A Proposal for Disequality Constraints in Curry

Emilio Jesús Gallego Arias[a,1,2] Julio Mariño Carballo[a,1,3]
José María Rey Poza[b,4]

[a] *Babel Research Group*
Universidad Politécnica de Madrid
Madrid, Spain
https://babel.ls.fi.upm.es

[b] *Telefónica I+D*

Abstract

We describe the introduction of disequality constraints over algebraic data terms in the functional logic language Curry, and their implementation in Sloth, our Curry compiler. This addition extends the standard definition of Curry in several ways. On one hand, we provide a disequality counterpart to the constraint equality operator (`=:=`). Secondly, boolean equality operators are also redefined to cope with constructive disequality information, which leads to a more symmetric design w.r.t. the existing one. Semantically speaking, our implementation is very similar to previous proposals, although there are some novel aspects. One of them is that the implementation is partly based on an existing finite domain (FD) constraint solver, which provides a more efficient execution in some examples and, more important, the first complete implementation of disequality constraints over finite types. A detailed description of the finite type case is provided, including: i) the use of the FD solver; ii) an algorithm for analysing cardinality of types, and iii) how to deal with cardinality information at run time. Some benchmarks, an operational semantics minimally extending the one in the Curry draft, and a moderately detailed description of the implementation complete the paper.

Keywords: Curry, Sloth, Disequality, Constraints, Implementation.

1 Introduction

This paper describes an extension that allows programming with disequality constraints over data terms in Sloth. Proposals of this kind [5,1] can be dated back to the *pre-Curry ages* (one of them is for the Babel language) and are already included in other FLP languages ($\mathcal{TOY}$ [7]). However, although the possibility of extending Curry with disequality constraints is explicitly mentioned in the Curry draft, they have not been included so far, perhaps because equality is too close to the core

[1] Authors partly supported by Spanish grants MCYT TIC 2002-0055 and CAM S-0505/TIC/0407

[2] Email: egallego@babel.ls.fi.upm.es

[3] Email: jmarino@fi.upm.es

[4] Email: jmrey@tid.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of the language (unification). Also, equality operators are closely related to some aspects of Curry that are likely to suffer changes in the near future — type classes, standard classes analogous to `Eq`, etc.

For these reasons, having experimental versions of this kind of features is a good starting point, possibly at the cost of inconsistencies with other features already present in the draft. Although not completely polished, the proposal described in this paper is already available in our Sloth system (`https://babel.ls.fi.upm.es/research/Sloth`) [5] including the examples presented below.

Sloth [9,2] is a compiler that translates Curry [3] programs into Prolog, continuing our group's previous experience that started with the translation of Babel programs into Prolog. Currently, Sloth generates Ciao Prolog [4] code and the system implements most of the 0.8 version of the Curry draft.

The main motivation for developing and maintaining a not-so-efficient implementation of Curry is the need of keeping up to date with a rapidly evolving language, easily introducing changes that would take longer in an abstract machine based implementation. Nevertheless, while slower than Prolog, Sloth is perfectly usable as a first contact with Curry.

These changes can be additions to the Curry standard or, perhaps, experimental features that need some testing preceding the mandatory discussion needed in order to include them in later versions of the standard.

Integration with the Ciao platform has a number of advantages considering its advanced features like: the realistic set of libraries – including, among other, constraint programming, concurrency, foreign language interface; the static analysis framework; meta-programming constructs, etc.

Improving the support of constraint programming in Curry is one of the main goals of Sloth, as many of the outstanding features of Ciao are constraint libraries themselves or libraries which provide support for programming new constraint extensions — although often in a nondeclarative manner. In fact, experimental support for constraint programming over rationals and finite domains is already available in the latest stable version of Sloth.

An example

The following example, taken from [5] – although adapted to our language – should serve to motivate the behavior expected from our extension:

Example 1.1 The *size* of a list is defined as the number of distinct elements it contains. [6]

```
module Size where

member x []     = False
member x (h:ts) = x == h || member x ts

data Nat = Zero | Succ Nat

size :: [a] → Int
size []     = 0
size (h:ts) = if   member h ts
              then size ts
```

[5] Use the "Development versions" link when in there.

[6] The `==` operator used here is a flexible one with disequality support.

```
                else 1+(size ts)
```

Consider a goal

```
let x, y, z free in size [(x, Zero), (y, z)]
```

The size of that list can be 1 or 2, depending on the actual values for x, y and z. The answers returned by our system are: [7]

```
1 :: Int
{z =:= Zero , x =:= y}
Try more (y/n)? y
2 :: Int
{x =/= y}
Try more (y/n)? y
2 :: Int
{z =:= Succ _}
Try more (y/n)? y
no solutions
```

Overview of the paper

Next section presents the current state of (equality) constraints in Curry and introduces our proposal for a new set of operators, allowing disequality constraints. An operational semantics dealing with the new operators is described in Sec. 3. It is intended to be as similar as possible to the one present in the Curry Report. Implementation details are not trivial, mainly due to interactions between the constraint operators and the type system and the choice to offload part of the constraint solving to a finite domains constraint solver. All of them are described in depth in section 4. Some benchmarks are shown in Sec. 5 and Sec. 6 discusses the results, shortcomings and point directions for future progress.

2 A proposal for equality and disequality operators

The core of the constraint system present in Curry is the `=:=` operator, which stands for constrained equality. This operator is overloaded so it can deal with `Integer` data types and the likes. Resolving such overloading problems is a different topic itself, so from now onwards we will restrict ourselves to allow only algebraic data types in Curry expressions.

Would not be natural to provide a counterpart constraint operator `=/=` so we can express disequality constraints? We think so, but it is not a trivial task. In our opinion, the main motivation for the absence of the counterpart disequality operator `=/=` from the Curry standard is due to two main factors:

- Semantics of the constrained equality are much simpler that the ones arising from disequality. For instance – in the equality case – the substitution happens to be unique, while when dealing with disequalities, most of the cases will not be representable by means of a single substitution, needing an associated constraint store.
- Correctly implementing such and operator will for sure augment the complexity of the Curry implementations to a great extent, which may not be appropriate

[7] the output has been slightly beautified to ease the reading.

for an experimental language.

Although not present in Curry, there have been two different proposals for the implementation of disequality constraints in functional logic programming, overcoming the previous difficulties. The first one we are aware of is a proposal for the Babel [10] programming language [5] and the second one adds disequality constraints to $\mathcal{TOY}$ [1]. Both proposals are – very roughly speaking – based on constraint accumulation using a global store and checking its satisfiability at variable binding. The Münster Curry Compiler (MCC) [8] implements partial disequality support inspired by the latter proposal, and uses the `=/=` operator.

The absence of disequality constraints in standard Curry has influenced the current choice of equality operators. As the report itself says in page 11:

> *"However, the evaluation of `[x]=:=[0]` does not deliver a `Boolean` value `True` or `False`, since the latter value would require a binding of `x` to all values different from 0 (which could be expressed if a richer constraint system than substitutions, e.g., disequality constraints, is used)..."*

So far, the solution adopted has been to limit the LP features of equality operators, i.e. we have to chose between losing the disequality information or having it only for ground instances of queries — which may lead to incompleteness.

The current set of equality operators is shown in table 1. Roughly speaking, they can be classified in two groups: operators returning `Bool` and operators returning `Success`. The operators returning `Bool` use residuation as their operational model while operators returning `Success` use narrowing.

Flexible operators (i.e. those using narrowing) are very interesting when a search is needed and to construct non-deterministic functions. On the other hand, residuation-based operators (the ones returning `Bool`) are the right choice when a deterministic function or predicate is required.

Further, operators returning `Bool` can be used to find *positive* as well as *negative* answers (i.e. answers making the goal `True` and `False` respectively) while operators returning `Success` find *positive* answers only, pruning the others (this is because `Success` is a one-point domain). The operator to choose depends on what the programmer wants to obtain from the program.

However, when dealing when disequalities, *negative* answers are often required as a result out from searches. In this case, one would like to have flexible operators returning `Bool`. There are none of these in current versions of Curry. The following example illustrates the problem:

Example 2.1 We want to search for the elements of a list:

```
elem :: a → [a] → Bool
elem _ []     = False
elem x (y:ys) = x == y || elem x ys

> let x free in elem x [1,2,3] =:= True
> Suspended
```

If `==` had a flexible implementation we would get only one result, because of the absence of negative answers:

```
> let x free in elem x [1,2,3] =:= True
> success    {x ↦ 1}
```

op. name	*type*	*flexible?*
`==`	`a → a → Bool`	no
`/=`	`a → a → Bool`	no
`=:=`	`a → a → Success`	yes

Table 1
Existing equality operators in Curry.

op. name	*type*	*flexible?*	*notes*
`==`	`a → a → Bool`	yes	
`/=`	`a → a → Bool`	yes	defined as `not.(==)`
`=:=`	`a → a → Success`	yes	no changes
`=/=`	`a → a → Success`	yes	the new operator
`===`	`a → a → Bool`	no	old rigid equality
`/==`	`a → a → Bool`	no	the negation of previous one

Table 2
Our proposal for equality operators in Curry.

```
> Try more (y/n)? y
> no solutions
```

The intended behavior would be more on the line of:

```
> let x free in elem x [1,2,3] =:= True
> success    {x ↦ 1}
> Try more (y/n)? y
> success    {x ↦ 2}
> Try more (y/n)? y
> success    {x ↦ 3}
> no solutions
```

In order to overcome this and trying to reach a more orthogonal operator set we are proposing a new set that can be seen in table 2. With our proposal we have:

- Two rigid operators on `Bool`. These operators already exist in Curry although we have renamed them.
- The flexible version of the two previous operators. Those operators are new.
- Finally, the operators returning `Success`. The disequality operator is new in Curry and is the key element in this work.

Last, but not least, is the fact that Curry semantics relates `Success` with total satisfiability. As can be seen in section 4.2, this requirement and the existence of finite data types makes the implementation overly complex, as terms belonging to finite types have to be handled separately from terms whose type has an infinite number of instances. To the extent of our knowledge, this is the first paper with an in-depth description of such a problem.

Computation step for a single expression:

$$\frac{\mathsf{Eval}[\![e_i]\!] \Rightarrow D}{\mathsf{Eval}[\![e_1 \& e_2]\!] \Rightarrow \mathit{replace}(e_1 \& e_2, i, D)} \quad i \in \{1,2\}$$

$$\frac{\mathsf{Eval}[\![e_i]\!] \Rightarrow D}{\mathsf{Eval}[\![c(e_1,\ldots,e_n)]\!] \Rightarrow \mathit{replace}(c(e_1,\ldots,e_n), i, D)} \quad i \in \{1,\ldots,n\}$$

$$\frac{\mathsf{Eval}[\![f(e_1,\ldots,e_n)]\!]T \Rightarrow D}{\mathsf{Eval}[\![f(e_1,\ldots,e_n)]\!] \Rightarrow D} \quad \text{if } T \text{ is a definitional tree for } f \text{ with fresh variables}$$

Computation step for an operation-rooted expression e:

$$\frac{}{\mathsf{Eval}[\![e]\!]\mathit{rule}(l\text{=}r) \Rightarrow \{\epsilon; \mathit{id}[\![\sigma(r)\}} \quad \text{if } \sigma \text{ is a substitution with } \sigma(l) = e$$

$$\frac{\mathsf{Eval}[\![e]\!]T_1 \Rightarrow D_1 \quad \mathsf{Eval}[\![e]\!]T_2 \Rightarrow D_2}{\mathsf{Eval}[\![e]\!]\mathit{or}(T_1,T_2) \Rightarrow D_1 \cup D_2}$$

$$\mathsf{Eval}[\![e]\!]\mathit{branch}(\pi,p,r,T_1,\ldots,T_k) \Rightarrow$$

$$\begin{cases} D & \text{if } e|_p = c(e_1,\ldots,e_n),\ \mathit{pat}(T_i)|_p = c(x_1,\ldots,x_n) \text{ and } \mathsf{Eval}[\![e]\!]T_i \Rightarrow D \\ \emptyset & \text{if } e|_p = c(\ldots) \text{ and } \mathit{pat}(T_i) \neq c(\ldots),\ i = 1,\ldots,k \\ \bigcup_{i=1}^{k}\{\epsilon; \sigma_i[\![\sigma_i(e)\} & \text{if } e|_p = x,\ r = \mathit{flex}, \text{ and } \sigma_i = \{x \mapsto \mathit{pat}(T_i)|_p\} \\ \mathit{replace}(e,p,D) & \text{if } e|_p = f(e_1,\ldots,e_n) \text{ and } \mathsf{Eval}[\![e|_p]\!] \Rightarrow D \end{cases}$$

Derivation step for a disjunctive expression:

$$\frac{\mathsf{Eval}[\![e]\!] \Rightarrow \{\gamma_1;\sigma_1[\![e_1,\ldots,\gamma_n;\sigma_n[\![e_n\}}{\{\gamma;\sigma[\![e\}\cup D \Rightarrow \mathit{clean}(\{\gamma_1 \wedge \sigma_1(\gamma);\sigma_1\circ\sigma[\![e_1,\ldots,\gamma_n \wedge \sigma_n(\gamma);\sigma_n\circ\sigma[\![\,e_n\}) \cup D}$$

Fig. 1: Operational semantics of Curry

3 Operational semantics

In this section we present an operational semantics that allows computing with disequality constraints. The presentation tries to be a minimal extension to that present in the Curry draft. This way, changes can be easily located. Essentially, there are two changes. First, the mechanism for accumulating answers is enhanced to include constraints in solved form in addition to answer substitutions. This extension is generic, i.e. largely independent of the constraint system in use. Secondly, specific rules for simplifying disequality constraints are included.

The main *execution cycle* is described by the rules in Fig. 1, that introduces the derivability relation $D_1 \Rightarrow D_2$ on pairs of *disjunctive expressions*.

Disjunctive expressions represent (fragments of) the fringe of a search tree. Formally, they are multisets of *answer expressions* of the form $\gamma;\sigma[\![e$, where γ is a constraint,[8] σ a substitution and e a Curry expression. An answer expression $\gamma;\sigma[\![e$ is *solved* when e is a data term and γ is solved and consistent. We will use ϵ to denote a trivial constraint.

The computation of an expression e *suspends* if there is no D such that $\mathsf{Eval}[\![e]\!] \Rightarrow D$. A constraint expression is *solvable* if it can be reduced to `success`. As can be seen in Fig. 1, reduction of terms rooted by user-defined function symbols is guided by an overloaded version of $\mathsf{Eval}[\![]\!]$ that takes a Curry expression and a *definitional tree* as arguments. For details on these, and other aspects of the semantics which are largely orthogonal to the questions discussed here – conditional rules, higher-order

[8] Here, the word *constraint* refers to *formal constraints* i.e. internal representations of constraint formulae, opposed to Curry constraint expressions.

$$\frac{\mathsf{Eval}[\![e_i]\!] \Rightarrow D}{\mathsf{Eval}[\![e_1\texttt{=:=}e_2]\!] \Rightarrow \mathit{replace}(e_1\texttt{=:=}e_2, i, D)} \quad \text{if } e_i = f(t_1, \ldots, t_n),\ i \in \{1, 2\}$$

$$\frac{}{\mathsf{Eval}[\![c(e_1, \ldots, e_n)\texttt{=:=}c(e'_1, \ldots, e'_n)]\!] \Rightarrow \{\epsilon; id[\![e_1\texttt{=:=}e'_1\texttt{\&}\ldots\texttt{\&}e_n\texttt{=:=}e'_n\}}$$

$$\frac{}{\mathsf{Eval}[\![c(e_1, \ldots, e_n)\texttt{=:=}d(e'_1, \ldots, e'_m)]\!] \Rightarrow \emptyset} \quad \text{if } c \neq d \text{ or } n \neq m$$

$$\frac{\mathsf{Eval}[\![x\texttt{=:=}e]\!] \Rightarrow D}{\mathsf{Eval}[\![e\texttt{=:=}x]\!] \Rightarrow D} \quad \text{if } e \text{ is not a variable}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=:=}y]\!] \Rightarrow \{\epsilon; \{x \mapsto y\}[\![\texttt{success}\}}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=:=}c(e_1, \ldots, e_n)]\!] \Rightarrow \{\epsilon; \sigma[\![y_1\texttt{=:=}\sigma(e_1)\texttt{\&}\ldots\texttt{\&}y_n\texttt{=:=}\sigma(e_n)\}} \quad \begin{array}{l}\text{if } x \notin cv(e_1, \ldots, e_n),\\ \sigma = \{x \mapsto c(y_1, \ldots, y_n)\},\\ y_1, \ldots, y_n \text{ fresh variables}\end{array}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=:=}c(e_1, \ldots, e_n)]\!] \Rightarrow \emptyset} \quad \text{if } x \in cv(c(e_1, \ldots, e_n))$$

Fig. 2: Solving equational constraints

features, freezing, etc – the reader is referred to [3].

The *disjunctive* behavior of disjunctive expressions is partly captured by the last rule in Fig. 1, which expresses how answers are accumulated. Observe that the combination of answers – both substitutions and new constraints – with the accumulated constraint might introduce inconsistency or perhaps constraints not in solved form. This is why a call to the auxiliary function *clean* is needed. Its definition depends on the actual constraint system and will be presented later for the disequality case.

The other half of this disjunctive behavior is captured by the auxiliary function *replace*, that inserts a disjunctive expression into a position in a term, giving another disjunctive expression as result:

$$\mathit{replace}(e, p, \{\gamma_1; \sigma_1[\![e_1, \ldots, \gamma_n; \sigma_n[\![e_n\}) = \{\gamma_1; \sigma_1[\![\sigma_1(e)[e_1]_p, \ldots, \gamma_n; \sigma_n[\![\sigma_n(e)[e_n]_p\}$$

Figure 2 shows the rules for solving equational constraints. These are practically identical to those in the Curry draft and are included here mainly to reveal the symmetries and dualities w.r.t. the rules for solving disequality constraints, shown in Fig. 3. Observe that the auxiliary function cv, such that $cv(e)$ collects the variables in e not inside a function call is used to implement an *occurs check*.

Although the theory of disequality constraints is well established, the actual choice of solved forms may vary with regard to implementation considerations. Following [5], we have chosen to avoid explicit disjunctive constraints and instead we carry those alternatives over the search tree of the Curry semantics, i.e. disjunctive constraints are distributed over disjunctive expressions.

This way, solved disequality constraints – those appearing in the left hand of answer expressions – amount to conjunctions of disequations between distinct variables, in the case where those variables range over infinite sets of values. When the variables can only take a finite set of values, solved forms are extended with the

$$\frac{\mathsf{Eval}[\![e_i]\!] \Rightarrow D}{\mathsf{Eval}[\![e_1\texttt{=/=}e_2]\!] \Rightarrow \mathit{replace}(e_1\texttt{=/=}e_2, i, D)} \quad \text{if } e_i = f(t_1, \ldots, t_n),\ i \in \{1, 2\}$$

$$\frac{}{\mathsf{Eval}[\![c(e_1, \ldots, e_n)\texttt{=/=}c(e'_1, \ldots, e'_n)]\!] \Rightarrow \{\epsilon; id[\![e_1\texttt{=/=}e'_1, \ldots, \epsilon; id[\![e_n\texttt{=/=}e'_n\}}$$

$$\frac{}{\mathsf{Eval}[\![c(e_1, \ldots, e_n)\texttt{=/=}d(e'_1, \ldots, e'_m)]\!] \Rightarrow \texttt{success}} \quad \text{if } c \neq d \text{ or } n \neq m$$

$$\frac{\mathsf{Eval}[\![x\texttt{=/=}e]\!] \Rightarrow D}{\mathsf{Eval}[\![e\texttt{=/=}x]\!] \Rightarrow D} \quad \text{if } e \text{ is not a variable}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=/=}y]\!] \Rightarrow \{x \neq y; id[\![\texttt{success}\}} \quad \text{if } \mathit{range}(x) \text{ is infinite}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=/=}y]\!] \Rightarrow \{x \neq y \wedge x \in \mathit{range}(x) \wedge y \in \mathit{range}(y); id[\![\texttt{success}\}} \quad \text{if } \mathit{range}(x) \text{ is finite}$$

$$\frac{}{\mathsf{Eval}[\![x\texttt{=/=}c_j(e_1, \ldots, e_n)]\!] \Rightarrow \{\epsilon; \sigma_1[\![\texttt{success}, \ldots, \epsilon; \sigma_j[\![\sigma_j(x)\texttt{=/=}c_j(e_1, \ldots, e_n), \ldots, \epsilon; \sigma_k[\![\texttt{success}\}}$$

if $x \notin \mathit{cv}(e_1, \ldots, e_n)$, $\sigma_i = \{x \mapsto c_i(y_{i1}, \ldots, y_{i\ ar(c_i)})\}$, y_{uv} fresh variables

$$\frac{}{\mathsf{Eval}[\![x\texttt{=/=}c(e_1, \ldots, e_n)]\!] \Rightarrow \texttt{success}} \quad \text{if } x \in \mathit{cv}(c(e_1, \ldots, e_n))$$

Fig. 3: Solving disequality constraints

corresponding *constraints for domain consistency*.

As we have said before, accumulating answers may corrupt the constraint store either by making it inconsistent or not solved. The task of tidying everything up – perhaps moving part of the constraint information back to the expression store – is the responsibility of function *clean*:

$$\begin{array}{lll}
\mathit{clean}(\emptyset) & = \emptyset & \\
\mathit{clean}(\{\gamma; \sigma[\![e\} \cup D) & = \mathit{clean}(D) & \text{if } \gamma \text{ inconsistent} \\
\mathit{clean}(\{e_1 \neq e_2 \wedge \gamma; \sigma[\![e\} \cup D) & = \mathit{clean}(\{\gamma; \sigma[\![e_1\texttt{=/=}e_2\texttt{\&>}e\} \cup D) & \text{if } e_1 \text{ or } e_2 \text{ nonvars} \\
\mathit{clean}(\{\gamma; \sigma[\![e\} \cup D) & = \{\gamma; \sigma[\![e\} \cup \mathit{clean}(D) & \text{otherwise}
\end{array}$$

4 Implementation details

We will consider two cases:

Implementation for infinite types.
In types with an infinite number of instances handling disequality is easier as we can guarantee that a disequality between an instance – likely partial – of such a type and a new free variable is always satisfiable. Support for constraints among variables of infinite types is already present in $\mathcal{TOY}$ and in the Münster Curry Compiler.

Extending our implementation to correctly handle finite types.
In finite types, testing consistency of constraints is harder, as there are disequality chains where one runs out of possible instances for variables.

The implementation has been done using Ciao Prolog and its attributed variables library. Regarding Sloth, it is a new library which plugs into the current module system supplying the `=/=` operator.

4.1 Implementation for infinite types

The basic technique used is to attach to each disequality constrained variable an attribute, which contains the set of disallowed instantiations for that variable:

```
DiseqAttribute = '$de_store'(Var, List))
```

where `List` containts all the terms that `Var` should be different from.

The system can just assume that each constrained variable must have its correspoding attribute, so the implementation should hook into the compiler in two ways:

- The disequality operator itself.
- Unification of constrained variables, both with terms or with other constrained variables.

which nicely maps to the semantics of attributed variables.

Attaching attributes

Disequality constraints are added when the execution path tries to reduce an expression whose head is the `=/=` operator to HNF. The implementation of this operator is fully native – we mean fully written in Prolog – using the standard hooks present in Sloth for external libraries.

The first action to be performed by the operator is to evaluate its arguments to HNF, then select the applicable case: both arguments are variables, only one is a variable or neither are.

In addition, we will use a predicate for adding constraints to the variables' store:

```
add_to_store(Var, L) :-
        (   get_attribute(Var, '$de_store'(Var, List)) →
            union_ro(L, List, LNew),
            update_attribute(Var, '$de_store'(Var, LNew))
        ;
            attach_attribute(Var, '$de_store'(Var, L))
        ).
```

Then, given the structure of our store, the case when both arguments are variables becomes trivial:

```
diseq_vars(A, B) :-
        add_to_store(A, [B]),
        add_to_store(B, [A]).
```

as is the one for a variable and an instantiated term:

```
%% Term is in HNF.
diseq_one_var(Var, Term) :-
        add_to_store(Var, [Term]).
```

The final case is when both arguments are not variables, so we need to check their parameters, if they have any:

```
diseq_spine(Term1, Term2) :-
        Term1 =.. [C1|L1],
```

```
        Term2 =.. [C2|L2],
        (   \+ C1 = C2 →
            true
        ;
            C1 = C2,
            diseq_or(L1, L2)
        ).
diseq_or([A|AL],[B|BL]) :-
        (   diseq(A, B)
        ;
            diseq_or(AL, BL)
        ).
```

In the code above we profit from our representation of Curry data constructors as Prolog terms.

Unification hooks

Unification of constrained variables has two different cases:

- The variable is being unified with a term instantiated to at least HNF form.
- The variable is being unified with another constrained variable.

Ciao Prolog provides the multifile predicates `verify_attribute/2` for the first item and `combine_attributes/2` for the second.

Unification with a term Unification with a term just checks that the set of accumulated disequality constraints in our constraint store holds, and then proceeds to unify the var with the term:

```
verify_attribute('$de_store'(Var, List), Term) :-
        detach_attribute(Var),
        Var = Term,
        diseq_and(Term, List).
```

The instantiation of `Var` will also instantiate all copies of the variable present in other constraint stores. This non-trivial detail is possible thanks to Ciao Prolog implementation of attributed variables, which allows us to store the *real* variables in the attributes.

`diseq_and` will just verify – by calling the main `diseq` predicate – that all the elements in the list are different from the term to unify with.

This has a very important effect, as it will create the new needed disequality constraints in the case `Term` would be partially instantiated, or the constraint store contained partially instantiated constraints.

Unification between variables When dealing with disequality between two already constrained variables, our new constraint store will be the union of their respective constraint stores, if there doesn't exist a previous disequality between the unifying variables:

```
combine_attributes('$de_store'(V1, L1), '$de_store'(V2, L2)) :-
        \+ member_ro(V1, L2),      % doesn't instantiate vars
        union_ro(L1, L2, NewL),
        detach_attribute(V1), detach_attribute(V2),
        V1 = V2,
        attach_attribute(V1, '$de_store'(V1, NewL).
```

It should be noted that like in the previous case, the `union/3` predicate will unify the non-instantiated terms present in the constraint stores. This is precisely what we are looking for, as any constraint attached to such terms will be combined.

The behavior of backtracking is solved as Ciao Prolog fully supports backtracking attributed variables, so it is not an issue, indeed it greatly helps our implementation, as when unification of a constrained variable needs to be undone, all the operations will be rolled back, including reattaching the previous attributes.

4.2 Implementation for finite types

As mentioned in the operational semantics, when the terms to be constrained do belong to a finite data type, the number of available instantiations for a variable is bound. This way, when dealing with the disequality case we must be proactive checking the constraint store consistency.

The following example

```
> let a,b,c free in a=/=b & b=/=c & c=/=a & a=:=True
```

would give a wrong answer under the previous implementation, given that it is assumed we have an infinite number of instantiations for variables, but in this case is not true, as the variables `a`, `b`, `c` can be only be instantiated to two different values, thus making the above constraint unsatisfiable.

At the implementation level, our view is to handle this situation as a $\mathcal{FD}$ problem, getting an important leverage from mature $\mathcal{FD}$ solvers. So our constraint store is extended with a $\mathcal{FD}$ variable:

```
DiseqAttribute = '$de_store'(Var, Fd, List))
```

being `Fd` the $\mathcal{FD}$ variable associated to `Var`.

For this scheme to work, we assume the existence of the following two predicates for obtaining type meta-information:

```
type(Term, Type, Range)
```

which returns the type and the range of any arbitrary translated Curry term, including variables.

```
index(Term, IndexList)
```

which returns the index i of `Term`, $i \subseteq range(type(Term))$ as a list. How this metainformation is obtained will be discussed in section 4.5.

The modified constraint solver

We will detail here only the parts of the solver affected by this extension, starting with the store handling operation:

```
add_to_store(Var, Fd, L) :-
        (   get_attribute(Var, '$de_store'(Var, FdOld, List)) →
            union_ro(List, L, LNew),
            Fd = FdOld,
            consistent(Fd), % Needed only in some FD solvers
            update_attribute(Var, '$de_store'(Var, Fd, LNew))
        ;
            attach_attribute(Var, '$de_store'(Var, Fd, L))
        ).
```

For the two variables case, we have to correctly constrain their associated $\mathcal{FD}$ variables:

```
diseq_vars(A, B) :-
        (   type(A, flat, Range) →
```

```
            get_fd_var(A, FdA), % Will return a fresh var if A has no FD var
            get_fd_var(B, FdB),
            FdA in 1..Range,
            FdB in 1..Range,
            FdA .<>. FdB
        ;
            true                % A is non-flat
        ),
        add_to_store(A, FdA, [B]),
        add_to_store(B, FdB, [A]).
```

Constraining a variable to be different from a term is achieved in this case by constraining its associated $\mathcal{FD}$ var:

```
%% Term is in HNF.
diseq_one_var(Var, Term) :-
         (    type(Term, flat, _) →
              get_fd_var(A, FdA),
              index(Term, TIndex),
              constraint_fd_list(FdA, TIndex)
         ;
              true
         ),
        add_to_store(Var, FdA, Term).
```

where `constraint_fd_list(Fd, List)` forces `Fd` to be different from all the elements in `List`.

When both arguments are instantiated, we don't need to change the behavior of the solver.The unification is sightly modified to care about the new $\mathcal{FD}$ variables in the store:

Unification with a term The new case introduced by the flat terms is taken care of by the `remap_term/2` predicate, which checks that the term is in the variable domain and maps any residual constraint into the possible subterms.

```
verify_attribute('$de_store'(Var, Fd, List), Term) :-
         (   type(Term, flat, Range) →
             remap_term(Term, Fd)
         ;
             diseq_and(Term, List),
         ),
         detach_attribute(Var),
         Var = Term.
```

Unification between variables The new added case is very simple, we only have to constraint our $\mathcal{FD}$ variables to be equal.

```
combine_attributes('$de_store'(V1, Fd1, L1), '$de_store'(V2, Fd2, L2)) :-
         (   type(Term, flat, Range) →
             Fd1 .=. Fd2
         ;
             \+ contains_ro(V1, L2),      % doesn't instantiate vars
             union(L1, L2, NewL)
         ),
         update_attribute(V1, '$de_store'(V1, NewL).
```

4.3 The $\mathcal{FD}$ solver

As the reader can see, the use of the $\mathcal{FD}$ solving library is fairly limited, as we only use the disequality (`.<>.`) predicate. This means that maybe using a full-featured $\mathcal{FD}$ solver is overkill for this application, but our hope is to profit from advanced arc-consistency algorithm found in this kind of constraint libraries so a considerable speedup can happen.

4.4 A combined approach example

To illustrate some of the gains from this mixed approach, we will showcase an example using both strategies. Let's use the query:

```
> let x free in [x] =/= [True] & [x] =/= [False]
```

Then our execution trace will look like:

```
> (x:[]) =/= (True:[]) & ... { spine case }
> success & (x:[]) =/= (False:[]) { x =/= True }
> (x:[]) =/= (False:[]) { x =/= True ∧ x =/= False }
> fail { inconsistent FD store }
```

The execution first uses the normal method for non-flat types – in this case the list type – but when it finds a flat variable can use that information to always return correct answers.

4.5 Type meta-data handling

An obvious problem of this approach is that it needs knowledge about term and variable types at runtime. The first step is to perform static analysis on the types so we can determine which types are flat and what others not.

Definition 4.1 [Type Graph] A type graph G for a type T is informally defined as the directed graph with types as nodes and edges from $T1$ to $T2$ when $T1$ contains $T2$. Type variables have the special type Pol.

Definition 4.2 [Finite types] A type T is *finite* when its associated graph G has no cycles and no leaf node is in Pol. Then we define *range* of T as $\Pi_{t \in N} \, |TC(t)|$.

Definition 4.3 [Polymorphic types] A type T is *polymorphic* when its associated graph G has a leaf with in Pol and has no cycles. We call the set of all leaves in Pol $PLevel(T)$.

Definition 4.4 [Infinite types] A type T is *infinite* when its associated graph G has one or more cycles.

Once this analysis is performed, we attach to each term e a finiteness annotation which will be:

- *fixed*(r) if *type*(e) is finite, where $r = range(type(v))$.
- *poly*(P) if *type*(e) is polymorphic, where $P = PLevel(T)$.
- *infinite* if *type*(e) is infinite.

With this information attached, we can easily determine the information needed for the predicate `type/3`, which is emitted at compile time.

The last step is to obtain the information given by the `index/2` predicate. The naive approach currently used is to instantiate all possible terms and assign to each one a numerical index. However, it should be noted that more sensible approaches do exist. As we are talking about typical algebraic data types, when finite, don't have a big number of elements.

Meta-data implementation in Sloth

There are different approaches to try when implementing this kind of runtime typing. In Sloth, we have opted for the simplest one, that is to convert every term in a tuple (t, a), where t was the original term and a is its finiteness annotation.

This approach allows to propagate finiteness annotations at the same time terms are unified, so it comes handy avoiding function specialization for flat types.

We are undertaking a complete redesign of this area, obviously to lower the overhead caused for the double unification that we currently use. Given the fact that terms instantiated to some degree already carry their type information [9], the only difficult case would be the variables' one, and using attributed variables to "tag" free variables with their type seems sensible, but then every function that can restrict a parameter with a more general type. Let us illustrate this problem:

```
f :: Bool → Bool
f a = a
```

Then `f` has to tag its argument, as would the incoming argument come with a polymorphic type, the operational semantics has no way to realize about that. This is a major drawback we hope to fix researching about type inference of `let x free` expressions.

4.6 Additional considerations

Fortunately, our compiler's modularity has allowed us to use the power of the module system so *all* the disequality related functionality has been encapsulated into a module.

We only had to modify the compiler in two places:

- Adding the static analyzer presented in section 4.5, so the runtime has knowledge about types.
- The shell now has special code for visualizing disequality constraints.

5 Experimental results

We present some preliminary experimental results just as a sample of what our system can do. We also have written more tests for the disequality implementation, which can be found in the compiler distribution.

The chosen problem is the coloring of a square map, where each cell has 4 neighbors and a cell cannot have the same color as any of them. The corresponding Curry program can be seen in figure 4. The disequality operator to use has been abstracted as an argument, so we can use the same code for all the tests.

The first test has been performed using the old `==` operator and narrowing, which implies that we have to instantiate the map first, the second one allowing residuation whereas the third one uses the new disequality constraint operator.

Our motivation for benchmarking is not proving big performance enhancements, but to check that there is no major slowdown going on with the new disequality

[9] This is a property of our name handling in the compiler

```
data Color = Red | Green | Yellow | Blue

diff x y   = (x==y) =:= False
diff_c x y = x =/= y

color Red    = success
color Yellow = success
color Green  = success
color Blue   = success

foldr_c l = foldr (&) success l

coloring :: (Color → Color → Success) → [[Color]] →  Success
coloring _ []     = success
coloring f (l:ls) = c_lists f l ls & map_c_list f (l:ls)

c_lists :: (Color → Color → Success) → [Color] → [[Color]] → Success
c_lists _ _ []     = success
c_lists f l (x:xs) = foldr_c (map (uncurry f) (zip x l)) & c_lists f x xs

map_c_list :: (Color → Color → Success) → [[Color]] → Success
map_c_list f l = foldr_c (map (c_list f) l)

c_list:: (Color → Color → Success) → [Color] → Success
c_list f [x]         = success
c_list f (x1:x2:xxs) = f x1 x2 & c_list f (x2:xxs)

p_naive map = constraint map & coloring diff map
p_reseq map = coloring diff map & constraint map
p_diseq map = coloring diff_c map & constraint map

constraint l = foldr_c (map (\ll → foldr_c (map color ll)) l)
```

Fig. 4: The map coloring problem.

System	Problem	$time_{normal}$	$time_{res}$	$time_{diseq}$
Sloth	Coloring	$>$ 100 sec.	20 ms.	24 ms.
$\mathcal{TOY}$	Coloring	$>$ 100 sec.	n.a.	20 ms.

Table 3
Benchmark results for the coloring problem with a 5x5 map

system. This seems to be true and indeed the disequality library used here lacks fine tuning and we are also planning to use a much more improved $\mathcal{FD}$ solver.

Anyways, comparing a system with disequality to a equality-only one is not easy, as the main advantage that disequality supports brings us is a far more expressive language. The results for a random map which has a proportion of 90% free variables among its elements are shown in table 3. As a matter of reference, we present the results for the same algorithm [10] using $\mathcal{TOY}$ [7], although direct comparison of the two systems is meaningless, as they use different Prolog compilers, etc.

6 Conclusions and future work

We have extended Curry's operational semantics to allow disequality constraints in a similar spirit to the proposal in [5]. A first implementation in the Sloth system that treats, for the first time, the case of finite types, has been described. This implementation is already available from our download site. So far, we believe that implementation results are promising, and we hope that practice with our prototype

[10] The code used for $\mathcal{TOY}$ can be found in the compiler tarball.

can help in improving the support for constraints in the Curry standard.

We should note that disequality has been studied as a narrowing helper [6]. In this paper, while we show an operational semantics for disequality, we have not studied its impact on the performance of narrowing, as this paper is mainly aimed at bringing more expressiveness to the language.

Due to preliminary character of our implementation, we plan to address in the near future some of the remaining corners, namely interaction with the $\mathcal{FD}$ solver, better run-time typing for terms and variables and better implementations for some predicates, such as `index/2`.

Another issue to consider in the near future, among others, is the interaction with the type classes extension. It would be desirable to restrict the types for which equality/disequality operators can be applied, analogously to the `Eq` standard type class in Haskell. Such an extension would allow to transparently mix advanced constraint solvers, allowing the overloading of the equality disequality operators.

References

[1] Puri Arenas-Sánchez, Ana Gil-Luezas, and Francisco Javier López-Fraguas. Combining lazy narrowing with disequality constraints. In *PLILP*, pages 385–399, 1994.

[2] Emilio Jesús Gallego Arias and Julio Mariño. An overview of the Sloth2005 Curry System: System Description. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 66–69, New York, NY, USA, 2005. ACM Press.

[3] Michael Hanus, Sergio Antoy, Herbert Kuchen, Francisco J. López-Fraguas, Wolfgang Lux, Juan José Moreno-Navarro, and Frank Steiner. *Curry: An Integrated Functional Logic Language*, 0.8.2 edition, April 2006. Editor: Michael Hanus.

[4] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, december 1995. Available from `http://www.clip.dia.fi.upm.es/`.

[5] Herbert Kuchen, Francisco Javier López-Fraguas, Juan José Moreno-Navarro, and Mario Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *JICSLP*, pages 207–221, 1992.

[6] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. Disequalities may help to narrow. In Maria Chiara Meo and Manuel Vilares Ferro, editors, *APPIA-GULP-PRODE*, pages 89–104, 1999.

[7] Francisco Javier López-Fraguas and Jaime Sánchez-Hernández. *TOY*: A multiparadigm declarative system. In Paliath Narendran and Michaël Rusinowitch, editors, *RTA*, volume 1631 of *Lecture Notes in Computer Science*, pages 244–247. Springer, 1999.

[8] Wolfgang Lux. *Münster Curry User's Guide*, 0.9.10 edition.

[9] Julio Mariño and José María Rey. The implementation of Curry via its translation into Prolog. In Kuchen, editor, *7th Workshop on Functional and Logic Programming (WFLP98)*, number 63 in Working Papers. Westfälische Wilhelms-Universität Münster, 1998.

[10] Juan José Moreno-Navarro and Mario Rodríguez-Artalejo. Babel: A functional and logic programming language based on constructor discipline and narrowing. In Jan Grabowski, Pierre Lescanne, and Wolfgang Wechler, editors, *ALP*, volume 343 of *Lecture Notes in Computer Science*, pages 223–232. Springer, 1988.