Under consideration for publication in Formal Aspects of Computing

Transforming Boolean Equalities into Constraints[1]

Sergio Antoy[1] and 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.

Keywords: functional logic programming; program analysis; program transformation; unification

1. Motivation

Functional as well as logic programming languages are based on the common idea of specifying 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`:[2]

Correspondence and offprint requests to: Michael Hanus, Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany. e-mail: mh@informatik.uni-kiel.de

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

[2] We use the syntax of Haskell [PJ03] for functional programs.

```
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 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, the intent in the first rule is not to evaluate `X=[]` to `True` or `False`, but to ensure that this equality holds for applying this rule, i.e., the equality 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 holds. 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 [Rob65] 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 [AH10, Han13] for recent surveys). In particular, the functional logic language Curry [Han16] 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 [Rob65]. 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`.

The following example highlights the differences between these two notions of equality. Consider the following type:

```
data Color = White | Black
```

and let t and u be colors. We discuss various cases. If t and u are the same color, then t `==` u returns `True` whereas t `=:=` u succeeds. If t and u are different colors, then t `==` u returns `False` whereas t `=:=` u fails (the computation is aborted). A more interesting situation occurs when one side is a variable. Suppose that t is a variable and u is `White`. In this case, t `==` u either binds t to `White` and returns `True` or it binds t to `Black` and returns `False`. By contrast, t `=:=` u binds t to `White` only and succeeds. Finally, if both t and u are variables, "`==`" binds them to all possible combinations of colors and return either `True` or `False` accordingly, whereas "`=:=`" unifies the variables and succeeds. "`==`" in Curry is a conservative extension of "`==`" in popular functional languages, e.g., Haskell. The behavior we just described conforms with version 0.9.0 of Curry [Han16]. In previous versions [Han12], "`==`" was *rigid*, i.e., the evaluation would suspend when either side, or both, were variables.

Although the distinction between these two kinds of equalities is present in Curry from its early design [Han97], 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 (in earlier versions of Curry [Han12]), 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 for transforming

Boolean equalities into constraint equalities, if it is appropriate. Hence, we automatically obtain the advantages 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.

In the next section, we review the main concepts of functional logic programming and Curry. Sect. 3 discusses the basic ideas of our transformation method. The analysis of required values is formalized and proved to be correct in Sect. 4. The actual implementation is sketched in Sect. 5 and followed by a discussion of some practical experiences with our transformation method in Sect. 6. Some related work is discussed in Sect. 7 before we conclude in Sect. 8.

2. Functional Logic Programming and Curry

We briefly review the elements of functional logic programming and Curry which are necessary to understand the contents of this paper. More details can be found in recent surveys on functional logic programming [AH10, Han13] and in the language report [Han16].

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). Curry's syntax is close to Haskell's [PJ03], 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. $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β (where β can also be a functional type, i.e., functional types are "curried"), and the application of an operation f to an argument e is denoted by juxtaposition ("$f\ e$").

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. In addition to Haskell, Curry allows *free* (*logic*) *variables* in conditions and right-hand sides of rules and expressions evaluated by an interpreter. These variables are a source of non-determinism in computations much in the same way of logic programming. Function calls with arguments that are or contain free variables are evaluated by narrowing.

Narrowing [Sla74, Red85] is a computation used in Curry with the same intent as resolution, i.e., guessing values for unknown information with guesses that keep the computation going. Narrowing extends and generalizes resolution in that resolution binds variables through predicates whereas narrowing through functions of any type. Furthermore, since function calls can be nested, free variables are bound only when a value is demanded by the computation. Curry narrows with possibly non-most-general unifiers to ensure the optimality of computations [AEH00]. Narrowing contributes elegant and conceptually simple solutions to non-trivial programming problems [Ant10].

Example 2.1. We present the above features 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:[3]

```
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 (`a` is a type variable ranging over all types and the

[3] 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.

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`. Here, "`=:=`" is the correct choice of equality since we have no interest for the value of `x` when the equation is falsified. In this situation it is not guaranteed 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
False == False = True
True  == False = False
False == True  = False

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

As already discussed in [AH14], 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, [AH14] proposes to omit the type `Success` from the definition of Curry (as done in the current version of the language [Han16]) and do not use the operator "`=:=`" in source programs. 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
False =:= False = True

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

Thus, "`=:=`" yields `True` for identical data terms or fails.[4] Operationally, these rules are not applied by narrowing but combined with the unification principle [Rob65], 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 [Han16] 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

[4] Note that this conforms with the current version of the language Curry [Han16]. In previous versions, equational constraints have the specific result type `Success`. A detailed discussion about the reasons to replace the type `Success` by `Bool` can be found in [AH14].

"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 [AHH+05], compile programs [BHPR11, HAB+16], or implement analyzers [HS14] 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)
	|	$C\ \overline{e_n}$	(constructor application)
	|	$f\ \overline{e_n}$	(function application)
	|	$case\ e_0\ of\ \{\overline{p_k \to e_k}\}$	(case distinction)
	|	e_1 `?` 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[5] has the form $case\ e\ of\ \{C_1\ \overline{x_{n_1}} \to e_1, \ldots, C_k\ \overline{x_{n_k}} \to e_k\}$ $(k > 0)$, 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 [Rey72]. 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 [Han13, War82] for more details).

Conditional rules are not present in FlatCurry since, as shown in [Ant01], 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 [BHPR11]):

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

[5] Since we do not discuss residuation and concurrent computations, we also omit the difference between rigid and flexible case expressions [Han13].

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, e.g., `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.[6]

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 [Myc80]. A strictness analysis provides information about the necessary demand of a 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

[6] The latter equality could also be improved when disequality constraints [AGL94, KLMR92] 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.

[ACE+02] or in the abstraction of term rewriting systems [BE95, BEØ93]), 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 [CC77], we define for each type τ an *abstract domain* τ^α, i.e., a set of *abstract values*, as follows. If $\mathcal{C}_\tau = \{C_1, \ldots, C_k\}$ is the set of all constructors of type τ, then $\tau^\alpha = \mathcal{C}_\tau \cup \{\bot, \top\}$, i.e., an abstract value of τ^α is either a constructor of type τ or the constants $\bot$ or $\top$. For instance, the abstract domain for Boolean values is

```
Bool^α = { ⊥, True, False, ⊤ }
```

Abstract values are ordered as a flat domain, i.e., $\bot \sqsubseteq a$ and $a \sqsubseteq \top$ for all abstract values a. Thus, for two constructors C_1 and C_2 considered as abstract values with $C_1 \neq C_2$, the least upper bound $C_1 \sqcup C_2$ is $\top$ and the greatest lower bound $C_1 \sqcap C_2$ is $\bot$.

The meaning of an abstract value a, i.e., the concretization $[\![a]\!]$ of a, is defined as follows (where $root(e)$ denotes the symbol at the root of the expression e):

$$[\![a]\!] = \begin{cases} \varnothing & \text{if } a = \bot \\ \{e \mid e \text{ expression}\} & \text{if } a = \top \\ \{e \mid root(e) = C\} & \text{if } a = C \end{cases}$$

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.

Proposition 4.1. For all abstract values a_1 and a_2 with $a_1 \sqsubseteq a_2$, $[\![a_1]\!] \subseteq [\![a_2]\!]$ holds.

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

```
(&&) ::α 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 in other words: unless both arguments are evaluated to `True`, the result cannot be `True`.

Definition 4.2. A typing $f ::^\alpha a_1, \ldots, a_n \to a$ of a function f is *correct* iff the following implication holds for all expressions $e_1, \ldots, e_n$: if $f\ e_1 \ldots e_n$ 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 $f ::^\alpha \top, \ldots, \top \to \top$ (with appropriate numbers of arguments) is correct since any expression is an element of $[\![\top]\!]$. Clearly, defined functions can have more than one correct typing. For instance, consider the Boolean negation and conjunction operators defined by

```
not True  = False          True  && x = x
not False = True           False && _ = False
```

Then the negation operator `not` has the types

```
not ::α True → False
not ::α False → True
```

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

```
(&&) ::α True,True → True
(&&) ::α ⊤,⊤ → False
```

The first type of "`&&`" expresses the fact that the expression e_1 `&&` e_2 cannot evaluate to `True` unless both e_1 and e_2 evaluate to `True`.

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 "`&&`"). The rule of "`&&`", shown later, executes a case distinction on the first argument, hence 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

$$\texttt{(\&\&)} ::^\alpha \texttt{False}, \top \to \texttt{False}$$

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

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 elements of the form $f ::^\alpha a_1, \ldots, a_n \to a$ or $f ::^\alpha \bot \to a$. The first form is related to the meaning defined in Def. 4.2, whereas the second form is used to express impossible evaluations. This is essential to derive precise information about required arguments.

For instance, consider an operator `solve` which enforces positive evaluations for Boolean expressions:

```
solve True = True
```

Typically, `solve` is used in the top-level (i.e., initial expressions to be evaluated) to get solutions to some equation, e.g.,

```
solve (xs ++ ys == [1,2,3])
```

By the use of `solve`, we ignore the computation of "non-solutions" like {`xs=[],ys=[]`}. Intuitively,

$$\texttt{solve} ::^\alpha \texttt{True} \to \texttt{True}$$

is a correct typing. This expresses the fact that one can compute the result `True` only if the argument evaluates to `True`. We can use this information to transform the above expression into the constraint

```
solve (xs ++ ys =:= [1,2,3])
```

Now consider the function `f` defined by

```
f True  _ = False
f False x = solve x
```

A correct type for this function is

$$\texttt{f} ::^\alpha \texttt{True}, \top \to \texttt{False}$$

since the second rule of `f` cannot be used to derive `False`. This reasoning requires the fact that `solve` cannot yield the result `False`, but this is not expressed by the above type of `solve`. For this purpose, we use the type

$$\texttt{solve} ::^\alpha \bot \to \texttt{False}$$

to express the impossibility to evaluate `solve` to `False`.

Apart from the types of functions, we also need to reason about the required values of argument variables in order to compute some value of an expression. Therefore, our type analysis also uses a *variable type environment* E which contains for each variable x at most one variable type $x ::^\alpha a$. The absence of a variable type is interpreted as type $\top$, i.e., the sets $\{x ::^\alpha a\}$ and $\{x ::^\alpha a, y ::^\alpha \top\}$ denote the same variable type environment, and the empty set $\varnothing$ denotes a variable type environment where all variables have type $\top$. We order variable type environments element-wise, i.e., $E_1 \sqsubseteq E_2$ iff, for all variables x with $x ::^\alpha a_1 \in E_1$ and $x ::^\alpha a_2 \in E_2$, $a_1 \sqsubseteq a_2$ holds. Hence, 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, e.g., $\{x ::^\alpha \texttt{True}, y ::^\alpha \texttt{True}\} \sqcup \{x ::^\alpha \texttt{False}\} = \{x ::^\alpha \top, y ::^\alpha \top\}$. Observe that $y ::^\alpha \top$ 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. We also assume a least element $\bot$ for all variable environments, i.e., $\bot \sqsubseteq E$ for all variable environments E. Since $\bot$ does not define a required value for any variable, it is used to specify the impossibility to compute some value. Note that the meaning of the variable type environment $\varnothing$ is different from $\bot$: whereas $\varnothing$ does not put any requirement on the variables, i.e., it denotes the set of all concrete substitutions, $\bot$ denotes a conflict, i.e., there is no concrete substitution that agrees with abstract value $\bot$.

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

$$\textit{Con1}\quad \frac{}{\varnothing \vdash^F C\ e_1 \ldots e_n ::^\alpha a} \quad \text{if } C \text{ and } a \text{ are compatible}$$

$$\textit{Con2}\quad \frac{}{\bot \vdash^F C\ e_1 \ldots e_n ::^\alpha a} \quad \text{if } C \text{ and } a \text{ are not compatible}$$

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

$$\textit{Fun1}\quad \frac{E_1 \vdash^F e_1 ::^\alpha a_1 \ \ldots\ E_n \vdash^F e_n ::^\alpha a_n}{\bigsqcap\{E_i \mid a_i \neq \top\} \vdash^F f\ e_1 \ldots e_n ::^\alpha a} \quad \text{if } f ::^\alpha a_1, \ldots, a_n \to a \in F$$

$$\textit{Fun2}\quad \frac{}{\bot \vdash^F f\ e_1 \ldots e_n ::^\alpha a} \quad \text{if } f ::^\alpha \bot \to a \in F$$

$$\textit{Case}\quad \frac{\begin{array}{ll} E_0 \vdash^F e_0 ::^\alpha a_0 \ \ E_1 \vdash^F e_1 ::^\alpha a \ \ldots\ E_j \vdash^F e_j ::^\alpha a & \text{where } a_0 = C_1 \sqcup \ldots \sqcup C_j \text{ (if } j > 0\text{)} \\ \bot \vdash^F e_{j+1} ::^\alpha a \ \ldots\ \bot \vdash^F e_n ::^\alpha a & \text{or } a_0 = \top \text{ (if } j = 0\text{)} \end{array}}{E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j) \vdash^F \mathit{case}\ e_0\ \mathit{of}\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\} ::^\alpha a}$$

Fig. 1. Abstract typing rules for FlatCurry expressions

In this sense, a variable type environment containing $x ::^\alpha \bot$ is identical to the environment $\bot$ since both denote the empty set of concrete substitutions.

The (abstract) typing rules are shown in Fig. 1. The notation $E \vdash^F e ::^\alpha a$ (with $a \neq \bot$) should be read as: "E are the required values of variables in order to evaluate e to some value of type a w.r.t. type environment F." Having this interpretation in mind, the typing rules can be read as follows:

- Rule *Var* expresses that computing a particular value for a variable is also the requirement for that variable.
- Rule *Con1* does not put requirements on variables since the term is already a value compatible with the abstract result. In contrast, rule *Con2* expresses by the variable environment $\bot$ that it is impossible to get a value of the abstract result.
- Rule *Or* joins the requirements expressed by the variable type environments to evaluate a choice expression of some type. Note that it is sufficient that one argument of a choice expression can be evaluated to the demanded value. For instance, $\varnothing \vdash^F \texttt{True ? False} ::^\alpha \texttt{True}$ is derivable by this rule since $\varnothing \vdash^F \texttt{True} ::^\alpha \texttt{True}$ and $\bot \vdash^F \texttt{False} ::^\alpha \texttt{True}$ are derivable by rules *Con1* and *Con2*, respectively, and $\varnothing \sqcup \bot = \varnothing$.
- Rule *Fun1* requires well-typed arguments and an appropriate function typing to apply a function but joins only the requirements of arguments where a value is definitely required, since other arguments might not be evaluated. Rule *Fun2* can be applied if there is no typing of the function for the given abstract result type.
- Rule *Case* requires that the constructors of the patterns in the various branches must be contained in the type of the discriminating expression (where there are no requirements, i.e., $a_0 = \top$, for the extreme case $j = 0$). However, branches which cannot be evaluated to the overall result type are ignored for the type of the discriminating expression. This refinement is essential to obtain precise information about required arguments, as shown in subsequent examples. Since the order of the branches in the case construct is irrelevant, we move at the end of the line the branches that cannot be evaluated to simplify the formulation.

In order to obtain information about impossible result types, as exploited in the rule *Case*, we allow two kinds of function types: $f ::^\alpha a_1, \ldots, a_n \to a$ and $f ::^\alpha \bot \to a$. They are related to different kinds of typings as defined next.

Definition 4.3. A program P is *well typed w.r.t. a type environment* F for P if, for each function defined

by the rule $f\ x_1 \ldots x_n = e \in P$ and each $f :: ^\alpha a_1, \ldots, a_n \to a \in F$, $\{x_1 :: ^\alpha a_1, \ldots, x_n :: ^\alpha a_n\} \vdash^F e :: ^\alpha a$ is derivable by the rules in Fig. 1, and for each $f :: ^\alpha \bot \to a \in F$, $\bot \vdash^F e :: ^\alpha a$ is derivable by the rules in Fig. 1.

Since F is usually fixed in all examples and proofs, we omit it from the symbol $\vdash$ in the following. We show the usage of this type system in a few examples that are relevant for the application intended by this paper. In these examples, `True` and `False` overload Boolean values and the corresponding abstract types. For the first example, consider the operator `solve` introduced above by the rule

```
solve True = True
```

In FlatCurry, `solve` can be defined by the rule

```
solve x = case x of { True  →  True }
```

Using our inference rules, we can derive the type $\texttt{solve} :: ^\alpha \texttt{True} \to \texttt{True}$ as follows:

$$\dfrac{\dfrac{}{\{\texttt{x} :: ^\alpha \texttt{True}\} \vdash \texttt{x} :: ^\alpha \texttt{True}}\ Var \quad \dfrac{}{\varnothing \vdash \texttt{True} :: ^\alpha \texttt{True}}\ Con1}{\{\texttt{x} :: ^\alpha \texttt{True}\} \vdash \texttt{case x of } \{\texttt{True} \to \texttt{True}\} :: ^\alpha \texttt{True}}\ Case$$

The second type of solve, $\texttt{solve} :: ^\alpha \bot \to \texttt{False}$, can be derived by:

$$\dfrac{\dfrac{}{\{\texttt{x} :: ^\alpha \top\} \vdash \texttt{x} :: ^\alpha \top}\ Var \quad \dfrac{}{\bot \vdash \texttt{True} :: ^\alpha \texttt{False}}\ Con2}{\bot \vdash \texttt{case x of } \{\texttt{True} \to \texttt{True}\} :: ^\alpha \texttt{False}}\ Case$$

Our second 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 can be shown well-typed w.r.t. $\texttt{cond} :: ^\alpha \texttt{True}, \top \to \top$ so that we can deduce that the first argument is required to be `True` in order to compute any value:

$$\dfrac{\dfrac{}{\{\texttt{x} :: ^\alpha \texttt{True}\} \vdash \texttt{x} :: ^\alpha \texttt{True}}\ Var \quad \dfrac{}{\{\texttt{y} :: ^\alpha \top\} \vdash \texttt{y} :: ^\alpha \top}\ Var}{\{\texttt{x} :: ^\alpha \texttt{True}, \texttt{y} :: ^\alpha \top\} \vdash \texttt{case x of } \{\texttt{True} \to \texttt{y}\} :: ^\alpha \top}\ Case$$

Note that this rule is also well typed w.r.t. $\texttt{cond} :: ^\alpha \top, \top \to \top$, but this typing provides less precise information about required arguments.

For the next example, consider the negation operator `not` which is defined in FlatCurry by

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

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

$$\dfrac{\dfrac{}{\{\texttt{x} :: ^\alpha \texttt{True}\} \vdash \texttt{x} :: ^\alpha \texttt{True}}\ Var \quad \dfrac{}{\varnothing \vdash \texttt{False} :: ^\alpha \texttt{False}}\ Con1 \quad \dfrac{}{\bot \vdash \texttt{True} :: ^\alpha \texttt{False}}\ Con2}{\{\texttt{x} :: ^\alpha \texttt{True}\} \vdash \texttt{case x of } \{\texttt{True} \to \texttt{False}; \texttt{False} \to \texttt{True}\} :: ^\alpha \texttt{False}}\ Case$$

In the application of the *Case* rule, the discriminating constructor `False` of the second case branch is ignored to compute the required type of `x`, since the result value `True` of this branch is not compatible with the overall result type `False`. Similarly, the following types (among others) can be derived to be well typed:

$\texttt{not} :: ^\alpha \texttt{False} \to \texttt{True}$
$\texttt{not} :: ^\alpha \top \to \top$

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

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

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

$$\dfrac{\dfrac{}{\{\texttt{x} ::^\alpha \texttt{True}\} \vdash \texttt{x} ::^\alpha \texttt{True}}\ Var \quad \dfrac{}{\{\texttt{y} ::^\alpha \texttt{True}\} \vdash \texttt{y} ::^\alpha \texttt{True}}\ Var \quad \dfrac{}{\bot \vdash \texttt{False} ::^\alpha \texttt{True}}\ Con2}{\{\texttt{x} ::^\alpha \texttt{True}, \texttt{y} ::^\alpha \texttt{True}\} \vdash \texttt{case x of } \{\texttt{True} \to \texttt{y}; \texttt{False} \to \texttt{False}\} ::^\alpha \texttt{True}}\ Case$$

Note that our typing rules allow the derivation of less precise typings, like $\texttt{(\&\&)} ::^\alpha \top, \top \to \top$, but no derivation with stronger information about arguments. For instance, the operation

```
f x = True
```

could be well typed with $\texttt{f} ::^\alpha \top \to \texttt{True}$, but the type $\texttt{f} ::^\alpha \texttt{True} \to \texttt{True}$ cannot be derived. This is intended since the latter type is not correct: any application of `f` can be derived to `True`, i.e., it is not required that the argument needs to be `True`.

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

Theorem 4.4. 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.

In order to prove this main theorem, we need a couple of statements about the typing rules, which are stated and proved in the following. The next lemma shows that the type system can derive that required values in variable type environments are actually required, i.e., if such a variable is instantiated with a constructor different from the required one, one can derive the impossibility of typing this instantiation.

Lemma 4.5. Let P be a well-typed program w.r.t. a type environment F for P and $E \vdash e ::^\alpha a$. If $x ::^\alpha C \in E$ and the substitution σ replaces x by a different constructor $C' \neq C$, i.e., $\sigma(x) = C' \ldots$, then $\bot \vdash \sigma(e) ::^\alpha a$.

Proof. We prove the claim by structural induction on the expression e.

- If e is a variable, then $e = x$ (otherwise, $x ::^\alpha C \notin E$ by rule *Var*) and $a = C$. Since $\sigma(x) = C' \ldots$ and $C \neq C'$, $\bot \vdash \sigma(e) ::^\alpha a$ by rule *Con2*.
- e cannot be constructor-rooted, otherwise $x ::^\alpha C \notin E$ by rules *Con1* and *Con2*.
- If $e = e_1 \,?\, e_2$, then, for $i = 1, 2$, $E_i \vdash e_i ::^\alpha a$ and $E = E_1 \sqcup E_2$ by rule *Or*. Since $x ::^\alpha C \in E$, either $x ::^\alpha C \in E_i$ or $x ::^\alpha \bot \in E_i$, for $i = 1, 2$. In the first case, $\bot \vdash \sigma(E_i) ::^\alpha a$ by the induction hypothesis. In the second case, $\bot \vdash \sigma(e_i) ::^\alpha a$ since $\bot \vdash e_i ::^\alpha a$ and the type of x is not relevant for a. Hence $\bot \vdash \sigma(e_1 \,?\, e_2) ::^\alpha a$.
- If $e = \mathit{case}\ e_0\ \mathit{of}\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\}$, then, by rule *Case*, $E_0 \vdash e_o ::^\alpha a_0$ and $x ::^\alpha C \in E_0$ (the case $x ::^\alpha \top \in E_0$ and $x ::^\alpha C \in E_1 \sqcup \ldots \sqcup E_j$ can be treated similarly to the *Or* case above). By the induction hypothesis, $\bot \vdash \sigma(e_o) ::^\alpha a_0$ so that $\bot \vdash \sigma(e) ::^\alpha a$ by rule *Case*.
- Consider the final case $e = f\ e_1 \ldots e_n$. Rule *Fun2* cannot be applied, otherwise $x ::^\alpha C \notin E$. Hence rule *Fun1* is applicable so that $f ::^\alpha a_1, \ldots, a_n \to a \in F$ and there is at least one $i \in \{1, \ldots, n\}$ with $E_i \vdash e_i ::^\alpha a_i$, $a_i \neq \top$, and $x ::^\alpha C \in E_i$. By the induction hypothesis, $\bot \vdash \sigma(e_i) ::^\alpha a_i$ so that $\bot \vdash \sigma(f\ e_1 \ldots e_n) ::^\alpha a$ by rule *Fun1*.

□

Next we show that the derivation of impossible typings is closed under substitutions, i.e., if we derive that some expression is not evaluable to some constructor value, the same holds for all instantiated expressions.

Lemma 4.6. Let P be a well-typed program w.r.t. a type environment F for P, $\bot \vdash e ::^\alpha a$ and σ a substitution. Then $\bot \vdash \sigma(e) ::^\alpha a$.

Proof. We prove the claim by structural induction on the expression e.

- The case that e is a variable cannot occur (since $a \neq \bot$).
- If $e = C\ e_1 \ldots e_n$, then C and a are not compatible by rule *Con2*. Since $\sigma(e) = C\ \sigma(e_1) \ldots \sigma(e_n)$, $\bot \vdash \sigma(e) ::^\alpha a$ again by rule *Con2*.
- If $e = e_1 \,?\, e_2$, then, for $i = 1, 2$, $\bot \vdash e_i ::^\alpha a$. By the induction hypothesis, $\bot \vdash \sigma(e_i) ::^\alpha a$ so that we have $\bot \vdash \sigma(e) ::^\alpha a$.

- If $e = case\ e_0\ of\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\}$, then, by rule *Case*, $E_0 \vdash e_o ::^\alpha a_0$, $E_i \vdash e_i ::^\alpha a$ $(i = 1, \ldots, j)$, and $\bot = E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j)$. If $E_0 = \bot$, then $\bot \vdash e_o ::^\alpha a_0$ and, by the induction hypothesis, $\bot \vdash \sigma(e_o) ::^\alpha a_0$ so that the claim holds by rule *Case*. Similarly, if $E_1 \sqcup \ldots \sqcup E_j = \bot$, then $\bot \vdash e_i ::^\alpha a$ for $i = 1, \ldots, j$ and, by the induction hypothesis, $\bot \vdash \sigma(e_i) ::^\alpha a$, for $i = 1, \ldots, j$, so that that the claim holds by rule *Case*. Otherwise, there is some variable x with $x ::^\alpha \bot \in E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j)$ and $x ::^\alpha C \in E_0$, $x ::^\alpha C' \in E_i$ for some $i \in \{1, \ldots, j\}$ such that $C \sqcap C' = \bot$, i.e., $C \neq C'$. By Lemma 4.5, σ cannot instantiate e_0 and e_i such that x satisfies both required constructors, i.e., it is impossible to type $\sigma(e_i)$ as well as $\sigma(e_j)$ with some environment different from $\bot$, which implies the claim.
- Finally, consider the case $e = f\ e_1 \ldots e_n$. If rule *Fun2* has been applied, we can also apply it to $\sigma(e)$. Hence we assume that rule *Fun1* has been applied, i.e., $f ::^\alpha a_1, \ldots, a_n \to a \in F$ and $E_i \vdash e_i ::^\alpha a_i$ for $i = 1, \ldots, n$. If there is some $i \in \{1, \ldots, n\}$ with $E_i = \bot$ and $a_i \neq \top$, then, by the induction hypothesis, $\bot \vdash \sigma(e_i) ::^\alpha a_i$ so that $\bot \vdash \sigma(e) ::^\alpha a$ by rule *Fun1*. Otherwise, there are $i, j \in \{1, \ldots, n\}$, $i \neq j$, such that $a_i \neq \top$, $a_j \neq \top$, $x ::^\alpha C \in E_i$, $x ::^\alpha C' \in E_j$ such that $C \sqcap C' = \bot$, i.e., $C \neq C'$. By Lemma 4.5, σ cannot instantiate e_i and e_j such that x satisfies both required constructors, i.e., it is impossible to type $\sigma(e_i)$ as well as $\sigma(e_j)$ with an environment different from $\bot$. Hence $\bot \vdash \sigma(f\ e_1 \ldots e_n) ::^\alpha a$.

□

The next lemma states that if we instantiate a required variable with an expression that cannot be meaningfully typed, then one can derive the impossibility of typing this instantiation.

Lemma 4.7. Let P be a well-typed program w.r.t. a type environment F for P, $\{x :: C\} \vdash e ::^\alpha a$, and $\bot \vdash e' ::^\alpha C$. If $\sigma = \{x \mapsto e'\}$, then $\bot \vdash \sigma(e) ::^\alpha a$.

Proof. We prove the claim by structural induction on the expression e.

- If e is a variable, then $e = x$, $a = C$, and $\sigma(e) = e'$. Since we assumed $\bot \vdash e' ::^\alpha C$, we have $\bot \vdash \sigma(e) ::^\alpha a$.
- e cannot be constructor-rooted, since none of the rules *Con1* and *Con2* can require $x ::^\alpha C$.
- If $e = e_1\,?\,e_2$, then, for $i = 1, 2$, $E_i \vdash e_i ::^\alpha a$ and $\{x :: C\} = E_1 \sqcup E_2$ by rule *Or*. Hence $x ::^\alpha b_i \in E_i$ $(i = 1, 2)$ such that $b_1 \sqcup b_2 = C$. Then either $b_1 = b_2 = C$ or $b_1 = \bot$ or $b_2 = \bot$. Hence we can apply the induction hypothesis to show the claim.
- If $e = case\ e_0\ of\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\}$, then, by rule *Case*, $E_0 \vdash e_o ::^\alpha a_0$ and $E_i \vdash e_i ::^\alpha a$ $(i = 1, \ldots, j)$. Moreover, $x ::^\alpha b_0 \in E_0$, $x ::^\alpha b_1 \in E_1 \sqcup \ldots E_j$ and $b_1 \sqcap b_2 = C$. Thus, either b_1 or b_2 is equal to C. Hence we can apply the induction hypothesis to show the claim.
- Consider the final case $e = f\ e_1 \ldots e_n$. Rule *Fun2* cannot be applied (otherwise $x ::^\alpha C$ is not required). Hence rule *Fun1* is applicable so that $f ::^\alpha a_1, \ldots, a_n \to a \in F$ and $E_i \vdash e_i ::^\alpha a_i$ $(i = 1, \ldots, n)$. If $a_i = \top$ for all $i = 1, \ldots, n$, then $x ::^\alpha C$ is not required in contrast to our assumption. Hence there is at least one $i \in \{1, \ldots, n\}$ with $a_i \neq \top$ and $x ::^\alpha b_i \in E_i$. If $b_i = C$, the induction hypothesis implies $\bot \vdash \sigma(e_i) ::^\alpha a_i$ which implies the claim by rule *Fun1*. Otherwise, there are two different $i, j \in \{1, \ldots, n\}$ with $a_i \neq \top$, $a_j \neq \top$, $x ::^\alpha b_i \in E_i$, $x ::^\alpha b_j \in E_j$, $b_i \sqcap b_j = C$, and $b_i \neq C$ and $b_j \neq C$. This is impossible by the structure of our abstract domain.

□

Now we can prove our first result relating type information with the operational behavior of programs: if we can derive an impossible typing for some result type, we know that this expression cannot be evaluated to a value of this type. In the following, "$\to$" denotes the rewrite relation [Ter03] defined by the program rules and, as customary, "$\overset{*}{\to}$" its reflexive, transitive closure.

Lemma 4.8. Let P be a well-typed program w.r.t. a type environment F for P and $\bot \vdash e ::^\alpha a$. Then e does not evaluate to some $t \in [\![a]\!]$.

Proof. Assume that there is a derivation of e of the form

$$e = t_0 \ \to\ t_1 \ \to\ t_2 \ \to\ \cdots \ \to\ t_n$$

where n is the smallest index such that t_n is a constructor-rooted expression. We show by the induction on the length of all such derivations that the root of t_n is not compatible with a.

Base case ($n = 0$): e is already an expression rooted by some constructor C. Since $\bot \vdash e ::^\alpha a$, by rule *Con2*, C and a are not compatible so that $e \notin [\![a]\!]$.

Induction step: We assume $\bot \vdash e ::^\alpha a$ and the claim holds for all derivations of length $n-1$. Since e is not constructor-rooted, the following outermost evaluation steps are possible on e:

- If $e = e_1 \,?\, e_2$, then $t_1 = e_1$ (the other case is symmetric). Since $\bot \vdash e ::^\alpha a$, we have $\bot \vdash e_1 ::^\alpha a$ by rule *Or*. Hence we can apply the induction hypothesis to the derivation starting with t_1.
- If $e = \mathit{case}\ e_0\ \mathit{of}\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\}$, then, by rule *Case*, either $\bot \vdash e_o ::^\alpha a_0$ or $\bot \vdash e_i ::^\alpha a$ $(i = 1, \ldots, n)$. In the first case, e_0 is not evaluable to some constructor in a_0 (by the induction hypothesis). Hence, e_0 evaluates to some constructor C_i with $i > j$ (where j is chosen as in rule *Case*). Since $\bot \vdash e_i ::^\alpha a$ by rule *Case*, $\bot \vdash \sigma(e_i) ::^\alpha a$ (by Lemma 4.6) so that the instantiated case-branch cannot be evaluated to some value in $[\![a]\!]$ by the induction hypothesis. In the second case, e_0 might be evaluable to some constructor C_i with $i \leq j$, but then the case-branch is not evaluable to some value in $[\![a]\!]$ as before.
- If $e = f\ e_1 \ldots e_n$ and $f\ x_1 \ldots x_n = e' \in P$, then $t_1 = \sigma(e')$ with $\sigma = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. If $f ::^\alpha \bot \to a \in F$, then $\bot \vdash e' ::^\alpha a$ (by Def. 4.3) so that $\bot \vdash \sigma(e') ::^\alpha a$ (by Lemma 4.6) and $\sigma(e')$ is not evaluable to some value in $[\![a]\!]$ by the induction hypothesis.
  If $f ::^\alpha a_1, \ldots, a_n \to a \in F$, then, by rule *Fun1*, there is some $i \in \{1, \ldots, n\}$ with $\bot \vdash e_i ::^\alpha a_i$ and $a_i \neq \top$, i.e., $a_i = C$ for some constructor C. Since $\{x_1 ::^\alpha a_1, \ldots, x_n ::^\alpha a_n\} \vdash e' ::^\alpha a$ (by Def. 4.3), $\bot \vdash \sigma(e') ::^\alpha a$ (by Lemma 4.7, which can be easily extended to environments with more than one element). By the induction hypothesis, $\sigma(e')$ (and also e) is not evaluable to some value in $[\![a]\!]$.

□

The next lemma shows that well-typed expressions put the correct demands on required values for variables occurring in the expression. To simplify the statements, we use the notation $e \xrightarrow{*} [\![a]\!]$ for the fact the e evaluates to some t with $t \in [\![a]\!]$.

Lemma 4.9. Let P be a well-typed program w.r.t. a type environment F for P and $E \vdash e ::^\alpha a$ derivable with $E \neq \bot$. If $\sigma(e)$ evaluates to some value (constructor-rooted term) $t \in [\![a]\!]$ for some substitution σ, then, for all $x ::^\alpha \tau \in E$, $\sigma(x) \xrightarrow{*} [\![\tau]\!]$.

Proof. Assume that F is a well-typed type environment for program P, $E \vdash e ::^\alpha a$ derivable, $E \neq \bot$, σ is a substitution such that $\sigma(e)$ evaluates to some constructor-rooted term $t \in [\![a]\!]$, and x is some variable with $x ::^\alpha \tau \in E$. We show, by the induction on the length k of the derivation of $\sigma(e)$ to t, that $\sigma(x) \xrightarrow{*} [\![\tau]\!]$ with at most k steps.

- If e is a variable, then, by rule *Var*, $e = x$ and $a = \tau$. Since $\sigma(x) = \sigma(e) \xrightarrow{*} [\![a]\!]$, we have $\sigma(x) \xrightarrow{*} [\![\tau]\!]$ with the same number of steps.
- If $e = C\ e_1 \ldots e_n$, then, by rule *Con1* (rule *Con2* is not applicable since we assumed $E \neq \bot$), $E = \varnothing$ and the claim trivially holds (since $\tau = \top$).
- If $e = e_1 \,?\, e_2$, then, by rule *Or*, $E_i \vdash e_i ::^\alpha a$ $(i = 1, 2)$ and $E = E_1 \sqcup E_2$. Since $\sigma(e)$ evaluates to t, there is some $i \in \{1, 2\}$ such that $\sigma(e_i)$ evaluates to t with at most $k-1$ steps. Furthermore, $E_i \neq \bot$ (otherwise, $\sigma(e_i)$ does not evaluate to t by Lemma 4.6 and Lemma 4.8) so that $x ::^\alpha \tau_i \in E_i$. Therefore, by the induction hypothesis, $\sigma(x) \xrightarrow{*} [\![\tau_i]\!] \subseteq [\![\tau]\!]$ since $\tau_i \sqsubseteq \tau$ and Prop. 4.1.
- If $e = \mathit{case}\ e_0\ \mathit{of}\ \{C_1\ \overline{x_{k_1}} \to e_1; \ldots; C_n\ \overline{x_{k_n}} \to e_n\}$, by rule *Case*, $E = E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j)$, $E_0 \vdash e_0 ::^\alpha a_0$, $E_1 \vdash e_1 ::^\alpha a, \ldots$, $E_j \vdash e_j ::^\alpha a$, $\bot \vdash e_{j+1} ::^\alpha a, \ldots$, $\bot \vdash e_n ::^\alpha a$, and $a_0 = C_1 \sqcup \ldots \sqcup C_j$. Since $\sigma(e)$ evaluates to t, by the operational meaning of case expressions, $\sigma(e_0)$ evaluates to $C_i\ \overline{e_{k_i}}$ and $\sigma'(e_i)$ evaluates to t for $\sigma' = \{\overline{x_{k_i} \mapsto e_{k_i}}\}$. If $i > j$, then $t \notin [\![a]\!]$ by Lemma 4.8 (since $\bot \vdash e_i ::^\alpha a$). Due to this contradiction, we know that $i \leq j$.
  Since $x ::^\alpha \tau \in E$ and $E = E_0 \sqcap (E_1 \sqcup \ldots \sqcup E_j)$, we can distinguish the following cases:
  1. $x ::^\alpha \tau_0 \in E_0$: By the induction hypothesis (the derivation of $\sigma(e_0)$ is a shorter than the derivation of $\sigma(e)$), $\sigma(x) \xrightarrow{*} [\![\tau_0]\!]$. If $x ::^\alpha \top \in (E_1 \sqcup \ldots \sqcup E_j)$, $\tau_0 = \tau$ and the claim holds. Otherwise, $x ::^\alpha \tau_i \in E_i$ with $\tau_i \neq \top$. Since $E_i \vdash e_i ::^\alpha a$, by the induction hypothesis (the derivation of $\sigma(e_i)$ is a shorter than the derivation of $\sigma(e)$), $\sigma(x) \xrightarrow{*} [\![\tau_i]\!]$. Furthermore, $\tau_i \sqsubseteq \tau$ so that we have $\sigma(x) \xrightarrow{*} [\![\tau]\!]$ by Prop. 4.1.
  2. $x ::^\alpha \tau_0 \notin E_0$: Assume that $\tau \neq \top$ (otherwise, the claim trivially holds). Then $x ::^\alpha \tau_i \in E_i$ with $\tau_i \neq \top$ and we can proceed as in the previous case.
- If $e = f\ e_1 \ldots e_n$, by rule *Fun1*, $f ::^\alpha a_1, \ldots, a_n \to a \in F$, $E_i \vdash e_i ::^\alpha a_i$ $(i = 1, \ldots, n)$, and $E = \bigsqcap\{E_i \mid$

$a_i \neq \top\}$. By Def. 4.3, $f\ x_1 \ldots x_n = e' \in P$, $E' \vdash e' ::^\alpha a$, and $E' = \{x_i ::^\alpha a_i \mid 1 \leq i \leq n\}$. Let $\sigma' = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. Then $\sigma(\sigma'(e'))$ evaluates to t with at most $k-1$ steps. By the induction hypothesis, $\sigma(e_i) = \sigma(\sigma'(x_i)) \stackrel{*}{\to} [\![a_i]\!]$ with at most $k-1$ steps. Assume that $x ::^\alpha \tau_i \in E_i$, $a_i \neq \top$ and $\tau_i \neq \top$ (otherwise, $\tau = \top$ and the claim trivially holds). Since $\sigma(e_i) \stackrel{*}{\to} [\![a_i]\!]$ and $a_i \neq \top$, by the induction hypothesis, $\sigma(x) \stackrel{*}{\to} [\![\tau_i]\!] \subseteq [\![\tau]\!]$.

□

Now we come back to the proof of Theorem 4.4:

Proof. Let P be a program which is well typed w.r.t. a type environment F for P, and $f ::^\alpha a_1, \ldots, a_n \to a \in F$. In order to show that this typing is correct, we have to show by Def. 4.2 that, for all expressions $e_1, \ldots, e_n$, the following implication holds: if $f\ e_1 \ldots e_n$ evaluates to some value (constructor-rooted term) $t \in [\![a]\!]$, then, for $i = 1, \ldots, n$, e_i evaluates to some $t_i' \in [\![a_i]\!]$. Hence, assume that $f\ e_1 \ldots e_n$ evaluates to some value $t \in [\![a]\!]$. Consider the rule $f\ x_1 \ldots x_n = e \in P$ and the substitution $\sigma = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. Since P is well typed w.r.t. F, by Def. 4.3, $E \vdash e ::^\alpha a$ with $E = \{x_i ::^\alpha a_i \mid 1 \leq i \leq n\}$. Since the function call $f\ e_1 \ldots e_n$ evaluates to t, $\sigma(e)$ also evaluates to t. By Lemma 4.9, each $e_i = \sigma(x_i)$ evaluates to some $t_i \in [\![a_i]\!]$ since $x_i ::^\alpha a_i \in E$. □

Although our main interest for this development is the transformation of Boolean equalities in Curry programs, the results in this section have also other interesting applications, even for purely functional programs, i.e., program without occurrences of free variables and non-deterministic choices. For instance, we can use the inferred abstract types to find restrictions on the use of defined operations. As an example, consider an operation `min` to compute the minimum value of a list which is defined as follows:

```
min xs = smallerThan (head xs) xs

smallerThan m [] = m
smallerThan m (x:xs) = if x<m then smallerThan x xs
                              else smallerThan m xs

head (x:_) = x
```

With our inference rules, we can derive the following type:

$\texttt{min} ::^\alpha \texttt{(:)} \to \top$

From this type we can infer that any successful evaluation of a call to `min` requires that the argument must be a non-empty list. Hence, we can document this restriction by adding it as a precondition to `min`, e.g., by using the contract/specification language for Curry proposed in [AH12].

Another application of our inference system is the detection of bugs in programs. If there is an expression e in a program for which one can derive $\bot \vdash e ::^\alpha C$ for all constructors C of the concrete type of e, then, by Lemma 4.6 and 4.8, this expression can never be evaluated to a value. A similar result holds for function types which might be useful to find buggy operations:

Corollary 4.10. Let P be a well-typed program w.r.t. a type environment F for P, f a function defined in P, and $f ::^\alpha \bot \to C \in F$ for all constructors C of the concrete result type of f. Then no application of f can be evaluated to a value.

Proof. Consider the definition $f\ x_1 \ldots x_n = e \in P$ of f, the application $e' = f\ e_1 \ldots e_n$, and some constructor C of the concrete result type of f. If $f ::^\alpha \bot \to C \in F$, by Def. 4.3, $\bot \vdash e ::^\alpha C$. Let $\sigma = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$. By Lemma 4.6, $\bot \vdash \sigma(e) ::^\alpha C$ so that, by Lemma 4.8, $\sigma(e)$ is not evaluable to some value in $[\![C]\!]$. Hence, e' is not evaluable to some value in $[\![C]\!]$. Since this holds for any constructor C, the claim holds. □

For instance, consider the following operation:

```
buggy x = solve (x && not x)
```

The following types can easily be derived with our inference rules:

$\texttt{buggy} ::^\alpha \bot \to \texttt{True}$
$\texttt{buggy} ::^\alpha \bot \to \texttt{False}$

Therefore, by the previous corollary, an application of `buggy` can never yield a value.

5. Implementation

We have seen in various examples that there does not exist a meaningful most general type for each function. Therefore, classical type inference systems [DM82] are not directly applicable to infer the non-standard types used in this work. Instead, we use the idea to compute types by a fixpoint analysis [Cou97]. The analysis is started with no information about each function (e.g., $f ::^{\alpha} \top, \ldots, \top \to \top$) 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 performs a new iteration 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 $f ::^{\alpha} \top, \ldots, \top \to C_i$ (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. Since the signature of a program is finite, the possible typings of a program are finite. Hence only a finite number of iterations are possible and the analysis always terminates.

The actual implementation of the analysis uses the Curry analysis system CASS [HS14]. 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 analysis of required values is a prerequisite to implement the transformation of equalities as discussed in Sect. 3. The analysis results 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 [HAB+16] and KiCS2 [BHPR11].

In order to assess 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

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

has an infinite search space, whereas the transformed expression

```
solve (xs =:= ys && xs++ys =:= [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 time since non-deterministic search is transformed into deterministic bindings, as demonstrated by some benchmarks.

We used the Curry implementation KiCS2 [BHPR11] for the benchmarks. KiCS2 evaluates the Boolean equality operator by narrowing with the "`==`" rules shown in Sect. 2 and the equational constraints by managing variable bindings [BHPR13]. 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 [AH10]), `simplify` (simplify a symbolic arithmetic expression), and `varInExp` (non-deterministically return a variable occurring in a symbolic arithmetic expression). Figure 2 shows the execution times for evaluating

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

Program	#lines	=:= (original)	== (transformed)	full	fast
CHR	474	11	11	2.65	0.76
CurryStringClassifier	194	21	21	0.82	0.25
HTML	1316	13	13	6.04	2.14
Parser	49	6	6	0.22	0.02
SetFunctions	90	28	28	0.40	0.07
AddTypes	117	4	4	1.46	0.20
Curry2JS	633	6	6	2.85	0.85
maxtree	17	3	3	0.18	0.01
queens	12	5	2	0.19	0.01

Fig. 3. Benchmarks: transforming Boolean equations into equational constraints

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

As discussed in the introduction, an objective of this work is to simplify the usage of Curry by using only Boolean equations in source programs and automatically replacing them by equational constraints, if possible. In order to assess whether our method is sufficient for this purpose in practice, we applied it to various existing Curry programs which use equational constraints. We replaced all such constraints by Boolean equalities and checked how many of these Boolean equalities are replaced by equational constraints with our transformation tool. The results are summarized in Fig. 3. The first group of Curry programs are standard libraries distributed with KiCS2, where HTML is the largest one (supporting programming of dynamic web pages [Han01]). The next two programs (AddTypes, Curry2JS) are tools contained in the KiCS2 distribution to add type signatures to top-level operations and compiling Curry programs into JavaScript programs (which is used to implement type-safe dynamic web pages [Han07]), respectively. The last two programs are small examples demonstrating typical functional logic programming techniques. The column "#lines" shows the number of source code lines of each program, and the column headed by "=:=" shows the number of occurrences of this operator in the original program[7]. In the benchmarks, all these occurrences were replaced by "==" before applying our transformation to these programs.

The column headed by "==" reports the number of occurrences of this operator that were transformed back into the original "=:=" operator. The numbers in these columns show that our tool was able to transform almost all of them into constraints. The rare cases where this was not possible (queens) are operations that

[7] A logic programmer might wonder about the low number of equational constraints even in larger functional logic programs. This is mainly due to the fact that functional logic programming supports nested expressions (where Prolog programmers have to use auxiliary variables and unification to connect the result from an inner computation to an outer one). Moreover, predicates delivering multiple results can also be expressed as non-deterministic functions.

return constraints to be solved instead of using them in a condition of a program rule. For a simplified example of this kind, 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 = solve (x==y && y==z)
```

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

As sketched in the previous section, our transformation tool is integrated into the compilation chain of the recent releases of the Curry systems PAKCS and KiCS2. In order to control the application of the transformation tool during the compilation process, the configuration files of these systems allow the user to set the following usage modes:

off: In this mode, the transformation is not applied to the source program.

full: In this mode, the source program is analyzed as described in Sect. 4 and then the analysis results are used to perform the transformation described in Sect. 5.

fast: We introduced this mode after realizing that, in applications programs, the opportunities to transform Boolean equations into constraints are often due to the use of Boolean functions defined in the prelude, like "`&&`" (conjunction), "`||`" (disjunction), `not` (negation), and the conditional operator `cond`. Therefore, the "fast" mode does not perform the fixpoint analysis of Sect. 4 but simply uses the pre-computed abstract types for the most relevant Boolean functions defined in the prelude to perform the transformation described in Sect. 5. In all benchmarks shown in Fig. 3, there was no difference in the transformation results between the "full" and the "fast" mode. This indicates that the "fast" mode is sufficient to perform the desired transformations in practical programs. Therefore, it is the default mode of the tool.

The last two columns show the run times of our transformation tool in the "full" and "fast" mode (in seconds, where the same machine as for the benchmarks in the previous section has been used). These numbers clearly indicate the advantage of the "fast" mode. Hence, the "fast" mode is a reasonable compromise between effectiveness and efficiency.

In order to speed up the actual program transformation, only functions that contain occurrences of "`==`" are checked for a potential transformation. Thanks to the fast mode and this simple optimization, even large modules can be transformed without any perceivable slowdown in the compilation chain.

7. Related Work

Although our approach is the first one that analyzes required types in order to compute particular values, there are various related works in the area of analyzing declarative programs. For instance, [BEØ93] discussed an abstraction of term rewriting (which was extended in [BE95] to conditional term rewriting systems). The objective of this abstraction is the approximation of the top-level constructors of term evaluations in order to improve E-unification. For this purpose, they associate to a set of rewrite rules an abstract rewrite system that is able to compute finite approximations of top-level constructors. Since their framework is restricted to constructor-based, confluent and terminating rewrite systems without partial functions, it is too limited for functional logic programming. Moreover, they approximate only result values but not the conditions to compute such values.

In order to detect program errors, [ACE+02] also approximated the constructor values of successful evaluations by a fixpoint characterization of the input/output relation of functions defined by term rewriting systems. In contrast to our approach, they approximate only the input/output behavior of functional programs but not the necessary input requirements to compute some abstract value. This is also the case in [Han08] where call patterns occurring in functional logic computations are analyzed and approximated in order to optimize programs or checking safety properties of functional logic programs. Another method with similar goals has been presented in [MR07], where Haskell programs are checked for the absence of

pattern-match errors due to functions with incomplete patterns in their definitions. A static checker extracts constraints from pattern-based definitions and tries to solve them by simplification and fixpoint iteration in order to compute approximations of data constructors occurring in computations.

A quite different approach to make logic computations more efficient has been taken in the declarative programming language Mercury [SHC96]. Mercury supports mode annotations for predicates. The Mercury compiler exploits mode annotations to re-order predicate calls so that the general unification of two terms is reduced to either comparing the known values of these terms or assignments where one side contains (at run time) free variables that are assigned to the value of the other side. This leads to a highly efficient implementation but restricts the usage of logic computations. Although some of these restrictions can be relaxed by a more advanced mode analysis [OSS02], mode and determinism restrictions are still present in Mercury in contrast to Curry. Note that our analysis and transformation is independent of the call modes of the operations at run time.

8. 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 type `Success`, present in previous versions of Curry, can be omitted. This has the consequence that quite similar operations, like inequalities between values (`<=`), do not need to be duplicated for the type of Booleans and constraints, as it is currently the case.
2. It is not necessary to consider the subtle differences between the types `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 [Han97].
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 [AGL94, KLMR92], 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 "`=/=`".

Apart from transforming Boolean equalities in functional logic programs, the type analysis developed in this paper has also applications, briefly discussed in Sect. 4, even for purely functional programs. For instance, we can use abstract type information to detect buggy operations that cannot yield any value, or to infer restrictions on operations that can serve as preconditions for their correct application.

Acknowledgments. The authors are grateful to the anonymous reviewers and Sandra Dylus for their suggestions to improve a previous version of this paper.

References

[ACE+02] 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.

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[AH10] S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[AH12] S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.

[AH14] 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.

[AHH+05] 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.

[Ant01] 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.

[Ant10] S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522, May 2010.

[AGL94] 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.

[BE95] 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.

[BEØ93] D. Bert, R. Echahed, and M. Østvold. Abstract rewriting. In *Proc. Third International Workshop on Static Analysis*, pages 178–192. Springer LNCS 724, 1993.

[BHPR11] 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.

[BHPR13] 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.

[CC77] 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.

[Cou97] P. Cousot. Types as abstract interpretations. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 316–331, 1997.

[DM82] 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.

[HAB+16] 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/`, 2016.

[Han97] 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.

[Han01] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[Han07] M. Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.

[Han08] M. Hanus. Call pattern analysis for functional logic programs. In *Proceedings of the 10th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'08)*, pages 67–78. ACM Press, 2008.

[Han12] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[Han13] 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.

[Han16] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.

[HS14] 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.

[KLMR92] 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.

[MR07] N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6, pages 15–30. Intellect, 2007.

[Myc80] 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.

[OSS02] D. Overton, Z. Somogyi, and P.J. Stuckey. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 109–120. ACM Press, 2002.

[PJ03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[Red85] U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

[Rey72] J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740. ACM Press, 1972.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[Sla74] J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[Ter03] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[War82] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, pages 441–454, 1982.