean of three runs. The programs used for the benchmarks are `last` n (compute the last element of a list containing $n-1$ variables and `True` at the end), `half` (compute the half of a Peano number using logic variables), `grep` (string matching based on a non-deterministic specification of regular expressions [5]), `simplify` (simplify a symbolic arithmetic expression), and `varInExp` (non-deterministically return a variable occuring in a symbolic arithmetic expression). Fig. 2 shows the execution times to evaluate some expressions without (`==`) or with (`=:=`) our transformation. As expected, the creation and traversal of a large search space introduced by "`==`" is much slower than manipulating variable bindings by "`=:=`".

6 Practical Evaluation

In this section we discuss some practical experiences we made with our transformation tool.

As mentioned above, the transformation tool is integrated into the compilation chain of the recent releases of the Curry systems PAKCS [17] and KiCS2 [10]. The configuration files of these systems allow the user to set the following usage modes: "off" (do not apply this transformation), "full" (analyze programs as described in Sect. 4 and perform the transformation described in Sect. 5), or "fast" (which is the default: use pre-computed analysis information of standard operations from the prelude to perform the transformation described in Sect. 5). The advantage of the "fast" mode is that it is a reasonable compromise between effectiveness and efficiency. In this mode, the transformation described

in Sect. 5 does not perform the fixpoint analysis of Sect. 4, but it simply uses the pre-computed abstract types for the most relevant Boolean functions defined in the prelude, like `(&&)` (conjunction), `(||)` (disjunction), `not` (negation), and the conditional operator `cond`. The transformation itself can be efficiently performed by considering only functions that contain occurrences of "`==`". Thus, even large modules are transformed without any perceivable slowdown in the compilation chain.

Although the "fast" mode uses only the results of a few Boolean operations defined in the standard prelude, it is sufficient in practice. For our tests, we replaced in existing Curry programs all equational constraints by Boolean equalities. Our transformation tool was able to transform almost all of them into constraints. The rare cases where this was not possible are operations that return constraints to be solved. For instance, consider an operation that returns `True` if its three arguments are pairwise equal:

```
equ3 x y z = x==y && y==z
```

Obviously, our transformation cannot replace the Boolean equalities by equational constraints since this may cause a loss of solutions. For instance, for Boolean values, the expression "`not (equ3 x y z)`" evaluates to `True` by binding `x` to `True` and `y` to `False` (among other solutions). Such solutions would be lost if we replace "`==`" by "`=:=`". However, if it is intended that the operation `equ3` should only be used for "positive" evaluations, one can easily redefine it by

```
equ3 x y z | x==y && y==z = True
```

With this definition, our transformation is able to replace both occurrences of "`==`" by "`=:=`".

7 Conclusions

We have presented an automatic method to replace Boolean equalities by equational constraints in functional logic programs. This can be done only if it is ensured that `True` is required as the result of a Boolean equality, which is the case, e.g., in conditions of rules. To this aim, we developed an analysis for required values. This analysis can be seen as a non-standard type inference where abstract types represent sets of required values. The results of this analysis are then used to drive the actual program transformation.

Our transformation method has the following advantages over the current design of functional logic languages like Curry:

1. The source language becomes simpler. Since equational constraints are considered as an optimization of Boolean equality, the existing type `Success` can be omitted (as proposed in [6]). This has the consequence that quite similar operations, like inequalities between values (`(<=)`), do not need to be duplicated for the type of Boolean and constraints, as it is currently the case.

2. It is not necessary to consider the subtle differences between the type `Bool` and `Success` and the operators "`==`" and "`=:=`". A programmer uses "`==`" only (where the operator "`=:=`" must still be provided for the transformation target and in exceptional cases where a programmer wants to write efficient code independent of a program transformation). This also simplifies the teaching of declarative multi-paradigm languages [15].
3. Equational constraints can be considered as an optimized implementation of Boolean equalities. Hence, from a declarative point of view, one has to deal with Boolean equalities only, which are easy to define by standard rewrite rules as shown in Sect. 2.

If the target system also supports disequality constraints, as proposed in early functional logic languages [7, 20], one could exploit them in an extension of our transformation tool. For instance, if an expression e_1`==`e_2 requires always `False` as its result, one could replace it by e_1`=/=`e_2, where the operator "`=/=`" represents a disequality constraint. This might be more efficient than guessing values by narrowing with the standard "`==`" rules but requires a specific implementation of a solver for "`=/=`".

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR 2002)*, pages 1–16. Springer LNCS 2664, 2002.
3. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.
7. P. Arenas-Sánchez, A. Gil-Luezas, and F.J. López-Fraguas. Combining lazy narrowing with disequality constraints. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 385–399. Springer LNCS 844, 1994.
8. D. Bert and R. Echahed. Abstraction of conditional term rewriting systems. In *Proc. of the 1995 International Logic Programming Symposium*, pages 147–161. MIT Press, 1995.
9. D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proc. Third International Workshop on Static Analysis*, pages 178–192. Springer LNCS 724, 1993.

10. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
11. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.
12. P. Cousot. Types as abstract interpretations. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 316–331, 1997.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
14. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.
15. M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.
16. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
17. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2014.
18. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
19. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
20. H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming*. MIT Press, 1992.
21. A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. International Symposium on Programming*, pages 269–281. Springer LNCS 83, 1980.
22. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
23. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
24. J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.
25. J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
26. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
27. D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.