A Generic Semantics for Constraint Functional Logic Programming

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

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

[b] *Telefónica I+D*

Abstract

We propose an operational semantics for lazy constraint functional programs that is *generic* in the sense of allowing the integration of different constraint solvers into an existing kernel language in a rather *clean* way. The design of the semantics has been driven by two principles often contradicting each other: the need to support laziness in the constraint side, and the need to adhere to the *black box* principle of constraint programming as much as possible. We start by taking a previous semantics for Curry with equality and disequality constraints over Herbrand terms and showing how to decouple constraint handling from a *neutral* kernel semantics. Then, we define a *Generic Constraint Solver Interface* that allows us to add new solvers, thus showing the applicability of the method. The paper contains also discussions on the role played by the type system and on how to incorporate constraint solver cooperation in this setting.

Keywords: constraint programming, functional-logic programming, Curry, Sloth

1 Introduction

Functional logic programming (FLP) and, particularly, the language Curry [5], proposes a standardization of multiparadigm declarative programming where syntax, type discipline and (most of the) declarative semantics are borrowed from Haskell and deduction methods such as *needed narrowing*, *residuation* or *encapsulated search* provide the operational semantics necessary to cover most of the functionality of existing logic programming languages.

[1] Authors supported by CAM/ESF grant 3060/2006 and MEC grant TIN2006-15660-C02-02 (DESAFIOS)

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

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

[4] Email: jmrey@tid.es

One of the most powerful and interesting features that FLP can adopt from logic programming is constraint programming. However, this integration is not as seamless as desired, due in part to the lazy semantics of Curry. Shortly, lazy evaluation imposes a demand driven strategy to the evaluation of expressions that is somehow aware of the internal structure of expressions being evaluated, while constraint programming as incorporated in CLP [7], follows a black-box approach. This conflict has precluded the treatment of constraint terms as first class citizens in the execution model of FLP.

Thus, the treatment of constraints in FLP has either dropped the black box requirement or is essentially ad-hoc. Some examples:

- Domain-specific semantics for Curry plus one constraint domain – e.g. [10,4] – are hard to extend and difficult to implement/verify.
- The use of CRWL to specify a CFL programming language as a whole (TOY [8]) is an interesting formal exercise, but the specifications tend to be quite lengthy and, of course, this gives up the black box principle.
- On the practical side, there are a number of implementations (PAKCS, Sloth) where constraint domains are added interfacing existing solvers with a FL host system. However, this is usually done in an ad-hoc manner and it is hard to justify the correctness of the resulting implementation.

Our position is that the idea of interfacing Curry with external (perhaps existing) solvers is a very practical one that should not be given up, [5] but that it must be done in a uniform way, so that existing semantics do not have to suffer modifications in order to accomodate a new constraint domain.

This is mainly motivated by our own experience implementing Curry (Sloth [3]). Currently, we have support for several constraint extensions in Sloth, taking advantage of the solvers available in the underlying Ciao Prolog system. Linear and finite domain constraints use, essentially, the existing solver as is, but we do not have a formal semantics for these extensions. On the other hand, our implementation of disequality constraints [4] has a domain-specific semantics. Incorporating the other solvers in that semantics would have made it unmanageable. However, a close inspection of the semantics showed that most of the semantics was unaware of constraints and that interaction occured at certain specific points. Isolating those points was the key for a modular semantics.

Our proposal rests on previous work [11] where a trivalued framework for constraint domains was introduced. This means that the black-box principle, as defined in [7] is modified, but not abandoned: only one extra basic operation is assumed for constraint domains. The reader is referred to that paper for a discussion on why this extension is needed in order to have solvers aware of lazy evaluation.

The paper is organized as follows. Next section reviews some basic definitions on constraints and the syntax of our language. In Section 3 we introduce a presentation of an operational semantics for Curry where the Herbrand ($\mathcal{H}$) operators are completely decoupled from a constraint-neutral core. Then, in section 4 we introduce a generic constraint solver interface and show how to integrate new con-

[5] The Curry report advocates this, but does not say how to build such an interface.

straint solvers in Curry using this approach. Section 5 discusses how to – partially – achieve solver cooperation within our framework and what influences it has from other cooperation frameworks. Section 6 points some places where the language or its type system needs to be modified to support this framework. Section 7 concludes and points issues for future research.

2 Preliminaries

A number of presentations exist for constraint and constraint logic programming. The following pretends to be as general as possible, and is liberally based on Jaffar and Maher's survey [7]. Let $\Sigma = (PS, FS)$ be a signature composed of a set PS of *predicate symbols* and a set FS of *function symbols*. A *primitive constraint* has the form $p(t_1, \ldots, t_n)$, where $p \in PS$ and the t_i's are (open) terms made from functors in FS and variables. A *constraint* is a (first-order) formula built from primitive constraints. Generally, only a proper subset $\mathcal{L}$ of these are considered admissible constraints for a given domain. A Σ-*structure* $\mathcal{D}$ consists of a domain D and an interpretation for every predicate and function symbol in Σ. The pair $(\mathcal{D}, \mathcal{L})$ is called the *constraint domain*.

The language $\mathcal{L}$ contains constraints which are, respectively, identical to true and false in $\mathcal{D}$, and is usually closed under variable renaming, conjunction and existential quantification. In order to be computationally useful, every constraint domain is expected to support, at least, the following operations and tests on constraints:

(i) Test for *consistency* or *satisfiability*, $\mathcal{D} \vDash \tilde{\exists} c$, where $\tilde{\exists} c$ denotes the existential closure of c.

(ii) The *implication* or *entailment* of one constraint by another: $\mathcal{D} \vDash c \rightarrow c'$, or more generally, the implication of a disjunction of constraints by another: $\mathcal{D} \vDash c \rightarrow c_1 \vee \cdots \vee c_n$.

(iii) The *projection* of a constraint c onto variables $\bar{x}$ in order to obtain an equivalent, hopefully simpler, c' such that $\mathcal{D} \vDash c' \leftrightarrow \exists_{-\bar{x}} c$, where $\exists_{-\bar{x}} c$ denotes the existential closure of c on all its variables except those in $\bar{x}$.

(iv) Detecting that, given a constraint c, there is a single value for x consistent with c, i.e. $\mathcal{D} \vDash \exists z. \forall x, \bar{y}. c(x, \bar{y}) \rightarrow x = z$. We say x is *grounded* by c.

Accommodating these definitions in a language with *types*, *lazy semantics* and *higher order features* requires a number of changes which should be kept to a minimum in order to preserve the good properties of the traditional framework. The most obvious is that now the domain D must be an information domain, e.g. a *cpo*, so that *undefined* or *partially defined* values can be dealt with appropriately. Elements of FS will be interpreted as (well typed) continuous functions. Elements of PS will be interpreted as typed functions to a domain with a *bottom* element and two *total* values, e.g. $\{\bot, \texttt{Success}, \texttt{Failure}\}$. Notice that this forces the introduction of three-valued interpretations for constraints.

Having a small, standard set of operations like those listed above is what allows CLP to be such a flexible framework. For our lazy scheme we will just require an additional test on the constraint domain:

5. Test for *strictness*: a constraint c is strict in variable x iff replacing $\perp$ for x in c gives an indefinite constraint. In other words, $\mathcal{D} \models \tilde{\exists} c[\perp/x] = \perp$.

Remember that existential quantification must be interpreted in a three valued setting – i.e. as a least upper bound – etc. We are not working out the details of this extension here.

3 Decoupling Curry from the Herbrand domain

Given the considerations above, our first step will be to take Curry to a state with no built in constraint operators — not even equality of data terms. This semantics will be refined stepwise in order to obtain a definition generic enough to accommodate other constraint systems.

3.1 The core Curry semantics

This semantics is intended to resemble the one in the Curry report.[6] However, we have split the standard $\Rightarrow$ relation into two new relations $\Downarrow, \downarrow$ (*both* big step) relating terms with its normal and head normal forms respectively.[7] This allows us to have a standard HNF evaluation mechanism to refer to when integrating lazy evaluation of constraints into the semantics. Also, matching against a definitional tree has been renamed from $\mathsf{Eval}[\![]\!]$, to an auxiliary relation labelled as $\mathsf{Match}[\![,]\!]$.

The core semantics is shown in Figures 1 (reduction to HNF) and 2 (reduction to NF).

Answers and constraints

The standard operational semantics in the Curry report defines an element of an answer set as a tuple $\{\sigma \,\|\, e\}$, namely a term in normal form and an associated substitution. Following [4] we consider both an answer substitution and a constraint

$\{c; \sigma \,\|\, e\}$

but now c will be a constraint store possibly made of heterogeneous constraints (i.e. constraints from different domains). Moreover, c is intended to be treated as an abstract data type (defined in 3.2), i.e. it will only be accessed from the semantic rules by means of a restricted set of operators. This is an immediate *space saver*, as adding new domains does not make the core bigger.

Auxiliary functions

The replacement of a position in a term by a disjunctive expression (resulting in another disjunctive expression) is carried out by the auxiliary function *replace*:

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

Function *clean* removes inconsistencies from the constraint stores. These inconsistencies can arise after any reduction step. Function *clean* can be defined as follows:

[6] In fact, it is an evolution of the semantics in our paper on disequality constraints [4].

[7] NF and HNF henceforth

Computation step for expressions:

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

$$\frac{}{\mathsf{Eval}[\![c(e_1,\dots,e_n)]\!] \downarrow \{\epsilon; \mathit{id} \,\|\, c(e_1,\dots,e_n)\}} \quad c \in DC$$

$$\frac{\mathsf{Match}[\![f(e_1,\dots,e_n), T]\!] \downarrow D}{\mathsf{Eval}[\![f(e_1,\dots,e_n)]\!] \downarrow D} \quad T \in DT(f)$$

Computation step for operation-rooted expression e:

$$\frac{dt(e,T) = e' \quad \mathsf{Eval}[\![e']\!] \downarrow D}{\mathsf{Match}[\![e,T]\!] \downarrow D}$$

$$dt(e, \mathit{rule}(l{=}r)) = \{\epsilon; \sigma \,\|\, \sigma(r)\} \quad \text{if } \sigma(l) = e$$

$$dt(e, \mathit{branch}(\pi, p, T_1, \dots, T_k)) =$$

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

Computation step for answers

$$\frac{\mathsf{Eval}[\![e]\!] \downarrow \{c_1; \sigma_1 \,\|\, e_1 \dots c_n; \sigma_n \,\|\, e_n\} \quad \mathsf{Eval}[\![D]\!] \downarrow D'}{\mathsf{Eval}[\![\{c; \sigma \,\|\, e\} \cup D]\!] \downarrow \mathit{clean}(\{\sigma_1 c \wedge \sigma c_1; \sigma \cdot \sigma_1 \,\|\, e_1 \dots \sigma_n c \wedge \sigma c_n; \sigma \cdot \sigma_n \,\|\, e_n\} \cup D')}$$

Fig. 1: Head normal form Curry operational rules

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

$$\frac{\mathsf{Eval}[\![e]\!] \downarrow D \quad \mathsf{Eval}[\![D]\!] \Downarrow c}{\mathsf{Eval}[\![e]\!] \Downarrow c} \quad \text{if } e \neq c(t_1,\dots,t_n),\ c \notin DC$$

Fig. 2: Normal form Curry operational rules

- $\mathit{clean}(\emptyset) = \emptyset$
- $\mathit{clean}(\{c; \sigma \,\|\, e\} \cup D) = \mathit{clean}(D)$ if c is inconsistent.
- $\mathit{clean}(\{c; \sigma \,\|\, e\} \cup D) = \{c; \sigma \,\|\, e\} \cup \mathit{clean}(D)$ otherwise.

3.2 Constraints and constraint stores

Following [4], we assume a syntactic category CP of *constraint predicate* symbols, so that *atomic constraints* take the form of Curry terms $p(e_1^{\tau_1}, \dots, e_n^{\tau_n})$, with $p \in CP$, and type $p : \tau_1 \to \cdots \to \tau_n \to \mathit{Success}$. In order to deal with different constraint domains, CP can be seen as a disjoint sum of several CP_i. For the sake of simplicity, in the following we will assume that the CP_i are pairwise disjoint. As usual, constraints (type *Constraint*) are first order formulae containing the atomic constraints and closed (at least) under conjunction and existential quantification.

As we previously said, we assume an opaque type which represent constraint stores CS, and we define the Extended Constraint Stores (ECS) type as the disjoint sum of:

$$ECS = CS + eval_{Expr,Pos} + fail$$

As every constraint store CS for domain constraint D[8] is dealt as a black box, we require the solvers to provide the following operations:

- $\epsilon_D : CS$ (empty constraint store).
- $tell_D : (CS \times Constraint) \to Set(ECS)$: This function is meant to tell a constraint. It returns the list of modified constraints stores if successful or:
  - $eval_{e,p}$, if the term e at position p needs to be further evaluated in order to post the constraint. Note that the position refers to the position the term was in the constraint predicate, so the term can be evaluated, substituted in the CP and we can call it again.
  - $fail$, if the constraint cannot be satisfied.

Notice that $tell_D$ return type is not a new constraint store, but a *set* of constraints. This is needed in order to allow solvers to return a set of disjoint constraint stores. Keep in mind that this allow solvers – such as the Herbrand disequality one – to return a set of answers, some of them can be successful and other could demand more evaluation:

$$tell_{\mathcal{H}}\ \epsilon\ \texttt{c(a,b)=/=c(x,f(m))} = [Success, eval_{f(m),2}]$$

with `x` a variable, then a choice is `a=/=x` or to evaluate more `f(m)`$(eval_{f(m),2})$, which as we'll see, they get transformed in two paths of computation.

As we have said above, we are requiring constraint solvers to be able to detect those constrained positions where more evaluation is needed. This idea is borrowed from the trivalued framework of [11], where a strictness test for constraints is introduced. This test consists in computing the existential closure of a constraint on all its free variables but one which is replaced by $\perp$. In order to implement this test, the solvers must interpret any symbol not in their signature as if it were $\perp$. This allows us to fed the solvers any Curry term, regardless of how normalized it is. We'll denote any term not in the solver's signature as $\boxed{t}$, meaning that the solver cannot "break-in" and see what's inside.

In practice, implementing this reflective feature in a real solver is not very hard, as most practical domains (arithmetic, finite domains, etc) are essentially strict, i.e. unknowns in the input produce unknowns in the output. More specifically:

- Solvers *must* interpret any symbol not in their signature as $\perp$.
- Solvers must be able to accept *all* their parameters as $\perp$.

[8] D will be substituted for a concrete constraint domain when we are speaking of a concrete one

$$\frac{tell\ \epsilon\ e = \{c_1, \ldots, c_n\}}{\mathsf{Eval}[\![e]\!] \downarrow \{c_1; id[\![Success, \ldots, c_n; id[\![Success\}} \quad e = p(e_1, \ldots, e_n)$$

$$\frac{tell\ \epsilon\ e = fail}{\mathsf{Eval}[\![e]\!] \downarrow \emptyset} \quad e = p(e_1, \ldots, e_n)$$

$$\frac{tell\ \epsilon\ e = eval_{s,p} \quad \mathsf{Eval}[\![s]\!] \downarrow D \quad \mathsf{Eval}[\![replace(e, p, D)]\!] \downarrow D'}{\mathsf{Eval}[\![e]\!] \downarrow D'} \quad e = p(e_1, \ldots, e_n)$$

Fig. 3: Constraint predicates operational rules

- Solvers must return both the arguments position and the unrecognized expression as $eval_{e,p}$. The solver should only request *one* argument to be evaluated.

Most of the above requirements are done in the name of efficiency. For instance, all the above requirements could be substituted for probing with $\perp$ each one of the constraint predicate arguments, and we'd get the same result.

3.3 Selecting the right domain

So far, we have defined the extended constraint stores and their operations, but for this scheme to support multiple solvers, we need to define $c \in [ECS]$, the sequence of ECS stores *indexed* by constraint domain.

This way, we get one element in the sequence for each solver in the system. To refer an specific constraint store, we'll use the standard sequence notation $c[D]$, so for example, to address the constraint store belonging to $\mathcal{H}$ we'll use $c[\mathcal{H}]$.

So then we can introduce the function $tell : (CS \times Constraint) \rightarrow [ECS]$, which selects the right $tell_D$ function:

$$tell\ c\ e = \begin{cases} eval_{e,p} & \text{if } tell_D\ c[D]\ e = eval_{e,p} \\ fail & \text{if } tell_D\ c[D]\ e = fail \\ c[D] := c_D & \text{if } tell_D\ c[D]\ e = c_D \end{cases} \quad \text{where } e = op_D(e_1, \ldots, e_n)$$

This means that the set of constraint predicates should be disjoint. We can overcome this problem as shown in section 6.

3.4 Operational rules for constraint solver integration

The integration of constraint solving in this framework is achieved by means of new operational rules (figure 3) dealing with constraint predicates.

The two first rules respectively deal with the case of a satisfiable constraint or a failing one. The interesting one is the third, which is needed when the constraint predicate needs its arguments to be more evaluated than in their current form.

The main novelty about the above rules is the possibility to use *lazy-aware* constraint solvers, such as the ones presented in section 3.5 and in section 4.2.

Rules for equality constraints

$$\frac{\{x \neq y\} \notin c}{put(c, \mathsf{x{=}{:}{=}y}) = c \cup \{x = y\}} \quad \text{if } \mathsf{x}, \mathsf{y} \text{ variables.}$$

$$\frac{\{x \neq t\} \notin c}{put(c, \mathsf{x{=}{:}{=}t}) = c \cup \{x = t\}} \quad \text{if } \mathsf{x} \text{ variable.}$$

$$\frac{\mathsf{d}_1 \neq \mathsf{d}_2}{put(c, \mathsf{d}_1(\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}){=}{:}{=}\mathsf{d}_2(\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{m})) = fail} \quad \text{if } \mathsf{d}_1, \mathsf{d}_2 \in DC$$

$$\frac{\mathsf{d}_1 = \mathsf{d}_2 \quad c_1 = put(c, \mathsf{t}_1{=}{:}{=}\mathsf{u}_1) \ldots c_n = put(c, \mathsf{t}_\mathsf{n}{=}{:}{=}\mathsf{u}_\mathsf{n})}{put(c, \mathsf{d}_1(\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}){=}{:}{=}\mathsf{d}_2(\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{n})) = hclean(\cup_{i=0..n} c_i)} \quad \text{if } \mathsf{d}_1, \mathsf{d}_2 \in DC$$

Fig. 4: Herbrand solver rules I

Rules for disequality constraints

$$\frac{\{x = y\} \notin c}{put(c, \mathsf{x{=}/{=}y}) = c \cup \{x \neq y\}} \quad \text{if } \mathsf{x}, \mathsf{y} \text{ variables.}$$

$$\frac{\{x = t\} \notin c}{put(c, \mathsf{x{=}/{=}t}) = c \cup \{x{=}/{=}t\}} \quad \text{if } \mathsf{x} \text{ variable.}$$

$$\frac{\mathsf{d}_1 \neq \mathsf{d}_2}{put(c, \mathsf{d}_1(\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}){=}/{=}\mathsf{d}_2(\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{m})) = c} \quad \text{if } \mathsf{d}_1, \mathsf{d}_2 \in DC$$

$$\frac{\mathsf{d}_1 = \mathsf{d}_2 \quad c_1 = put(c, \mathsf{t}_1{=}/{=}\mathsf{u}_1) \ldots c_n = put(c, \mathsf{t}_\mathsf{n}{=}/{=}\mathsf{u}_\mathsf{n})}{put(c, \mathsf{d}_1(\mathsf{t}_1, \ldots, \mathsf{t}_\mathsf{n}){=}/{=}\mathsf{d}_2(\mathsf{u}_1, \ldots, \mathsf{u}_\mathsf{n})) = \{c_1, \ldots, c_n\}} \quad \text{if } \mathsf{d}_1, \mathsf{d}_2 \in DC$$

Fig. 5: Herbrand solver rules II

3.5 A Herbrand domain solver

Once settled our core semantics, we will use a constraint solver over $\mathcal{H}$, with constraint predicates $=, \neq$ to illustrate the usefulness of our semantics as well as to help Curry to regain its previously built in constraint capabilities.

The idea of representing $\mathcal{H}$ as constraint system is already present in [9], although this approach needs the full specification of the constraint system in a white-box fashion.

We could use whatever solver for $\mathcal{H}$ we feel like, but for completeness of the paper we'll specify one – in a style rather different from Curry semantics – and show an example of the full system in action.

Let us consider the Herbrand solver state to be composed of a set of equality and disequality constraints $c = \{c_1, \ldots, c_n\}$.

Then, the predicate *put* – which is just a name for $tell_{\mathcal{H}}$ – takes as arguments a constraint store c and the constraint to evaluate.

Some auxiliary functions are needed, namely:

- *hclean cs*: Returns failure if the constraint store *cs* is inconsistent.

The reader should have noticed that this specification is not completely equivalent to the built in constraint solving in Curry, as Curry implements the optimization of applying a substitution as a side effect of the `=:=` constraint operator.

New rules for laziness

$$\frac{x = \boxed{t}}{put(c, \mathsf{x\ op\ y}) = eval_{x,1}}$$

$$\frac{y = \boxed{t}}{put(c, \mathsf{x\ op\ y}) = eval_{y,2}}$$

Fig. 6: $\boxed{t}$ handling rules

Also, the second `=:=` rule in figure 4 doesn't do occurs check. This can be easily added using *hclean*, but it would mean complete evaluation of `t`.

For the sake of simplicity we haven't yet included the support in the semantics for solvers returning substitutions, however it should be easy, as we should just modify the *tell* type to return a substitution. Then the rules dealing with *tell*, should just apply the solver's proposed substitution to the answer expression.

This constraint solver is also missing the analysis for finite types, as presented in [4]. This is a benefit of these semantics, that we can just enhance the constraint solving side without having to care too much about Curry itself.

Another good effect is that the constraint solver is way clearer than the one which has to take Curry into account.

3.6 An example

As an example of the new hybrid system, consider the program:

```
[] ++ x      = x
(x:xs) ++ y = x : (xs ++ y)

>x ++ y =/= y
```

Then the program develops as:

(i) $\mathsf{Eval}[\![\mathsf{x{+}{+}y{=}/{=}y}]\!] \Rightarrow$ (as x++y is not in solver's signature, it's read as $\boxed{t}$).

(ii) $tell(\epsilon, \boxed{\mathsf{x{+}{+}y}} \neq y) = eval_{\mathsf{x{+}{+}y},1} \Rightarrow$ (tell is invoked as =/= is in CP)

(iii) $\mathsf{Eval}[\![\mathsf{x{+}{+}y}]\!] = \{\epsilon; [x \mapsto []][\![\mathsf{y}]\!]\} \cup \{\epsilon; [x \mapsto (x_1 : x_2)][\![\mathsf{x_1} : (\mathsf{x_2{+}{+}y})]\!]\} \Rightarrow^*$

(iv) First case:
   (a) $tell(\epsilon, y \neq y) = fail \Rightarrow \emptyset$

(v) Second case:
   (a) $put(\epsilon, x_1 : (x_2{+}{+}y) \neq y) = \top \Rightarrow$
   (b) $\{\{y \neq x_1 : (x_2{+}{+}y)\}; [x \mapsto (x_1 : x_2)][\![Success\}$

The most important point is the step between point 3 and 4, where *clean* gets applied and both the inconsistent elements of the disjunctive expression get eliminated and the substitutions get applied to the constraints store.

4 General constraint solvers integration

We have shown that the above scheme is suitable for splitting $\mathcal{H}$ to its own constraint solver. What about applying it to general constraint solvers?

Below we list some example solvers and how to integrate them in this scheme.

4.1 Arithmetic solvers

All arithmetic solvers ($\mathcal{FD}$, CLPR) are completely strict in their arguments, so we only need to modify $tell_{\mathcal{FD}}$ – for instance – to return *eval* in every case their argument is not in HNF.

We already have implemented support for CLPR and $\mathcal{FD}$ in Sloth, using the underlying solvers available in Ciao.

As the solvers are completely strict, but they don't directly support tri-valued semantics, we had to write a wrapper for each solver[9]:

```
tell_fd(Constraint, Res) :-
    Constraint ..= [Op, Arg1, Arg2],
    (
        bot(Arg1) →
        Res = bot(1, Arg1)
    ;
        bot(Arg2) →
        Res = bot(2, Arg2)
    ;
        call(Constraint) →
        Res = Success
    ;
        fail
    ).
```

Note that the Prolog solvers carry around the constraint store *c* as a global variable.

This doesn't preclude the obvious optimization of reducing the arguments to HNF previously to any call to the solvers.

4.2 String equality solver

A better example is the constraint solver for string equality, in the style of Prolog III [1].

Such a constraint solver carries a signature of text strings plus string concatenation (•) and the equal $=_S$ constraint operator.

This way, the Curry program:

```
f = "a"
m = loop

g = f • m =S "ba"
```

can be solved, whereas in a non lazy language the program would loop.

This is due to the fact that previously, telling the constraint would force both `f` and `m` to be evaluated to NF, but in this case, we call the solver with both arguments

[9] actually, it is the same as both are strict.

evaluated, and the solver requests the evaluation of `f`, and this is enough to solve the constraint.

5 Bridges between solvers in Curry

Cooperation among constraint solvers has been a topic which has been treated on several previous works (see, for example [6]). These works are based, at least partially, on sharing variables among solvers (that is, a given variable is known by more than one solver).

This is perfectly possible on some logic languages where a type system does not exist. It is possible, for example, to write constraints such as:

```
X>2.5, X in [2,2.5,3,7,9.8]
```

In the example above it is clear that the type for `X` is numeric but it is not as clear whether it is Int or Float since the first expression seems to hint that `X` has type `Float` while the second one states that `X` belongs to an enumerated type (usually, an integer). Additionally, the list of allowed values for the variable contains both integer and floating point values.

As said before, this can happen in untyped languages, however, this is not the case when we switch to typed functional logic languages. In such functional-logic languages, every term *has* a specific type, so variables definitively belong to either `Int` of `Float`, but no to both.

To remedy this, a new operator called *bridge* has been proposed in [2]. Bridges are a new kind of constraints adopting the shape `x#=y`, where `x` and `y` are different free variables of different types.

A bridge like `x#=y` states that from that point on, variables `x` and `y` will always have *equivalent* values from their respective domains. Think, for example of `x` as an integer variable and `y` as a floating point variable. Then, the values 1 for `x` and 1.0 for `y` are equivalent. However, we should note that domain equivalence is very specific to the solvers. For more details about bridges, we suggest to read the related bibliography.

A example of bridge use in Curry would be:

```
x > 2.5 & x #= fx & fx in [2,2.5,3,7,9.8] where x, fx ↩
    free
```

5.1 How to incorporate bridges

Although we haven't fully developed a semantics for solver cooperation in this framework, our first analysis shows that there is no need to modify the presented semantics at all! This result is surprising, but it comes as a result of encapsulation of the constraint programming side in the semantics.

This way, solver cooperation can be achieved within this semantics just by defining `#=` as a new constraint predicate with its associated constraint store, where bridges are stored in a similar way to other proposals.

Before going on, we should note that due to the typing discipline present in

Curry, it must exist a different bridge operator for every supported scheme cooperation. How can the system infer what bridge to apply is described in section 6.4.

Let's call the *tell* operation for bridges *link*, then link can be just defined as:

$$link\ c\ \texttt{a\#=b} = c \cup \{bridge(a, b)\}$$

5.2 *Hooking bridge propagation*

Unfortunately, the *link* definition above is not enough, as bridges must *propagate* the constraints to be effective.

This means bridges must be able to access other constraint stores, and modify them. The way we achieve this is by redefining the given *tell* definition (in section 4), so the constraint propagation takes place.

Let's define a new *tell'*, to be used when the system supports cooperation, so for a bridge $\texttt{\#=}_{D,E}$, meaning that it links variables of the domain D with E, and be B the domain of the bridge constraint solver, then the new *tell* would be:

$$tell'\ c\ e = \begin{cases} eval_{e,p} & \text{if } tell_D\ c[D]\ e = eval_{e,p} \\ fail & \text{if } tell_D\ c[D]\ e = fail \\ check(e, D, c) & \text{otherwise} \end{cases}$$

where $e = op_D(e_1, \ldots, e_n)$ and

$$check(e, D, c) = tell_D(c, e) \quad \text{if doesn't exist a } bridge(a, b) \in c[B] \text{ for all variables } a \text{ in } c[D]$$

$$check(e, D, c) = hp(e, c[D], c[E]) \quad \text{for all } bridge(a, b) \in c[B] \text{ and } a \in c[D], b \in c[E]$$

5.3 *The cooperation mechanism*

The last bit needed is to define the *hp* operator. For this we need a way to:

- Find the projection of the new domain for x over y. That is, find what constraints over y are *equivalent* to the conditions just imposed on x. Let us call *propagate* the function that receives the origin constraint store, the destination one, the newly *told* constraint and returns the relevant projection (i.e. $propagate(S_x, S_y, e)$) will be calculated so as to find a new constraint $C_{S_x \to S_y}$).
- The constraint $C_{S_x \to S_y}$ must be fed into the destination solver S_y: $tell(C_{S_x \to S_y})$.

Then *hp* gets defined as

$$hp(e, c_D, c_E) = \begin{cases} c[D] = tell_D\ c[D]\ propagate(D, E, e, c_D) \\ c[E] = tell_E\ c[E]\ propagate(E, D, e, c_E) \end{cases}$$

Propagation has to be done currently in both ways.

6 Modifications to Curry syntax

In the previous sections we have discussed the operational semantics, without digging into the details of the actual modifications to the language itself.

In fact, for this proposal to be usable at source level, we need to greatly improve some of other language features.

Our preliminary proposal is just to add type classes [13] to Curry and let them resolve the likely ambiguity in constraint predicates.

6.1 Using type classes

The basic operator we want to support is the constrained equality `=:=` operator:

```
class Constrainable a where
   =:= :: a → a → a
```

But, what about algebraic data types? We have suddenly lost the possibility of doing

```
data A = A
> x =:= A
```

as the compiler cannot find an instance of `Constrainable` for A.

There are two possible remedies for this:

- Require a `deriving` declaration for every ADT we want to be constrainable:

  ```
  data A = A
       deriving(Show,Constrainable)
  ```

- Let the compiler automatically derive such an instance for every `data` declaration.

This has the effect[10] of tagging every function where `=:=` is used with the `Constrainable` type constraint:

```
f x y z = x =:= y & y =:= z
> :t f
> f has type Constrainable a ⇒ a → a → a → Success
```

6.2 Constraint flattening

Then, using `Success` as our basic tool for constraint programming feels a little bit unnatural when dealing with functions. Thus the objective of constraint flattening is to allow constraints predicates to be represented as pure functions, by hiding the step of creating a new fresh variable from the user.

Think about the constraint predicate `:+: :: a →a → a → Success`, which is the usual sum constraint operator present in CLPR.

Then, as we have guaranteed that every constraint domain will at least support `=:=` we can implement a more natural plus operator that allows us to sum a variable

[10] We think it is a nice side effect, but we recognize that this “tagging” can be seen as very cumbersome by some people

number of reals, and to get the resulting constraint operator.

```
(+) :: Constrainable a ⇒ a → a → a
a + b = (:+:) a b c &> c where c free
```

This way, we can write the expression:

```
> let a,b free in a + b - 3 + a =:= 3
> Success {2a + b = 6}
```

and the flattening does its job.

6.3 More type classes

Of course, once we have settled the basic class for constrainable data types, we can extend other standard type classes to include support for constraints.

For instance the `Num` type class could look like:

```
class Constrainable a ⇒ Num a where
  (+) :: a → a → a
...
```

We did informally implement this kind of behavior with run-time type detecting, but of course, we believe using type classes for this is a way better approach.

6.4 Representing bridges

This type classes scheme could allow more sophisticated constraints operations, such as representing bridges for constraint solving cooperation:

```
class (Constrainable a, Constrainable b) ⇒ Bridge a b ↩
   where
  =# :: a → b → Success
```

This however is a little bit far fetched, as requires multiparametric type classes[12].

7 Conclusions and future work

We have defined an operational semantics for Curry that allows to incorporate new constraint solvers to a constraint neutral core. This semantics makes explicit the genericity that was suggested by a previous semantics for Curry with disequality constraints over the Herbrand universe.

In order to allow for this modularity we require some functionality to be provided by the constraint solvers, namely the ability to detect and communicate demand information to the core semantics. This simple extension permits a fully lazy behaviour for constraint expressions and a uniform interface when adding new solvers to the core.

We have shown that the semantics is expressive enough to incorporate the constraint extensions currently present in one implementation of Curry (Sloth), namely: equality and disequality constraints over Herbrand terms, linear constraints over rationals, finite domain constraints over naturals and propositional constraints.

In our opinion this is at least a good indication on the practicality of this approach but, of course, a correctness and completeness proof of the semantics is needed. This will be the immediate step following this presentation.

The paper contains also some ideas on the realization of solver cooperation in our framework. These are far more speculative, and it is required to test its practicality by means of representative examples, and to investigate more on the role that type classes can play in this integration.

References

[1] Colmerauer, A., *An introduction to prolog III*, Commun. ACM **33** (1990), pp. 69–90.

[2] Estevez, S., A. J. Fernandez, T. Hortala, M. Rodríguez Artalejo and R. del Vado Vírseda, *A fully sound goal solving calculus for the cooperation of solvers in the cflp scheme*, in: *WFLP '06: Proceedings of the 2006 workshop on Curry and functional logic programming* (2006), to appear.

[3] Gallego Arias, E. J. and J. 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* (2005), pp. 66–69.

[4] Gallego Arias, E. J., J. Mariño and J. M. Rey, *A proposal for disequality constraint in curry*, in: *WFLP '06: Proceedings of the 2006 workshop on Curry and functional logic programming* (2006), to appear.

[5] Hanus, M., S. Antoy, H. Kuchen, F. J. López-Fraguas, W. Lux, J. J. Moreno-Navarro and F. Steiner, "Curry: An Integrated Functional Logic Language," 0.8.2 edition (2006), editor: Michael Hanus.
URL http://www.informatik.uni-kiel.de/~mh/curry/report.html

[6] Hofstedt, P., *Cooperating constraint solvers*, in: *Principles and Practice of Constraint Programming*, 2000, pp. 520–524.
URL citeseer.ist.psu.edu/hofstedt00cooperating.html

[7] Jaffar, J. and M. J. Maher, *Constraint logic programming: A survey*, Journal of Logic Programming **19/20** (1994), pp. 503–581.
URL http://citeseer.ist.psu.edu/jaffar94constraint.html

[8] López-Fraguas, F. J., M. Rodríguez-Artalejo and R. del Vado Vírseda, *A lazy narrowing calculus for declarative constraint programming.*, in: E. Moggi and D. S. Warren, editors, *PPDP* (2004), pp. 43–54.

[9] López-Fraguas, F. J. and J. Sánchez-Hernández, *Disequalities may help to narrow.*, in: M. C. Meo and M. V. Ferro, editors, *APPIA-GULP-PRODE*, 1999, pp. 89–104.

[10] Lux, W., *Adding linear constraints over real numbers to curry*, in: *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming* (2001), pp. 185–200.

[11] Mariño, J. and J. M. Rey, *Adding constraints to curry via flat guards*, in: *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming* (2005), pp. 14–22.

[12] Simon Peyton Jones, M. J. and E. Meijer, *Type classes: exploring the design space*, in: *Procedings of the Haskell Workshop 1997*, 1997.
URL
http://research.microsoft.com/Users/simonpj/Papers/type-class-design-space/multi.ps.gz

[13] Wadler, P. and S. Blott, *How to make ad-hoc polymorphism less ad-hoc*, in: *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages* (1989), pp. 60–76.
URL citeseer.ist.psu.edu/wadler88how.html