

Disjunctive Constraint Lambda Calculi

Matthias M. Hölzl¹ and John N. Crossley^{2,*}

¹ Institut für Informatik, LMU, Munich, Germany
Matthias.Hoelzl@ifi.lmu.de

² Faculty of Information Technology, Monash University, Australia
John.Crossley@infotech.monash.edu.au

Abstract. Earlier we introduced Constraint Lambda Calculi which integrate constraint solving with functional programming for the simple case where the constraint solver produces no more than one solution to a set of constraints. We now introduce two forms of Constraint Lambda Calculi which allow for multiple constraint solutions. Moreover the language also permits the use of disjunctions between constraints rather than just conjunction. These calculi are the Unrestricted, and the Restricted, Disjunctive Constraint-Lambda Calculi. We establish a limited form of confluence for the unrestricted calculus and a stronger form for the restricted one. We also discuss the denotational semantics of our calculi and some implementation issues.

1 Introduction

Constraint programming languages have been highly developed in the context of logic programming (see e.g. [9, 3] and, regarding confluence, [14]). In [11] Mandel initiated the use of the lambda calculus as an alternative to a logic programming base. There were many difficulties and, in particular, the treatment of disjunction was not very satisfactory (see [12]). It has turned out to be surprisingly difficult to get a transparent and elegant system for the functional programming paradigm. This was ultimately accomplished in [6] and [8], where we introduced the unrestricted and restricted constraint-lambda calculi. In this paper we expand the language of these calculi to include disjunction in constraints.

The basic problem with the introduction of disjunction or, indeed with multiple solutions, is easily demonstrated by the example (first noted, we believe, by Hennessy [5]) $(\lambda x.x + x)(2|3)$ where “2|3” means “2 or 3”. If a choice is first made of a value of the disjunction “2|3”, then there are two answers: 4 and 6. If the β -reduction is performed first, then the result is $(2|3) + (2|3)$. In this case there is also the possible interpretation that the first value should be chosen to be 2 and the second to be 3 (or *vice versa*) yielding an additional answer: 5.

We propose two solutions, one for each possibility, in Sections 2 and 8.

* Special thanks to Martin Wirsing for his support, interest and extremely helpful criticism. Thanks also to three anonymous and helpful referees.

The systems that we define are extensions of our calculi in [8]. Because we now have multiple solutions as a matter of course we cannot expect confluence.¹ Nevertheless we are able to establish a weaker property, which we call *path-confluence*, in Theorem 2 for the Restricted Disjunctive Constraint-Lambda Calculus.

We briefly discuss the denotational semantics of our systems and implementation issues. Then we turn to the question of multiple constraint stores and finally we compare our systems with the earlier work of Mandel and Cengarle [13] and other current approaches to constraint-functional programming integration.

2 Unrestricted Disjunctive Constraint-Lambda Calculus

The Constraint Language. A constraint is a relation that holds between several entities from a fixed domain. We assume a notion of equality, denoted by $=$, is given. Typical constraint domains are the real numbers, the integers, or a finite subset of the integers.

A *constraint language* is a 4-tuple $\mathcal{L} = (\mathcal{C}, \mathcal{V}, \mathcal{F}, \mathcal{P})$, where $\mathcal{C} = \{c_1, c_2, \dots\}$ is a set of *individual constants*, $\mathcal{V} = \{X_1, X_2, \dots\}$ is a set of *constraint variables*, $\mathcal{F} = \{f_1, f_2, \dots\}$ is a set of function letters with fixed arity, and $\mathcal{P} = \{P_1, P_2, \dots\}$ is a set of *predicate symbols*, again with fixed arities. We assume that a constant, \perp , representing the undefined value is included in \mathcal{C} . The set \mathfrak{T} of *constraint terms* over a constraint language \mathcal{L} is defined inductively in the usual way. Constraint terms containing no variables are called *ground* and \mathfrak{T}_g is the set of all ground constraint terms. Model-theoretic notions such as *model* and *satisfaction* are defined for sets of formulae in the constraint language in the usual way.

Definition 1. *If P is a predicate letter with arity n and t_1, \dots, t_n are constraint terms, then $P(t_1, \dots, t_n)$ is an atomic constraint. The set of constraints \mathfrak{C} is the closure of the atomic constraints under conjunction (\wedge) and disjunction (\vee). The empty conjunction is written as *true* and the empty disjunction as *false*.*

Definition 2 (Inconsistent constraints). *A set $S = \{C_1, C_2, \dots, C_n\}$ of constraints is said to be inconsistent, if S is not satisfiable.*

The denotation of a constraint term in a constraint language \mathcal{L} over a constraint domain D , is defined by evaluating it in the usual way (which gives the usual properties): $value : (\mathcal{V} \rightarrow D) \rightarrow \mathfrak{T} \rightarrow D$. So if $\theta : \mathcal{V} \rightarrow D$ then $value(\theta) : \mathfrak{T} \rightarrow D$.

Convention 1 (Canonical names). *We assume that there is an idempotent mapping (canonical naming) $n : \mathfrak{T}_g \rightarrow \mathcal{C}$ with the following properties:*

$$value(\theta)(n(t)) = value(\theta)(t) \tag{1}$$

$$(value(\theta)(t_1) = value(\theta)(t_2)) \implies n(t_1) \equiv n(t_2) \tag{2}$$

¹ Confluence is the property that when a lambda-calculus-style term M is reduced in two different ways (possibly in many steps) to M_1 and M_2 then (up to renaming of bound variables) there is a third term M_3 to which both M_1 and M_2 reduce.

for all maps $\theta : \mathcal{V} \rightarrow D$, where $=$ is the semantic equality of the constraint domain and \equiv is syntactic equality. The image of a ground constraint term under n is called its canonical name, the image of the constraint domain under n is the set of canonical names. We write cn or cn_i for canonical names and CN for $n[\mathcal{T}_g]$.

A *constraint store* is a set of constraints. The only operation on constraint stores is the addition of a new constraint to the store, denoted by $S \oplus C$:

$$S \oplus C = S \cup \{C\}.$$

We shall only be concerned with formulae, principally equations, implied by a constraint store S , therefore a constraint solver may simplify the set of constraints contained in the constraint store without changing the possible reductions. Since, for our purposes, all inconsistent stores are equivalent, we write \otimes to denote any inconsistent store and we then write $S = \otimes$.

Syntax. The syntax for constraint-lambda terms is given by:²

$$\begin{aligned} A &::= x \mid X \mid c \mid f(A, \dots, A) \mid \lambda x. A \mid \Lambda A \mid \{GC\}A, \\ GCT &::= A, \quad GC ::= P(GCT, \dots, GCT) \mid (GC \wedge GC) \mid (GC \vee GC). \end{aligned}$$

The syntactic categories are:

- **Constraint-lambda terms** (A): These are the usual lambda terms augmented with a notation for constraint-variables (variables whose values are computed by the constraint solver) and a notation to describe the addition of constraints to the constraint store.
- **General constraint terms** (GCT): These are augmented terms of the constraint language. Constraint-variables may appear as part of a lambda term or as part of a general constraint term. This makes it possible to transfer values from the constraint store to lambda terms. Similarly, a lambda term may appear inside a constraint term. Having lambda variables inside constraints allows us to compute values in the lambda calculus and introduce them as part of a constraint. We also allow arbitrary lambda terms inside constraints. These terms have to be reduced to constraint terms before being passed to the constraint solver.
- **General constraints** (GC): These are primitive constraints as well as disjunctions and conjunctions of constraints (defined in terms of general constraint terms instead of the usual constraint terms). They correspond to, but are slightly more general than, the notion of constraint in the previously defined constraint-language, since they may include lambda terms as constituents.

² In the rest of the paper we sometimes omit the parentheses around disjunctions and conjunctions.

Note. The generalized constraint terms correspond exactly to the constraint-lambda terms. Nevertheless we consider it important to distinguish these two sets, since the set of *pure* constraint-lambda terms and *pure* constraint terms are disjoint:

Definition 3. We call a constraint-lambda term *pure* if it contains no term of the form $\{C\}M$; we call a constraint term *pure* if it contains no lambda term, i.e., if the only constraint-lambda terms it contains are constraint variables, constants or applications of function-symbols to pure constraint terms. A constraint C is called a *pure constraint* if every constraint term appearing in C is pure. We write Λ_p for the set of all pure constraint-lambda terms not containing \perp .

Free and bound variables and substitution are defined in a straightforward way (see [6] for details). Only lambda variables may appear as free and bound variables, i.e., $FV(X) = \emptyset = BV(X)$. As usual we identify α -equivalent terms, so we can freely rename bound variables and also ensure no variable appears both free and bound in M . We postulate the following:

Convention 2 (Variable Convention). The following property holds for all λ -terms M : No variable appears both free and bound in M , $FV(M) \cap BV(M) = \emptyset$. Furthermore, we can always assume by changing bound variables (if necessary) that for different subterms $\lambda x.M_1$ and $\lambda y.M_2$ of M , we have $x \neq y$.

Reduction Rules. It is necessary to take the constraint stores into account in defining the reductions of our constraint terms since the stores interact with these terms, so we define reductions on pairs (M, S) where S is a constraint store.

Rule 1. Fail on an Inconsistent Store $(M, \otimes) \rightarrow (\perp, \otimes)$ (\perp)

Rule 2. Beta-reduction $((\lambda x.M)N, S) \rightarrow (M[x/N], S)$ (β)

Rule 3. Reduce Pure Constraint Terms
 $(C, S) \rightarrow (n(C), S)$ if C is pure, $C \in \mathfrak{T}_g$ and $C \neq n(C)$ (CR)

Rule 4. Introduce Constraint
 $(\{C\}M, S) \rightarrow (M, S \oplus C)$ if C is a pure constraint (CI)

Rule 5. Use Constraint
 $(X, S) \rightarrow (cn, S \oplus (X = cn))$ if $(S \oplus (X = cn)) \neq \otimes$ and $cn \in CN$ (CS)

Notes on the rules.

Rule 1. Reductions resulting in inconsistent stores correspond to failed computations in logic programming languages.

Rule 2. We allow full beta-reduction in the disjunctive constraint-lambda calculi. E.g., if we have the integers as constraint domain, $(\lambda x.x + 1)5 \rightarrow 5 + 1$.

Rule 3. This rule ties the constraint system into the lambda calculus. E.g., continuing our example: $5 + 1 \rightarrow 6$. We do not allow arbitrary transformations

between pure constraint terms, since this does not increase the expressive power of the system.³

Rule 4. We only allow pure constraints to be passed to the constraint store since otherwise the constraint solver could perform transformations other than β -reduction on lambda terms. This would increase the power of the system since “oracles” might be introduced as predicates in the constraint language. But it would also require the constraint theory to be a true superset of the lambda calculus. This would pose a major problem for practical applications of the calculus, since most constraint systems cannot handle lambda terms.

Rule 5. A constraint variable may be instantiated to any value that is consistent with the constraint store. We only introduce canonical names into the lambda term since this allows us to obtain confluent restrictions of the disjunctive calculus. We introduce the constraint $X = cn$ into the constraint store to remove the possibility of substituting different values for the same variable.

Definition 4. We say a constraint lambda term M is reducible with store S if one of the rules (\perp) , (β) , (CR) , (CI) or (CS) is applicable to the pair (M, S) . We say M is reducible if it is reducible for all stores S . We write $M \rightarrow M'$ as an abbreviation for $\forall S. \exists S'. (M, S) \rightarrow (M', S')$.

We call a sequence of zero or more reduction steps $(M_1, S_1) \rightarrow (M_2, S_2), \dots, (M_{n-1}, S_{n-1}) \rightarrow (M_n, S_n)$ a reduction sequence and abbreviate it by $(M_1, S_1) \rightarrow^* (M_n, S_n)$. We write $M \rightarrow^* M'$ as an abbreviation for $\forall S. \exists S'. (M, S) \rightarrow^* (M', S')$.

Example 1. Without the addition of $X = M$ to the store we would have:

$$\begin{aligned} (X + X, \{X = 2 \vee X = 3\}) &\rightarrow (2 + X, \{X = 2 \vee X = 3\}) \\ &\rightarrow (2 + 3, \{X = 2 \vee X = 3\}). \end{aligned}$$

If we add the new constraint to the store, there are only two (essentially different) possible reduction sequences:

$$\begin{aligned} (2) \quad &(X + X, \{X = 2 \vee X = 3\}) \rightarrow (2 + X, \{X = 2 \vee X = 3, X = 2\}) \\ &\quad \rightarrow (2 + 2, \{X = 2 \vee X = 3, X = 2\}) \\ (3) \quad &(X + X, \{X = 2 \vee X = 3\}) \rightarrow (3 + X, \{X = 2 \vee X = 3, X = 3\}) \\ &\quad \rightarrow (3 + 3, \{X = 2 \vee X = 3, X = 3\}). \end{aligned}$$

Obviously the order in which the variables are instantiated can be changed.

We need to have the reductions commute with the constructions of constraints in order to allow reductions of subterms. (For example, a pair of the form $(\lambda x. (\lambda y. y)x, S)$ ought to be reducible to $(\lambda x. x, S)$.) If the reduction of a subterm changes the store, then this change propagates to the store associated with the enclosing term. We give only a few examples. If $(M, S) \rightarrow (M', S')$,

$$\begin{aligned} &(f(M_1, \dots, M, \dots, M_n), S) \rightarrow (f(M_1, \dots, M', \dots, M_n), S') \\ &(L \wedge M, S) \rightarrow (L \wedge M', S'), \quad (LM, S) \rightarrow (LM', S') \\ &(\lambda x. M, S) \rightarrow (\lambda x. M', S'), \quad (\{M\}N, S) \rightarrow (\{M'\}N, S') \end{aligned}$$

³ This rule was not included in our earlier work [8] but it is easy to verify that it does not affect the confluence properties.

To avoid infinite reduction paths where the terms differ only in the names of constraint variables we impose:

Convention 3. *We assume a well-founded partial order \prec on the set of constraint variables. Substitution in rule (CS) is only allowed if, for every variable Y in M , we have $Y \prec X$.*

Example 2. We write $(x|y)_X$ as an abbreviation for $\{X = x \vee X = y\}X$ with a fresh constraint-variable X . When we reduce the term $(\lambda x.x + x)(2|3)_X$ with an empty constraint store, we obtain as one possible reduction sequence:

$$\begin{aligned}
((\lambda x.x + x)(2|3)_X, \{\}) &\rightarrow (\{X = 2 \vee X = 3\}X + \{X = 2 \vee X = 3\}X, \{\}) \\
&\rightarrow (X + \{X = 2 \vee X = 3\}X, \{X = 2 \vee X = 3\}) \\
&\rightarrow (2 + \{X = 2 \vee X = 3\}X, \{X = 2\}) \\
&\rightarrow (2 + X, \{X = 2\}) \\
&\rightarrow (2 + 2).
\end{aligned}$$

3 Confluence

It is not possible to have confluence in the traditional sense for the unrestricted calculus because different reductions can lead to different constraint stores as well as to different solutions.

Example 3. Consider the pair $((\lambda x.X)(\{X = cn\}M), \emptyset)$, where the constraint store is initially empty. This can be reduced in two different ways. In the first the final store contains $X = cn$ but in the second the store remains empty and it is not possible to carry out any further reduction. Thus we have the reductions:

$$\begin{aligned}
((\lambda x.X)(\{X = cn\}M), \emptyset) &\rightarrow ((\lambda x.X)M, \{X = cn\}) && \text{by (CI)} \\
&\rightarrow (X, \{X = cn\}) && \text{by } (\beta) \\
(*) &\rightarrow (cn, \{X = cn\}) && \text{by (CS)}
\end{aligned}$$

but we also have

$$(**) ((\lambda x.X)(\{X = cn\}M), \emptyset) \rightarrow (X, \emptyset) \text{ by } \beta\text{-reduction,}$$

and there is no way to reduce $(*)$ and $(**)$ to a common term.

Note that the constraint store may contain different sets of constraints at different stages of the the reduction so that, while a constraint substitution may not be possible at some reduction step, it may become possible later.

Definition 5. *Suppose that in a reduction sequence $(M_1, S_1) \rightarrow^* (M_n, S_n)$ we apply rule (CS) zero or more times and replace X_i by cn_i . If a store S exists, such that, for all these applications of rule (CS), we have $S \models X_i = cn_i$, then we say that $(M_1, S_1) \rightarrow^* (M_n, S_n)$ is a reduction sequence that can be restricted to store S .*

Let $(M, S) \rightarrow^ (M_1, S_1)$ and $(M, S) \rightarrow^* (M_2, S_2)$ be two reduction sequences. We say these reduction sequences are compatible if $S_1 \cup S_2$ is consistent.*

Definition 6. We call the following property confluence as a reduction system: For every pair of reductions $(M, S) \rightarrow^* (M_1, S_1)$ and $(M, S) \rightarrow^* (M_2, S_2)$ such that both reduction sequences can be restricted to store S there exist a term N and stores S'_1, S'_2 such that $(M_1, S_1) \rightarrow^* (N, S'_1)$ and $(M_2, S_2) \rightarrow^* (N, S'_2)$.

Example 1 shows that the unrestricted disjunctive constraint-lambda calculus is not confluent as a reduction system since different reductions may introduce different values for a constraint variable. But if two reductions introduce the same values for all constraint-variables then their results can be reduced to a common term. This property is made explicit in the remainder of this section.

Since each application of the rule (CS) introduces a constraint $X_i = cn_i$ into the store it is clear that all applications of rule (CS) for a variable X in two compatible reduction sequences substitute the same value for X . From this we may conclude that the reduction sequences $(M, S_1 \cup S_2) \rightarrow^* (M_1, S_1 \cup S_2)$ and $(M, S_1 \cup S_2) \rightarrow^* (M_2, S_1 \cup S_2)$ (obtained from the original sequences by extending the stores but not changing any reductions) are reduction sequences in the single-valued calculus of [8]. These sequences can trivially be restricted to $S_1 \cup S_2$. It follows from the confluence as a reduction system of the single-valued constraint-lambda calculus which was proved as Theorem 1 in [8] that there is a term N and a store S' such that both $(M_1, S_1 \cup S_2)$ and $(M_2, S_1 \cup S_2)$ reduce to (N, S') . We therefore have:

Theorem 1. Let $(M, S) \rightarrow^* (M_1, S_1)$ and $(M, S) \rightarrow^* (M_2, S_2)$ be compatible reduction sequences. Then there is a term N and a store S' such that both $(M_1, S_1 \cup S_2)$ and $(M_2, S_1 \cup S_2)$ reduce to (N, S') .

4 Restricted Disjunctive Constraint-Lambda Calculus

The restricted constraint-lambda calculus has the same reduction rules as the unrestricted constraint-lambda calculus, but the allowed terms are only those from λI (not λK , see Barendregt [1], Chapter 9) so an abstraction $\lambda x.M$ is only allowed if $x \in FV(M)$:

Definition 7. The set of restricted constraint-lambda terms, RCTs, Λ_I is defined inductively by the following rules:

- Every lambda variable x and every constraint variable X is a RCT.
- If M is a RCT and $x \in FV(M)$, then $\lambda x.M$ is a restricted lambda term.
- If M and N are restricted lambda terms, then MN is a RCT.
- If C is an extended constraint and M a restricted constraint-lambda term, then $\{C\}M$ is a RCT.

The sets of extended constraints and extended constraint terms (corresponding to GC and GCT) are defined similarly to the sets of general constraints and general constraint terms, but with RCTs in place of general constraint terms.

We write $M \in \Lambda_I$ if M is a restricted lambda term.

We use the same conventions as for the unrestricted constraint-lambda calculus, most importantly, we use the variable convention. The reduction rules for the restricted constraint-lambda calculus are the same as for the unrestricted constraint-lambda calculus. The terms of the restricted constraint-lambda calculus satisfy certain properties that are not necessarily true of unrestricted terms.

- Lemma 1.**
1. $\lambda x.M, N \in \Lambda_I \implies M[x/N] \in \Lambda_I$,
 2. $\lambda x.M \in \Lambda_I \implies FV((\lambda x.M)N) = FV(M[x/N])$,
 3. $M \in \Lambda_I, M \rightarrow^* N \implies N \in \Lambda_I$, and
 4. $M \in \Lambda_I, M \rightarrow^* N, N \neq \perp \implies FV(M) = FV(N)$.

For the proof see [6].

The following Lemma holds for terms of Λ_I . A *normal form* is a term which cannot be reduced.

Lemma 2. *Let $M \in \Lambda_I$. If $(M, S) \rightarrow^* (N, S')$, where N is a normal form, then every reduction path starting with (M, S) is finite.*

The proof is similar to the one in [1]. We make use of the previously introduced Convention 3 for rule *(CS)* (see page 6) to show that no infinite *(CS)*-reduction sequences can occur. This Lemma is also true for the restricted single-valued calculus. Since we make no other use of this Lemma we omit the details.

5 Path-Confluence

The single-valued restricted constraint-lambda calculus was proved in [8] to be confluent so we can improve Theorem 1 for terms of the Restricted Disjunctive Constraint-Lambda Calculus to the following whose proof may be found in [6]. Path-confluence requires a controlled sequence of choices of extensions to stores.

Theorem 2 (Path-confluence). *Let M be a RCT and let $(M, S) \rightarrow^* (M_1, S_1)$ and $(M, S) \rightarrow^* (M_2, S_2)$ be compatible reduction sequences. Then there is a term N and a store S' such that both (M_1, S_1) and (M_2, S_2) reduce to (N, S') .*

6 Denotational Semantics

We defined the denotational semantics of the constraint-lambda calculus without disjunction in [8] and we recall only a few key points here. We let E denote the semantic domain of the constraint-lambda terms. The denotational semantics are defined in such a way that each model for the usual lambda calculus can be used as a model for the constraint-lambda calculus provided that the model is large enough to allow an embedding $emb : D \rightarrow E$ of the underlying constraint domain D into E . This is usually the case for the constraint domains appearing in applications. As usual we have an isomorphism $E \rightarrow E \simeq E$ (see e.g. [1], chapter 5). We denote environments by η (a mapping from lambda variables to E). We can then define a semantic valuation from the set of constraint terms, \mathcal{T} , into D which we call $val : \mathcal{T} \rightarrow D$. We shall write val' for $emb \circ val : \mathcal{T} \rightarrow E$.

We associate a pure lambda term with every constraint-lambda term by replacing all constraint variables by lambda variables. Let M be a constraint-lambda term with constraint variables $\{X_1, \dots, X_n\}$ and let $\{x_1, \dots, x_n\}$ be a set of distinct lambda variables not appearing in M . Then the *associated constraint-variable free term*, $cvt(M)$, is the term

$$\lambda x_1 \dots \lambda x_n. (M[X_1/x_1] \dots [X_n/x_n]).$$

We separate the computation of a constraint-lambda term into two steps. First we collect all constraints appearing in the term and compute all the lambda terms contained therein in the appropriate context. Then we apply the associated constraint-variable free term to the values computed by the constraint-solver to obtain the value of the constraint-lambda term.

For a constraint-lambda term M and store S we set

1. \mathfrak{D}_η as the denotation of a constraint-lambda term in an environment η when the constraints are deleted from the term.⁴
2. The function $\mathfrak{C}^{\mathcal{C}}$ applied to the constraint-lambda term, M , collects all constraints appearing in M and evaluates the lambda expressions contained within these constraints. The superscript \mathcal{C} on \mathfrak{C} denotes the recursively generated context.

The semantics of a single-valued constraint-lambda term with respect to a store S is defined as

$$\llbracket (M, S) \rrbracket = \{ \mathfrak{D}_\eta(cvt(M)v_1 \dots v_n) \mid S \cup \mathfrak{C}^\circ(M) \vdash X_1 = v_1, \dots, X_n = v_n \}$$

where \mathfrak{D}_η defines the usual semantics for pure lambda terms and ignores constraints contained within a term. The superscript \circ on \mathfrak{C} indicates that we are starting with the empty context and building up \mathfrak{C} as we go into the terms. The environment η is supposed to contain bindings for the free variables of M .

Intuitively, this definition means that the semantics of a single-valued constraint-lambda term is obtained as the denotation of the lambda term when all constraints are removed from the term and all constraint-variables are replaced by their values. In particular we have (by footnote 4):

Fact 1 *The denotational semantics of a pure lambda term is the same as in the traditional denotational semantics.*

The denotation of a constraint-lambda term in an environment η , \mathfrak{D}_η , is defined as follows:⁵

$$\begin{aligned} \mathfrak{D}_\eta(\lambda x. M) &= \lambda v. \mathfrak{D}_{\eta[x/v]}(M) \\ \mathfrak{D}_\eta(x) &= \eta(x) & \mathfrak{D}_\eta(MN) &= \mathfrak{D}_\eta(M)\mathfrak{D}_\eta(N) \\ \mathfrak{D}_\eta(c) &= val'(c) & \mathfrak{D}_\eta(\{C\}M) &= \mathfrak{D}_\eta(M) \\ \mathfrak{D}_\eta(f(M_1, \dots, M_n)) &= val'(f)(\mathfrak{D}_\eta(M_1), \dots, \mathfrak{D}_\eta(M_n)) \end{aligned}$$

⁴ Therefore, for pure constraint-lambda terms, \mathfrak{D}_η represents the usual semantics.

⁵ Notice that the semantic function \mathfrak{D} is only applied to constraint-variable-free terms and that it does not recurse on constraints, therefore there is no need to define it on constraints or constraint terms. Furthermore the interpretations of a constant, when regarded as part of a lambda term or as part of a constraint, coincide, as expected.

When evaluating lambda terms nested inside constraints, we are only interested in results that are pure constraints, since the constraint solver cannot handle any other terms. Therefore we identify all other constraint-lambda terms with the failed computation.

We can now show that the semantics of a constraint-lambda term is compatible with the reduction rules.

Lemma 3. *For all environments η and all terms M, N , we have*

$$\mathfrak{D}_\eta(M[x/N]) = \mathfrak{D}_{\eta[x/\mathfrak{D}_\eta(N)]}(M).$$

For unrestricted constraint-lambda terms without disjunction we may lose a constraint during the reduction and then we get $\llbracket(M, S)\rrbracket \supseteq \llbracket(M', S')\rrbracket$. However in the case of the disjunctive calculus the situation is reversed: Now a smaller set of constraints implies a larger set of values, therefore if $(M, S) \rightarrow (M', S')$ it may be the case that $\llbracket(M', S')\rrbracket$ contains values that are not contained in $\llbracket(M, S)\rrbracket$. Therefore the operational semantics are not correct with respect to the denotational semantics in this case. This, however, is not surprising if we consider the meaning of $\llbracket(M, S)\rrbracket$. We have defined the semantics so that this expression denotes the precise set of values that can be computed in such a way that all constraints are satisfied. If constraints are dropped during a β -reduction step the new term places less restrictions on the values of the constraint variables, thus we obtain an approximation “from above” as the semantics of the new term.⁶

7 Implementation issues

Application of the rule (CS) to a variable with a large range of possible values may lead to many unnecessary reductions. If, for example, we introduce

$$(\{X = 100\}X, \{1 \leq X, X < 500\})$$

⁶ The evaluation of constraints in the denotational semantics is currently done in a very “syntactical” manner. To see why this is the case, we have to make a short digression into the motivations for defining the semantics in the way they are defined. As M. B. Smyth points out in [18], the Scott topology is just the Zariski topology ([2]) on the ring defined by the lattice structure of the domain in question and corresponds to the notion of an *observable property*. It is evident that this topology cannot be Hausdorff for any interesting domain. The denotational semantics of logic programming languages, on the other hand, is generally defined on the Herbrand-universe, and the fixed points are calculated using consequence operators, see [10] or [4]. It seems that these two methods of defining the denotational semantics do not match well. A more natural approach in our setting would be to regard the predicates of the constraint theory as boolean functions over the constraint domain and constraints as restrictions on the known ranges of these functions. However, this definition results in a Hausdorff topology on the universe in question, and is therefore incompatible with the topology of the retract definition. It would be interesting to see whether this problem can be resolved by a suitable denotational semantics for the constraint theory. The resulting topology shows another problem: A Hausdorff topology cannot be the topology resulting from observable properties. This suggests a connection with the sometimes difficult to control behavior of constraint programs.

with rule (CS) we may have to try many substitutions for X before instantiating X with the only value that does not lead to an inconsistent store in the next reduction step. If we introduce the constraint $X = 100$ into the store the next reduction step immediately leads to the normal form 100. Therefore one has to be careful not to apply the (CS)-rule indiscriminately in an implementation of the constraint-lambda calculus. We discuss practical issues about implementation in our paper [7].

For applications of the constraint-lambda calculus it is sometimes useful to extend the system with additional capabilities. One such extension is the addition of multiple constraint stores, another is the computation of fresh constraint variables. We discuss this extension in the next section. It adds some additional complexity to the calculus but we think that this is more than compensated for by the added expressive power.

8 Multiple Constraint Stores

For some applications it is desirable to split the problem into several smaller parts and to have each part operate on its own constraint store. This can be done by extending the constraint-lambda calculus to incorporate multiple constraint stores. The addition of multiple stores allows us to provide a choice for the following problem: If a function is applied to a non-deterministic argument, should all references to this argument be instantiated with the same value or should it be possible to instantiate each reference individually? For example, should $(\lambda x.x + x)(2|3)$ return only the values 4 and 6 or should it also return 5? In Section 2 we restricted ourselves to the first solution. With the extension discussed in this section we allow the user to choose the preferred alternative by means of a *store assignment*. To keep the strict separation between program logic and control, the store assignment is defined on the meta-level.

Syntax. When we add multiple stores to a constraint-lambda calculus we need a means of showing on which store the rules (CI) and (CS) operate. To this end we extend the syntax of the calculus with *names for stores*, denoted by the letter S (with indices and subscripts if necessary) and with *locations*. Syntactically, any constraint-lambda term can be used as a location, but only locations evaluating to a store-name can actually select a store. We write the locations as superscripts to other constraint-lambda terms. For example, in the term M^N , the term N is used as the location for M . In terms of the form $\{C\}^N M$, the term N is used as the location for the constraint C , and terms of the form $\{C\}M$ without location for the constraint C are not valid terms of the constraint-lambda calculus with multiple stores. The context-free syntax is therefore

$$A ::= \perp \mid x \mid X \mid c \mid S \mid A^A \mid f(A, \dots, A) \mid \lambda x.A \mid (AA) \mid \{GC\}^A A.$$

We write \mathcal{N} for the set of all names for stores and \mathfrak{C} for the set of all constraints. We extend substitution to the new terms in the natural way:

$$S[x/L] = S; M^N[x/L] = M[x/L]^{N[x/L]}; (\{C\}^N M)[x/L] = \{C[x/L]\}^{N[x/L]} M[x/L].$$

Reduction Rules. We want to be able to “alias” store names, i.e., we want to be able to have two different names refer to the same constraint store. Therefore we define reductions on triples $(M, \sigma, \mathfrak{S})$ where M is a constraint-lambda term, σ is a map from store names to integers, $\sigma : \mathcal{N} \rightarrow \omega$ and \mathfrak{S} is a map from integers to sets of constraints, $\mathfrak{S} : \omega \rightarrow \mathfrak{P}(\mathcal{C})$. where \mathfrak{P} denotes “power set”. For any integer n we write $\mathfrak{S} \oplus_n C$ for the following mapping:

$$(\mathfrak{S} \oplus_n C)(m) = \begin{cases} \mathfrak{S}(m) & \text{if } m \neq n \\ \mathfrak{S}(n) \cup \{C\} & \text{if } m = n. \end{cases}$$

If σ is clear from the context, we write $\mathfrak{S} \oplus_S C$ for $\mathfrak{S} \oplus_{\sigma(S)} C$.

We consider a branch of the computation to fail if *any* constraint store becomes inconsistent in that branch.

With these notations we can define the reduction rules for the disjunctive constraint-lambda calculus with multiple stores:

$$\begin{aligned} (M, \sigma, \mathfrak{S}) &\rightarrow (\perp, \sigma, \mathfrak{S}) \text{ if } \exists n \in \omega. \mathfrak{S}(n) = \otimes. & (\perp) \\ ((\lambda x.M)N, \sigma, \mathfrak{S}) &\rightarrow (M[x/N], \sigma, \mathfrak{S}). & (\beta) \\ (C, \sigma, \mathfrak{S}) &\rightarrow (n(C), \sigma, \mathfrak{S}), \text{ if } C \text{ is a pure constraint \& } C \neq n(C). & (\text{CR}) \\ (\{C\}^S M, \sigma, \mathfrak{S}) &\rightarrow (M, \sigma, \mathfrak{S} \oplus_S C), \text{ if } C \text{ is a pure constraint.} & (\text{CI}) \\ (X^S, \sigma, \mathfrak{S}) &\rightarrow (M, \sigma, \mathfrak{S} \oplus_S (X = M)), \text{ if } \mathfrak{S} \oplus_S (X = M) \neq \otimes. & (\text{CS}) \end{aligned}$$

The closure rules can be transferred *mutatis mutandis* from the disjunctive constraint-lambda calculus. We allow reductions in locations: If $(M, \sigma, \mathfrak{S}) \rightarrow (M', \sigma, \mathfrak{S}')$, then $(L^M, \sigma, \mathfrak{S}) \rightarrow (L^{M'}, \sigma, \mathfrak{S}')$, and similarly for $\{C\}^M N$.

Next we show how the addition of multiple stores adds even more flexibility.

Example 4. On p. 5, we argued that when we substitute M for X using the rule (CS) we have to add the constraint $X = M$ to the store to avoid substitutions such as those in Example 1. With the addition of multiple stores we have more liberty to define whether we want to allow this kind of behaviour. To illustrate this we slightly modify the example.

We define the abbreviation $M|N$ by: $M|N := \lambda x_S. \{X = M \vee X = N\}^{x_S} X^{x_S}$ where X is a fresh constraint variable. This term can be applied to a store name and evaluates to either M or N . For example, if we write \mathfrak{S}_0 for the map $n \mapsto \emptyset$, and if σ is any map $\mathcal{N} \rightarrow \omega$, then we obtain the following reductions:

$$\begin{aligned} ((2|3)S, \sigma, \mathfrak{S}_0) &\rightarrow (\{X = 2 \vee X = 3\}^S X^S, \sigma, \mathfrak{S}_0) \\ &\rightarrow (X^S, \sigma, \mathfrak{S}_0 \oplus_S \{X = 2 \vee X = 3\}) \\ &\rightarrow (2, \sigma, \mathfrak{S}_0 \oplus_S \{(X = 2 \vee X = 3), X = 2\}) \\ \text{and } ((2|3)S, \sigma, \mathfrak{S}_0) &\rightarrow (\{X = 2 \vee X = 3\}^S X^S, \sigma, \mathfrak{S}_0) \\ &\rightarrow (X^S, \sigma, \mathfrak{S}_0 \oplus_S \{X = 2 \vee X = 3\}) \\ &\rightarrow (3, \sigma, \mathfrak{S}_0 \oplus_S \{(X = 2 \vee X = 3), X = 3\}). \end{aligned}$$

Now consider a more complicated expression (corresponding to Example 2): $(\lambda x.x^{S_1}+x^{S_2})(2|3)$. If we evaluate this expression with a map σ for which $\sigma(S_1) = \sigma(S_2)$ it is obvious that this expression only evaluates to the values 4 and 6. If we change σ to a map where $\sigma(S_1) \neq \sigma(S_2)$ we obtain the three values 4, 5 and 6. In general this is not the desired behavior for arithmetic problems, but for other problems this behavior is more sensible. For example, if we allow constraints to range over job-titles in an organization, then it might be reasonable for a function $talkTo(programmer|manager)$ to talk to the manager in the part dealing with business matters and to the programmer when deciding technical details.

Another example where the choice of different values for a single constraint variable is useful are compilers. One specific example is code generators: An optimizing compiler might have different code generators for the same intermediate-language expression; these code generators usually represent different trade-offs that can be made between compilation speed, execution speed, space and safety. For example, the `d2c` compiler can assign either a *speed-representation* or a *space-representation* to a class. The CMUCL Common Lisp compiler has different policies (`:fast`, `:safe`, `:fast-safe` and `:small`) with which an intermediate representation might be translated into machine code. In a compiler based on the constraint-lambda calculus the policy used for the translation of some intermediate code could be determined by a constraint solver. This constraint-solver might compute disjunctive solutions, e.g., the permissible policy values might be `:safe` and `:fast-safe`, but not `:small` and `:fast` because some constraint on the safety of the program part in question has to be satisfied. In this case it is obviously desirable if different instantiations of the “policy-variable” can be instantiated with different values: An innermost loop might be compiled with `:fast-safe` policy to attain the highest possible execution speed while user-interface code might be compiled with the `:safe` policy to reduce the size of the program.

9 Comparison with earlier work

In [13], Mandel and Cengarle provided a partial solution of the disjunction problem only. We have now provided mechanisms for resolving Hennessy’s problem (see Section 1) in both directions.

A current example for a constraint-functional language is *Alice* [16] which is based on a concurrent lambda calculus with futures, $\lambda(\text{fut})$ [15, 17]. The $\lambda(\text{fut})$ calculus is not directly concerned with integration of constraints but rather allows the integration of constraint solvers via general-purpose communication mechanisms. There are two major technical differences between $\lambda(\text{fut})$ and our work: the treatment of concurrency, and how far the order of evaluations is restricted.

In our constraint-lambda calculi we do not deal with concurrency in our formulations of the the reduction rules of the calculi, we use *reduction strategies* to specify parallel executions on the meta-level. $\lambda(\text{fut})$ incorporates an interleaving

semantics for concurrent execution of multiple threads directly in the reduction rules. This makes it possible to talk about communication between concurrently executing threads in $\lambda(\text{fut})$ but not in the basic constraint-lambda calculi. In [6] we have developed an extension of the constraint-lambda calculi that can model explicit communication with the environment.

The $\lambda(\text{fut})$ calculus uses the call-by-value β -reduction rule, which requires all arguments to functions to be evaluated before the function can be applied. Furthermore, to preserve confluence, futures may only be evaluated at precisely specified points of a reduction sequence. The constraint-lambda calculi do not restrict applications of the β -rule at all and in general impose very few restrictions on allowed reductions.

10 Conclusions and future work

We have extended constraint functional programming to accommodate disjunctions. In particular we have introduced the unrestricted disjunctive constraint-lambda calculus and the restricted disjunctive constraint-lambda calculus in a simple and transparent fashion which, unlike previous attempts at defining combinations of constraint solvers and lambda calculi, makes them conservative extensions of the corresponding traditional lambda calculi.

The interface between the constraint store and the lambda terms ensures clarity and the smooth movement of information into and out of the constraint store.

We have shown that the restricted disjunctive constraint-lambda calculus satisfies a restricted form of confluence, namely that it is path-confluent as a reduction system. In the case of the the unrestricted disjunctive constraint-lambda calculus the stores play an important rôle and we can prove convergence of the terms only under certain conditions on the stores (Theorem 1).

In addition, we have given the denotational semantics for each of these theories.

Finally, we have shown how both horns of Hennessy's dilemma: e.g., the evaluation of $(\lambda x.x + x)(2|3)$ to either $\{4, 6\}$ or $\{4, 5, 6\}$, can be accommodated by the appropriate choice of one of our calculi.

In the future we are planning to extend our implementation of the constraint lambda calculi without disjunction (see [6, 7]) to the disjunctive constraint lambda calculi treated here.

References

1. Hendrik Pieter Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North Holland, 1995.
2. Nicolas Bourbaki. *Commutative Algebra, Chapters 1-7*. Elements of Mathematics. Springer, 1989, first published 1972.
3. Bart Demoen, María García de la Banda, Warren Harvey, Kim Marriott, and Peter Stuckey. An overview of *HAL*. In *Proceedings of Principles and Practice*

- of *Constraint Programming*, pages 174–188. Association for Computing Machinery, 1999.
4. Kees Doets. *From Logic to Logic Programming*. The MIT Press, 1994.
 5. Matthew C. B. Hennessy. The semantics of call-by-value and call-by-name in a nondeterministic environment. *SIAM Journal on Computing*, 9(1):67–84, 1980.
 6. Matthias Hözl. *Constraint-Lambda Calculi: Theory and Applications*. PhD thesis, Ludwig-Maximilians-Universität, München, 2001.
 7. Matthias M. Hözl and John Newsome Crossley. Parametric search in constraint-functional languages. In preparation.
 8. Matthias M. Hözl and John Newsome Crossley. Constraint-lambda calculi. In Alessandro Armando, editor, *Frontiers of Combining Systems, 4th International Workshop*, LNAI 2309, pages 207–221. Springer, 2002.
 9. Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Conference Record, 14th Annual ACM Symposium on Principles of Programming Languages, Munich, West Germany, 21–23 Jan 1987*, pages 111–119. Association for Computing Machinery, 1987.
 10. John Wylie Lloyd. *Foundations of Logic Programming*. Artificial Intelligence. Springer, second edition, 1987. First edition, 1984.
 11. Luis Mandel. *Constrained Lambda Calculus*. PhD thesis, Ludwig-Maximilians-Universität, München, 1995.
 12. Luis Mandel and María Victoria Cengarle. The disjunctive constrained lambda calculus. In Dines Bjørner, Manfred Broy, and Igor Vasilevich Pottosin, editors, *Perspectives of Systems Informatics, (2nd. International Andrei Ershov Memorial Conference, Proceedings)*, volume 1181 of *Lecture Notes in Computer Science*, pages 297–309. Springer Verlag, 1996.
 13. Luis Mandel and María Victoria Cengarle. The disjunctive constrained lambda calculus. In Dines Bjørner, Manfred Broy, and Igor Vasilevich Pottosin, editors, *Perspectives of Systems Informatics, (2nd. International Andrei Ershov Memorial Conference, Proceedings)*, volume 1181 of *Lecture Notes in Computer Science*, pages 297–309. Springer Verlag, 1996.
 14. Kim Marriott and Martin Odersky. A confluent calculus for concurrent constraint programming. *Theoretical Computer Science*, 173(1):209–233, 1997.
 15. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers in Combining Systems*, Lecture Notes in Computer Science. Springer, May 2005. Accepted for publication.
 16. Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklau, and Gert Smolka. Alice through the looking glass. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming, Volume 5*, volume 5 of *Trends in Functional Programming*. Intellect, Munich, Germany, 2004.
 17. Jan Schwinghammer. A concurrent lambda-calculus with promises and futures. Master’s thesis, Programming Systems Lab, Universität des Saarlandes, February 2002.
 18. Michael B. Smyth. Topology. In *Handbook of Logic in Computer Science*, pages 641–761. Oxford Science Publications, 1992.