A Tiny Constraint Functional Logic Language and Its Continuation Semantics

Andy Mück, Thomas Streicher

Ludwig-Maximilians-Universität München
Leopoldstr. 11B, D-80802 München
{mueck,streiche}@informatik.uni-muenchen.de

Hendrik C.R. Lock

IBM Scientific Centre Heidelberg
Vangerowstr. 18, D-69020 Heidelberg
lock@dhdibm1.bitnet

Abstract: We present an extension of λ-calculus by logical features and constraints, which yields a minimal core language for constraint functional logic programming. We define a denotational semantics based on continuation passing style. The operational semantics of our language is given as a set of reduction rules. We prove soundness of the operational semantics w.r.t. the continuation semantics. Finally, we show how pure functional logic programs can be translated to this core language in a sound way.

1 Introduction

In recent years, many research activities focused on the combination of functional and logic programming (FLP) [3, 12, 17, 19, 22, 23, 24, 28] as well as on the combination of constraint and logic programming (CLP) [5, 6, 11, 14, 15, 16]. Similar to FLP, where the motivation is to enrich logic programming languages with functions, the motivation of our work is to combine the essential features of the constraint, functional and logic programming paradigm. For that reason, we extend λ-calculus [2] by logical features and constraints to a constraint functional logic core language (CFLP-L). We will give a continuation semantics for this language and show soundness of the operational semantics w.r.t. the continuation semantics.

The notion of constraint functional logic programming (CFLP) has been introduced by Darlington et al. [8] and further has been investigated by Lopez-Fraguas [20]. Since [8] does not even allow to use constraints in function definitions, it could not be considered as full CFLP. The work of Lopez-Fraguas starts with a lazy functional logic programming language and extends it by a constraint structure and so called "constraint conditional rewrite rules". On denotational level, a model theoretic semantics is given. Additionally, an operational semantics is presented, which is based on narrowing. A soundness result for the operational semantics w.r.t. the declarative semantics is proved.

Our approach differs from [8] and [20]. Instead of extending a functional logic language by constraints to a full blown language, we capture the essential features of each single paradigm (constraint, functional and logic concepts) by minimal extensions of λ-calculus. These extensions are: *logical variables, non-deterministic choice, and constraints*.

Existentially quantified variables (*logical variables*) are an essential concept in logic programming. In combination with λ-calculus it is necessary to introduce an existential quantifier to declare the scope of a logical variable. Similar to logic programming a logical variable denotes the set of all its ground instances.

Non-deterministic choice between clauses is another basic concept of logic programming. We extend λ-calculus by a choice operator in order to make non-determinism explicit. Choice operators also have been studied before in connection with concurrent λ-calculi ([1, 4, 13]). However, the results of that work are less relevant for our studies, because concurrency deals with "don't care" non-determinism, whereas in a logical setting we are interested in all the choices, i.e. we are concerned with "don't know" non-determinism (compare [24]).

Unification, an operational concept in logic programming, is not present explicitly in our core language. We follow the idea of CLP(X) [15], which is replacing unification by constraint solving.

Similar to CLP(X), CFLP-L is parameterized by a *constraint theory*. In our approach, the constraint theory is considered as the initial model of an algebraic specification. Solving equations within this model is, on an abstract level, constraint solving. Furthermore, using algebraic specifications to describe a constraint theory has the nice side effect that, from an implementation point of view, the specification directly serves as a requirement specification for the design of the constraint solver.

As in a functional logic setting computations mostly do not terminate with a single value, but with a (possibly infinite) set of values, the denotational semantics of logical extensions of λ-calculus is usually based on powerdomains. Paterson [24] has defined a logical extension of λ-calculus with a semantics based on powerdomains. Although we have defined a powerdomain semantics for our language, too, we propose a continuation semantics of our language. First, continuations give us a useful insight to the operational behaviour. Second, continuation semantics enables later extensions by control operators [10]. The operational semantics of our language is given as set of reduction rules, where β-reduction plays the key role. Soundness of the operational semantics w.r.t. the continuation semantics is proved.

Similar to Lopez-Fraguas, we show that pure functional logic programming can be modelled within our framework. Therefore, we translate the rewrite rules of a functional logic program to CFLP-L expressions. The free constructor term algebra extracted from the functional logic program serves as the required constraint theory. Unification, then, is constraint solving in this term algebra.

Summarizing, we combine the essential features (λ-calculus, logical variables, choice operator, and constraints) of three programming paradigms (functional, logic and constraint programming) in a tiny core language. Similar to λ-calculus, which serves as *the* central core of functional languages, CFLP-L is proposed as a core of constraint functional logic languages.

The paper is organized as follows. In the next section we give the basic definitions of algebraic specifications. In section 3 we define syntax of CFLP-L. In section 4 we give a continuation semantics for CFLP-L. The operational semantics together with a soundness theorem are presented in section 5. In section 6 we show how pure functional logic programs can be translated to CFLP-L. Finally, we discuss related work.

2 Algebraic Specification of a Constraint Theory

As noted in the introduction, our language should be independent from a certain constraint theory. In order to instantiate CFLP-L with a constraint theory, we need a method to describe such theories. Since constraint theories can be considered as Σ-algebras, they can syntactically be represented by algebraic specifications.

In this section, we only give the basic definitions concerned with algebraic specifications. The reader interested in more details is referred to [27].

Definition 2.1 (Σ-Formulas)

Let X be a set of variables, $\Sigma=\{S,F\}$ be a signature with a set of sorts S and a set of function symbols F together with their arity. Let $T_\Sigma(X)$ be the set of well-formed terms over Σ and X. The set of *well-formed formulas WFFS(X)* over Σ and X is inductively defined as follows:

i) if $t,s \in T_\Sigma(X)$ then $(t=s) \in WFF_\Sigma(X)$
ii) if $\Phi \in WFF_\Sigma(X)$ then $\neg\Phi \in WFF_\Sigma(X)$
iii) if $\Phi_1,\Phi_2 \in WFF_\Sigma(X)$ then $\Phi_1 \wedge \Phi_2 \in WFF_\Sigma(X)$
iv) if $\Phi \in WFF_\Sigma(X)$ then $\forall x.\Phi \in WFF_\Sigma(X)$

A formula is called *closed* if it does not contain free variables.

Definition 2.2 (Algebraic Specification)

An *algebraic specification* $SP=(\Sigma,AX)$ consists of a signature Σ and a set of closed formulas AX over Σ. The formulas AX are called *axioms* of the specification.

Definition 2.3 (Model of an Algebraic Specification)

Let $SP=(\Sigma,AX)$. A partial Σ-algebra A is called *model* of SP, if A is term generated and if A satisfies the axioms of SP, i.e. $\forall\Phi \in AX. A \models \Phi$.
Mod(SP) denotes the set of all models of a specification SP.
Let $t \in T_\Sigma(X)$ and $A \in Mod(SP)$ then $t^{A,e}$ denotes the *interpretation* of t in A under a valuation (environment) e.
Let $f{:}s_1 \times .. \times s_n \rightarrow s_{n+1} \in F$. Then f^A denotes the *interpretation* of f in $A \in Mod(SP)$, i.e. f^A is a partial and strict function in $\mathcal{A}_1 \times .. \times \mathcal{A}_n \rightarrow \mathcal{A}_{n+1}$, where $\mathcal{A}_1,..,\mathcal{A}_{n+1}$ are the *carrier sets* of the sorts $s_1,..,s_n$.

We restrict attention to term generated models, because computing in non term generated models is not possible since their elements cannot be represented. In fact, the term "CLP($\mathbb{R}$)" is misleading because a constraint solver will calculate with representable approximations of real numbers but never will calculate with "real" real numbers.

Definition 2.4 (Set of Constructor Symbols)

Let $SP=(\Sigma,AX)$, $\Sigma=(S,F)$ be an algebraic specification. A subset $D \subseteq F$ is called a *set of constructor symbols*, if each element in each model of SP can be represented by a unique term consisting of constructors only, i.e. $\forall A \in Mod(SP)\ \forall a \in \mathcal{A}\ \exists! t \in T_D(\emptyset)$ such that $t^A=a$. Note that $T_D(\emptyset)$ denotes the set of all ground terms built with elements from D only (*ground constructor terms*).

Definition 2.5 (Initial Model of an Algebraic Specification)

Let $SP=(\Sigma,AX)$ be a specification. An algebra $A\in Mod(SP)$ is called *initial model* of SP, if

$\forall B\in Mod(SP)$ there exists a unique Σ-homomorphism $h:A\to B$.

We say that a specification SP is a *constraint specification* if SP has an initial model. The initial model itself is called *constraint theory.*

3 Syntax of CFLP-L

To define syntax of CFLP-L, we extend the usual formation rules of λ-calculus in order to incorporate logical variables, a choice operator and constraints.

Definition 3.1 (Syntax of CFLP-L)

Let $SP=(\Sigma,AX)$ be a constraint specification with $\Sigma=(S,F)$. Let $D\subseteq F$ be a set of constructors. The syntax of CFLP-L is given as follows.

$V\in$Value Expressions $\equiv x \mid \lambda x.E \mid \mu g,x.E \mid c(V,...,V)$	where $c\in D$
$E\in$Expressions $\equiv V \mid EE \mid E\Box E \mid f(E,...,E) \mid \exists x.E \mid \{C\}E$	where $f\in F$
$C\in$Constraints $\equiv T=T \mid C,C$	
$T\in$Terms $\equiv V \mid f(T,...,T)$	where $f\in F$

Due to semantic reasons we distinguish between value-expressions and expressions. Value expressions denote expressions which are in *weak head normal form* (WHNF). Note that the restriction to term generated models of the constraint specification is necessary in order to represent each element of the constraint theory as a weak head normal form. We restrict constraints to be equations on terms. Note that predicates, if needed, may be defined as boolean valued functions within the constraint specification.

Rather than simulating recursion by the Y-combinator, we explicitly use the μ-operator to define recursive functions. Roughly speaking, $\mu g,x.M$ corresponds to an SML-like function definition fun g x = M.

The operator $\Box$ is used to express a "don't know" non-deterministic choice between two expressions. In an expression $\exists x.M$ the abstraction operator $\exists$ is used to declare a logical variable x with scope M. The intuitive meaning of $\exists x.M$ is the set of all $M[a/x]$ such that $a\in\mathcal{A}$, where $\mathcal{A}$ is the carrier set of initial model A of SP. Accordingly, the meaning of $M\Box N$ is the union of the meanings of M and N. Both operators set up the logic programming concepts of CFLP-L.

In order to restrict an expression M by a certain constraint C we use the notation $\{C\}M$. The meaning of $\{C\}M$ under a valuation e is equal to the meaning of M under e, if C holds under e. Otherwise the meaning of $\{C\}M$ is the empty set of results.

Note that the usual definitions of free variables and closed expression easily can be extended for our language. In the following we feel free to use these notations without explicitly defining them.

Example 3.1

Let SP be a specification of natural numbers with multiplication. The expression $\lambda x.(\exists y.\{x=2*y\}x)$ denotes a function that returns $\{x\}$ if x is even or returns the empty set ($\emptyset$) of results if x is odd.
Applying the function to a choice between 1,2,3,4 or 5

$(\lambda x.(\exists y.\{x=2*y\}x))\ (1 ▯ 2 ▯ 3 ▯ 4 ▯ 5)$

returns $\{2,4\}$.
Of course, applying the function to a higher order expression

$(\lambda x.(\exists y.\{x=2*y\}x))\ (\lambda z.z)$

returns $\emptyset$, because $\lambda z.z$ is not element of the constraint theory (*here:* $\mathbb{N}$).

(Note that within $\mathbb{N}$ as constraint theory, the intuitive meaning of the expression $\exists x.M$ is $\{M[a/x] \mid a\in\mathbb{N}\}$.)

4 A Continuation Semantics for CFLP-L

Lafont, Reus and Streicher [18] have shown that for functional languages continuation semantics [10] is useful to deal with operational aspects on a denotational level. Using continuation semantics has also the advantage that later control operators can be included easily.

Let SP=(Σ,AX) be a constraint specification, Σ=(S,F). For simplicity, we suppose that S={s} consists of a single sort. Let $\mathcal{A}$ be the carrier set of the initial model A of SP.

As example 3.1 shows, *results* of a computation in a functional logic setting are sets over $\mathcal{A}$. Therefore, the semantic domain $\mathcal{R}$ of results (objects of interest) is given by

$$\mathcal{R} = \mathcal{A} \rightarrow \mathcal{O}$$

where $\mathcal{O}$ is the Sierpinski-space (i.e. the two-element lattice consisting of $\top$ and $\bot$). Since $\mathcal{A}$ is discrete, $\mathcal{A} \rightarrow \mathcal{O}$ is isomorphic to $(\mathcal{P}(\mathcal{A}),\subseteq)$, where $\mathcal{P}(\mathcal{A})$ is the powerset of $\mathcal{A}$. Therefore, we feel free to use set notation in the context of $\mathcal{R}$. Let in the following $\emptyset$ denote the result $\lambda a.\bot$.

Values in our computation are either elements of $\mathcal{A}$ or mappings from values and continuations to results, where *continuations* are mappings from values to results. Thus, the semantic domain $\mathcal{V}$ of values and the domain $\mathcal{C}$ of continuations are given by the following equations.

$$\mathcal{V} = \mathcal{A} + (\mathcal{V} \rightarrow \mathcal{C} \rightarrow \mathcal{R})$$

$$\mathcal{C} = \mathcal{V} \rightarrow \mathcal{R}$$

Environments are mappings from variables to values.

$$\mathrm{Env} = \mathrm{Var} \rightarrow \mathcal{V}$$

In the following, let

"$f^A(v_1,..,v_n)\downarrow$"

abbreviate

"$v_1,..,v_n \in A$ and there exists $a \in A$ such that $a =^A f^A(v_1,..,v_n)$"

The semantics of value-expressions is given by a function mapping value-expressions and environments to $\mathcal{V}_\perp$.

$$[\![\_]\!]_V : V \to \mathrm{Env} \to \mathcal{V}_\perp$$

$$[\![x]\!]_V\, e = e(x)$$

$$[\![\lambda x.M]\!]_V\, e = \lambda v.[\![M]\!]_E\, e[v/x]$$

$$[\![\mu g,x.M]\!]_V\, e = \mathrm{fix}_{\mathcal{V} \to \mathcal{C} \to \mathcal{R}}(\lambda h.\lambda v.[\![M]\!]_E\, e[h/g,v/x]),$$

where $\mathrm{fix}_{\mathcal{V} \to \mathcal{C} \to \mathcal{R}}$ (F) is the least fixpoint of F.

$$[\![c(v_1,..v_n)]\!]_V\, e = \begin{cases} c^A([\![v_1]\!]_V\, e,..,[\![v_n]\!]_V\, e), \text{ if } c^A([\![v_1]\!]_V\, e,..,[\![v_n]\!]_V\, e)\downarrow \\ \perp, \text{ otherwise} \end{cases}$$

The semantics of expressions is a function $[\![\_]\!]_E : E \to \mathrm{Env} \to \mathcal{C} \to \mathcal{R}$.

$$[\![\_]\!]_E : E \to \mathrm{Env} \to \mathcal{C} \to \mathcal{R}$$

$$[\![v]\!]_E\, e\, k = \begin{cases} k([\![v]\!]_V\, e), \text{ if } [\![v]\!]_V\, e \in \mathcal{V} \\ \emptyset, \text{ otherwise (note: } [\![v]\!]_V\, e \text{ might be } \perp) \end{cases}$$

$$[\![MN]\!]_E\, e\, k = [\![M]\!]_E\, e\, \lambda m.(\, [\![N]\!]_E\, e\, \lambda n.(m\, n\, k)\,)$$

$$[\![f(M_1,..,M_n)]\!]_E\, e\, k =$$
$$[\![M_1]\!]_E\, e\ \lambda v_1.\, (\, [\![M_2]\!]_E\, e\, \lambda v_2.\, ...\, (\, [\![M_n]\!]_E\, e$$
$$\lambda v_n.\, \lambda a.\text{ if } f^A(v_1,..,v_n)\downarrow \text{ then } k(f^A(v_1,..,v_n))\, a\,)\, ...\,)$$

$$[\![\exists x.M]\!]_E\, e\, k = \bigcup_{a \in \mathcal{A}} [\![M]\!]_E\, e[a/x]\, k$$

$$[\![M \| N]\!]_E\, e\, k = [\![M]\!]_E\, e\, k \cup [\![N]\!]_E\, e\, k$$

$$[\![\{C\}M]\!]_E\, e\, k\, a = [\![C]\!]_C\, e \wedge [\![M]\!]_E\, e\, k\, a \qquad (\wedge \textit{ in Sierpinski-space})$$

The semantics of values and applications is defined analogously to a continuation semantics of functional languages.

In $f(M_1,..,M_n)$ the evaluation order is from left to right. The continuation of the last argument first checks, whether $f^A(v_1,..,v_n)$ is defined in A and then continues with $f^A(v_1,..,v_n)$. If $f^A(v_1,..,v_n)$ is not defined in A then we abort computation. In such a case $\lambda a.\perp$ ($\emptyset$ resp.) is the result. This strategy corresponds to a call-by-value evaluation strategy.

In the context of a logical variable x, the semantics of an expression M is defined as the union of all $a \in \mathcal{A}$ of the semantics of M under the environment extension [a/x]. The semantics of the non-deterministic choice simply is the union of the results of the single expressions.

A constraint expression {C}M either returns $\emptyset$ if the constraints C are not satisfied, or it returns the results of M. Note that the above semantic equation for {C}M is equivalent to the following one:

$$[\![\{C\}M]\!]_E\, e\, k = \begin{cases} [\![M]\!]_E\, e\, k, \text{ if } [\![C]\!]_C\, e = \top \\ \emptyset, \text{ otherwise} \end{cases}$$

The semantic equations for $\exists x.M$ and $\{C\}M$ both reflect the essence of CLP, i.e. the computation of valuations of logical variables that satisfy the constraints, and the evaluation only of those expressions which satisfy their constraints.
The semantics of constraints is given by a function mapping constraints to $\mathcal{O}$.

$$[\![\_]\!]_C : C \to Env \to \mathcal{O}$$

$$[\![c_1,..,c_n]\!]_C\ e = [\![c_1]\!]_{EQ}\ e \wedge ... \wedge [\![c_n]\!]_{EQ}\ e$$

A sequence of constraints must be interpreted as the conjunction of all the single constraints. An equation s=t holds under an environment e, if the interpretations of s and t under e are defined and equal elements in A. For equality we restrict attention to elements in $\mathcal{A}$, because equality on non-flat predomains is not even monotonic. Therefore, an equation t=s returns $\perp$ if either t or s is a higher order term (compare example 3.1).

$$[\![\_]\!]_{EQ} : (T \times T) \to Env \to \mathcal{O}$$

$$[\![t=s]\!]_{EQ}\ e = \begin{cases} \top, \text{ if } [\![t]\!]_T\ e\ ,\ [\![s]\!]_T\ e \in \mathcal{A} \text{ and } [\![t]\!]_T\ e\ =^A [\![s]\!]_T\ e \\ \perp, \text{ otherwise} \end{cases}$$

$$[\![\_]\!]_T : T \to Env \to \mathcal{A}_\perp$$

$$[\![v]\!]_T\ e = \begin{cases} [\![v]\!]_V\ e, \text{ if } [\![v]\!]_V\ e \in \mathcal{A} \\ \perp, \text{ otherwise} \end{cases}$$

$$[\![f(t_1,..,t_n)]\!]_T\ e = \begin{cases} f^A([\![t_1]\!]_T\ e,..,[\![t_n]\!]_T\ e), \text{ if } f^A([\![t_1]\!]_T\ e,..,[\![t_n]\!]_T\ e)\downarrow \\ \perp, \text{ otherwise} \end{cases}$$

Note that each computation starts with the empty environment e_0 and the continuation stop$\equiv\lambda v.\lambda a.(a=v)$ mapping a value v either to the singleton set $\{v\}$ if $v \in \mathcal{A}$ or to $\emptyset$ otherwise.

Example 4.1

An example often presented in non-deterministic λ-calculi (compare [24]) is the following one:

$(\lambda x.x+x)\ (2\ [\!]\ 3)$

The question is whether this expression is equal to $\{4,5,6\}$ or only to $\{4,6\}$. Here, the essential operational issue is whether or not the choice transported by argument x is shared in the expression x+x. From a denotional point of view, it is a question whether λ-abstractions denote mappings from single values to a set of results or whether λ-abstractions denote mappings from a set of values to a set of results. Since in our approach λ-abstractions denote mappings from single values to a set of results we obtain

$[\![(\lambda x.x+x)\ (2\ [\!]\ 3)]\!]_E\ e_0\ stop = [\![(\lambda x.x+x)]\!]_E\ e_0\ \lambda m.(\ [\![(2\ [\!]\ 3)]\!]_E\ e_0\ \lambda n.(m\ n\ stop)\ ) =$
$[\![(2\ [\!]\ 3)]\!]_E\ e_0\ \lambda n.(([\![(\lambda x.x+x)]\!]_V\ e_0)\ n\ stop) = [\![2]\!]_E\ e_0\ \lambda n.([\![x+x]\!]_E\ e_0[n/x]\ stop) \cup$
$[\![3]\!]_E\ e_0\ \lambda n.([\![x+x]\!]_E\ e_0[n/x]\ stop) =$
$([\![x+x]\!]_E\ e_0[2/x]\ stop) \cup ([\![x+x]\!]_E\ e_0[3/x]\ stop) = \{4,6\}.$

5 Operational Semantics for CFLP-L

In the following, we give reduction rules which define the operational semantics of our language. These rules can be split into 4 groups: reduction rules, constraint rules, structural rules and distribution rules.

Let SP=(Σ,AX) be a constraint specification, Σ=(S,F). Let f∈F. Let D⊆F be a set of constructor symbols. Let A be the initial model of SP with carrier set $\mathcal{A}$. Let $v,v_1,..,v_n$ denote value-expressions (expressions in WHNF, see section 3), M,M_1,M_2 and N denote expressions and let C,C' denote constraints. The usual conventions of λ-calculus to avoid unintended bindings of free variables are assumed to be granted within the following rules.

Reduction Rules

$(\lambda x.M)\ v \rightarrow M[x{:=}v]$ (*note: v is a WHNF*)

$\mu g,x.M \rightarrow \lambda x.M[(\mu g,x.M)/g]$

$f(v_1,..,v_n) \rightarrow t$, if $f(v_1,...,v_n)=t$ holds in A for some $t\in T_D(X)$

Since the continuation semantics is based on a call-by-value strategy, β-reduction applies only if the argument is evaluated to WHNF (β_{value}-reduction). The reduction rule for defined function symbols hides a call of the constraint solver, because only the constraint theory tells us how to reduce such a function application. Unfolding the μ-operator is straight forward to its treatment in a pure functional approach.

Since on an operational level neither we have the possibility to express the semantics of ∃x.M by a union over all elements of A (as in the continuation semantics) nor is such a (possibly infinite) union desirable, we describe the operational behaviour of ∃ in connection with a constraint expression.

Constraint Rules

Constraint Modification:

$\exists \bar{x}.\{C\}M \rightarrow \exists \bar{x}.\{C'\}M$, if $A\models C \Leftrightarrow A\models C'$ and $FV(\{C'\}M)\subseteq FV(\{C\}M)=\{x_1,...,x_n\}$

where $\exists \bar{x}$ abbreviates $\exists x_1 ..\exists x_n$.

Constraint Propagation:

$\exists x.\{C,x{=}t,C'\}M \rightarrow (\{C,C'\}M)[t/x]$, if $t\in T_\Sigma(X)$ and $x\notin VAR(t)$

Analogous to the continuation semantics, the operational semantics of our language should be independent from the constraint theory. Therefore, the constraint modification rule only describes the *essential requirement* for any constraint solver, i.e. modification of constraints in a sound and complete way. For an instantiation of our language with a certain constraint theory, this rule may be replaced by the reduction rules of a certain constraint solver (see section 6 for an example).

A constraint solver can solve constraints over its specific constraint domain only (compare example 3.1). In our approach this constraint domain is given by the constraint specification. For that reason, constraint propagation only applies if $t\in T_\Sigma(X)$ (see above). Similar constraint rules can be found in [7]. A similar propagation rule has been proposed in [19].

However, our operational semantics would even work without any propagation rule. In such a case, the reduction rules in connection with the structural and distribution rules (see below) will compute *elementary problems* of the form

$$\exists\, x_1. \dots \exists\, x_n.\{C\}t, \text{ where } t \in T_D(X).$$

which finally can be handled by the constraint solver.

The reader might be surprised that there are no failure-rules which prune computation paths with inconsistent constraints. Failure-rules would require an additional failure-constant in CFLP-L syntax. Instead of blowing up syntax and operational semantics of CFLP-L with such rules, we consider failure-rules as part of an implementation rather than as a necessary evil on semantic level.

In order to prepare expressions either for β-reduction or for constraint propagation, we need the following structural rules and distribution rules.

Structural Rules

$\{C\}(\{D\}M) \to \{C,D\}M$	$(\exists x.M)N \to \exists x.(MN)$, if $x \notin FV(N)$
$(\{C\}M)N \to \{C\}(MN)$	$\{C\}(\exists x.M) \to \exists x.(\{C\}M)$, if $x \notin VAR(C)$
$v(\{C\}N) \to \{C\}(vN)$	
$f(\dots,\{C\}M,\dots) \to \{C\}f(\dots,M,\dots)$	$v(\exists x.M) \to \exists x.(vM)$, if $x \notin FV(v)$
	$\exists x.\exists y.M \to \exists y.\exists x.M$

$$f(M_1..,\exists x.M_i,..,M_n) \to \exists x.(f(M_1,..,M_i,..,M_n)), \text{ if } x \notin FV(M_1,..,M_n)$$

Distribution Rules

$\{C\}(M \Box N) \to \{C\}M \Box \{C\}N$	$v(N_1 \Box N_2) \to (vN_1) \Box (vN_2)$
$f(\dots,M_1 \Box M_2,\dots) \to f(\dots,M_1,\dots) \Box f(\dots,M_2,\dots)$	$\exists x.(M \Box N) \to (\exists x.M) \Box (\exists x.N)$
	$(M_1 \Box M_2)N \to (M_1N) \Box (M_2N)$

The distribution rules serve to distribute constraints, functions, abstractions and ∃ among choices. Within a single choice the structural rules move constraints and ∃ outwards. Both sets of rules interact to prepare expressions for β-reduction and constraint propagation.

Logic as well as functional logic languages usually are implemented upon several abstract machines ([12,17,19,23,26]). These abstract machines either use sophisticated graph-based [17, 19] or heap-based [12, 23, 26] term representations which have the advantage that state transitions corresponding to our structural rules can be omitted. But, the price which must be paid for saving structural rules is a complicated data structure to represent terms. Therefore, in our operational semantics we use structural rules in order to abstract from such implementation details.

Our distribution rules are closely related to choice point concepts know from implementation of logic and functional logic languages. For example, in a functional logic language, application of a function f to a choice $M_1 \Box (M_2 \Box M_3)$ stores $f(M_2 \Box M_3)$ in a choice point and executes $f(M_1)$. In our operational semantics, $f(M_1 \Box (M_2 \Box M_3))$ reduces to $f(M_1) \Box f(M_2 \Box M_3)$. Roughly speaking, the first term of a choice expression can therefore be considered as actual computation, where the second term is a choice point.

Example 5.1

From an operational point of view, the question of the example 4.1 is whether or not choice expressions may be bound to a shared formal parameter.

Since in our operational semantics we have β_{value}-reduction (i.e. β-reduction applies to value expressions only) it is not possible that choice expressions are bound to formal parameters. The distribution rule

$v(N_1 \Box N_2) \rightarrow (vN_1) \Box (vN_2)$

serves to distribute an application among choices. Therefore, we obtain

$(\lambda x.x{+}x)\ (2 \Box 3) \rightarrow_{distr.} (\lambda x.x{+}x)\ 2 \Box (\lambda x.x{+}x)\ 3 \rightarrow_{\beta} 4 \Box 6.$

The following theorem states soundness of the operational semantics w.r.t. the continuation semantics of CFLP-L. I.e. if an expression M reduces via the operational semantics to an expression N, then in the continuation semantics M and N return the same set of results for an arbitrary environment e and for an arbitrary continuation k.

Theorem 5.1 (Soundness of the Operational Semantics)

The operational semantics is sound w.r.t. the continuation semantics.

I.e. if $M \rightarrow^* N$ then $\llbracket M \rrbracket_E\ e\ k = \llbracket N \rrbracket_E\ e\ k$, for any environment e and any continuation k, where $\rightarrow^*$ is the reflexive and transitive closure of reduction relation $\rightarrow$.

Proof: *It is sufficient to show that all rules of the operational semantics preserve the continuation semantics. We show the proof only for β-reduction and constraint propagation. A complete proof can be found in [21].*

β-Reduction:

$\llbracket(\lambda x.M)\ V\rrbracket_E\ e\ k = \llbracket\lambda x.M\rrbracket_E\ e\ \lambda m.(\llbracket V\rrbracket_E\ e\ \lambda n.(m\ n\ k)) =$

$\llbracket V\rrbracket_E\ e\ \lambda n.((\llbracket\lambda x.M\rrbracket_V\ e)\ n\ k) =$

$\llbracket V\rrbracket_E\ e\ \lambda n.(\llbracket M\rrbracket_E\ e[n/x]\ k) = \llbracket M\rrbracket_E\ e[(\llbracket V\rrbracket_V\ e)/x]\ k = \llbracket M[V/x]\rrbracket_E\ e\ k.$

Constraint Propagation Rule:

$\llbracket\exists\ x.\{C,c{=}t,C'\}M\rrbracket_E\ e\ k = \bigcup_{a\in A} \llbracket\{C,x{=}t,C'\}M\rrbracket_E\ e[a/x]\ k =$

$\bigcup_{a\in A} \lambda b.(\llbracket C,x{=}t,C'\rrbracket_C\ e[a/x] \wedge \llbracket M\rrbracket_E\ e[a/x]\ k\ b) =$

$\bigcup_{a\in A} \lambda b.(\quad \llbracket C\rrbracket_C\ e[a/x] \wedge \llbracket x{=}t\rrbracket_C\ e[a/x] \wedge \llbracket C'\rrbracket_C\ e[a/x] \wedge$
$\qquad\qquad \llbracket M\rrbracket_E\ e[a/x]\ k\ b) =$

(because $\llbracket x{=}t\rrbracket_C\ e[a/x]$ only holds if $x \notin VAR(t)$ and if a is equal to t^A)

$\lambda b.(\llbracket C\rrbracket_C\ e[t^A/x] \wedge \llbracket C'\rrbracket_C\ e[t^A/x] \wedge \llbracket M\rrbracket_E\ e[t^A/x]\ k\ b) =$

$\lambda b.(\llbracket C[t/x]\rrbracket_C\ e \wedge \llbracket C'[t/x]\rrbracket_C\ e \wedge \llbracket M[t/x]\rrbracket_E\ e\ k\ b) =$

$\llbracket(\{C,C'\}M)[t/x]\rrbracket_E\ e\ k.$

6 Pure Functional Logic Programming

In this section we will show how a certain instance of CFLP-L serves as a target language to translate pure functional logic programs.

Roughly speaking, a functional logic program consists of a signature Σ including a set D of free constructor symbols (i.e. constructor symbols not affected by the rewrite rules) and a set E of rewrite rules of the form

$f(t_1,..,t_n) \rightarrow r$

where f is a non-constructor function symbol, t_i ($1 \leqslant i \leqslant n$) are constructor terms and r is an arbitrary term over Σ.

Narrowing [9, 25] has been established as operational model for functional logic programs. In particular, narrowing tries to solve an equation eq by finding a rule $l \rightarrow r$, a non-variable subterm t of eq and a most general unifier σ such that $\sigma t = \sigma l$. Then, r is substituted for t in eq and $\sigma(eq[r/t])$ returns a new equation.

Unification is constraint solving of equations in a free term algebra. Hence, an instantiation of CFLP-L with the free constructor term algebra of a functional logic program P can serve as a target language to compile P.

As noted in section 5, in an instantiation of CFLP-L with a certain constraint theory, the constraint modification rule presented above may be replaced by the reduction rules of a certain constraint solver. For the purpose of unification, we replace this rule by the following *unification rules* which serves to reduce unification constraints.

$\{C, c(t_1,..,t_n) = c(s_1,..,s_n), C'\}M \rightarrow \{C, t_1=s_1,..,t_n=s_n, C'\}M$, c is a constructor symbol
$\exists\, x.\{C, x=x, C'\}M \rightarrow \exists\, x.(\{C,C'\}M)[t/x]$ *(note: x may occur in C, C' and M)*

Together with the constraint propagation rule (see section 5) the unification rules solve equations in the free constructor term algebra of a functional logic program P. I.e. these rules build a *specialized constraint solver* for unification.

Since detailed definition of a compiler is out of scope of this paper, let us demonstrate the compilation of a functional logic program P by means of a short example.

Example 6.1

The following functional logic program for natural numbers with addition

```
program    NAT
sort       Nat
cons       0 : Nat,
           succ : (Nat) Nat
func       add : (Nat,Nat) Nat
axioms
           add(0,y)→y,
           add(succ(x),y)→succ(add(x,y))
end.
```

translates to the CFLP-L expression

$$\mu\ add,(x,y)\ .\ \{x=0\}\ y\ \mathbin{[\!]}\ \exists z.\{x=succ(z)\}\ succ(add\ z\ y)$$

where the free term algebra over 0 and succ is considered as underlying constraint theory specified by the constructor signature without any axioms. (Note that we extended the μ-operator for n-ary tuples of arguments.)

The rules for addition are translated to a choice between two expressions. Applying the first axiom to a term add(m,n) requires unification of m with 0. Therefore, the translation the left hand side of this rule handles the constraint $\{x=0\}$ to the constraint solver. Analogously, application of the second rule requires unification of m with succ(z), where z is a new logical variable. For that reason, we translate the left hand side to the constraint $\{x=succ(z)\}$, where z is existentially quantified in order to express that z is a new logical variable. The translation of the right hand sides is straight forward.

As noted above, narrowing tries to solve an equation $t_1=t_2$ over some functional logic program P. In order to solve such an equation, CFLP-L will first evaluate t and s to WHNFs (value expressions), and then pass the equation between the corresponding WHNFs to the constraint solver. For that reason, an equation $t_1=t_2$ is translated to the following CFLP-L expression which, in connection with the translation of a program, essentially implements narrowing.

$$\exists\, x_1. \ldots \exists\, x_n.(\lambda t.\lambda s.(\{t=s\}t)\ t_1\ t_2), \text{ where } \{x_1,.., x_n\} = VAR(t_1=t_2).$$

(Please note that expression t behind the constraint $\{t=s\}$ occurs for syntactical reasons only. It is also possible to replace this expression by a new constant SUCCESS to be defined in the constraint specification.)

Example 6.2

Finally, we show the CFLP-L computation for solving the equation add(m,n)=2 until the first result (0/m,2/n) is computed. With

$$ADD \equiv \mu\ add,x,y\ .\ \{x=0\}\ y\ \square\ \exists z.\{x=succ(z)\}\ succ(add\ z\ y).$$

we obtain

$\exists\, m.\exists\, n.\lambda t.\lambda s.(\{t=s\}t)\ (ADD\ m\ n)\ 2 \rightarrow$ *(unfolding ADD)*

$\exists\, m.\exists\, n.\lambda t.\lambda s.(\{t=s\}t)\ (\{m=0\}\ n\ \square$
$\exists z.\{m=succ(z)\}\ succ(ADD\ z\ n))\ 2 \rightarrow_{(struct.\ rules)}$

$\exists\, m.\exists\, n.\lambda t.\lambda s.(\{t=s\}t)\ (\{m=0\}\ n)\ 2\ \square$
$\exists\, m.\exists\, n.\lambda t.\lambda s.(\{t=s\}t)\ (\exists z.\{m=succ(z)\}\ succ(ADD\ z\ n))\ 2 \rightarrow_{(struct.\ rules)}$

$\exists\, m.\exists\, n.\{m=0\}(\lambda t.\lambda s.\{t=s\}t)\ n\ 2\ \square$
$\exists\, m.\exists\, n.\exists z.\{m=succ(z)\}\ \lambda t.\lambda s.(\{t=s\}t)\ (succ(ADD\ z\ n))\ 2 \rightarrow_\beta$

$\exists\, m.\exists\, n.\{m=0\}(\{n=2\}n)\ \square$
$\exists\, m.\exists\, n.\exists z.\{m=succ(z)\}\ \lambda t.\lambda s.(\{t=s\}t)\ (succ(ADD\ z\ n))\ 2 \rightarrow_{(struct.\ rule)}$

$\exists\, m.\exists\, n.\{m=0,n=2\}n)\ \square$
$\exists\, m.\exists\, n.\exists z.\{m=succ(z)\}\ \lambda t.\lambda s.(\{t=s\}t)\ (succ(ADD\ z\ n))\ 2 \rightarrow \ldots$

7 Related Work and Conclusions

We strongly have been influenced by the work of Crossley, Mandel and Wirsing on constraint λ-calculus [7]. Constraint λ-calculus is an extension of untyped λ-calculus by constraints in order to provide a scheme for constraint functional programming similar to CLP(X). Therefore, the constraint solver is considered as a black box, containing a decidable and representable domain of constraints. Our original motivation was to implement functional logic languages by translating them to constraint λ-calculus where a first order unification theory should fill the black box. But we soon realized that for our special needs, constraint λ-calculus lacks important logic programming concepts, namely logical variables and non-deterministic choice. Therefore, we extended constraint λ-calculus by those concepts. It turned out that this extension not only is suitable to implement functional logic programming languages, but also serves as a new approach to constraint functional logic programming. However, extending constraint λ-calculus by logical variables changes its semantics essentially.

The work of Ross Paterson [24] was the motivation for a first semantic approach to our language. Paterson extended λ-calculus with logical features and gave a denotational semantics for his language, which is based on lower powerdomains. Rather than introducing logical variables, Paterson uses a syntactic constant $\mho$ denoting his semantic domain $\mathcal{D}\cong\mathcal{D}\rightarrow\mathcal{P}\mathcal{D}$. In a similar semantic treatment of our language it turned out that Paterson's $\mho$ can in our language be expressed by $\exists x.x$. Therefore, the existential quantifier is a more *"fine grain"* logical feature than $\mho$.

The continuation semantics of CFLP-L has been derived from a powerdomain semantics by restricting attention to observable objects, i.e. elements in the powerset of $\mathcal{A}$ (where $\mathcal{A}$ is the carrier set of the initial model of the constraint specification). Continuations have been helpful when defining the operational semantics of our language, because they allow to reason about operational questions on a denotational level [18]. On the other hand, a continuation semantics keeps our language open for later extensions by control operators.

The syntax of the functional logic core language presented in [19] is very similar, but richer because it has been designed for specifying compilation schemes for functional logic languages with different semantics. Furthermore, unification has not been treated as a separate constraint system.

As noted in the introduction, our approach differs from that of Lopez-Fraguas [20] because he extends a lazy functional logic language by constraints to a full blown language.

In this paper, we combined the *essential* features of functional, logic and constraint programming in a tiny core language. CFLP-L is intended to serve as a core of constraint functional logic programming languages. The continuation semantics presented is, as far as we know, a new approach to denotational semantics of constraint functional logic languages. We have demonstrated that narrowing can be expressed by β-reduction and constraint solving in a free term algebra.

Acknowledgements

We are very much indebted to John Crossley, Luis Mandel, and Martin Wirsing for their fruitful remarks, interesting ideas, and for many stimulating discussions. Without their work on constraint λ-calculus, this paper never would have been written.

Thanks to Rita Loogen, Michael Hanus, Heinrich Hußmann, and anonymous referees for valuable comments on a draft of this paper.

This work partially has been sponsored by the ESPRIT working group COMPASS.

References

[1] E. Astesiano, G. Costa: Sharing in Nondeterminism. In 6th International Conference on Automata, Languages and Programming, LNCS 71, Springer Verlag 1979.

[2] H.P. Barendregt. The Lambda Calculus. Its Syntax and Semantics. North Holland, 1984.

[3] P.G. Bosco, E. Giovanetti, G. Levi, C. Palamedessi: A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions. In *Proc. 4th Symposium on Logic Programming*, San Francisco, pp. 318-327. 1987.

[4] R.M. Burstall, J. Darlington: A Transformation System for Developing Recursive Programs. Journal of the ACM, 24(1), pp. 44-67, 1977.

[5] J. Cohen: Constraint Logic Programming Languages. In CACM 33 (7), pp. 52-68, 1990.

[6] A. Colmerauer: An Introduction to Prolog III. Communications of the ACM, 33 (7), pp. 52-68, 1990.

[7] J.N. Crossley, L. Mandel, M. Wirsing: Untyped Constraint Lambda Calculus is Weakly Church Rosser, Proc. NATO-ASI Constraint Programming Summer School, Pärnu, Estonia, 1993. An extended version of this paper appears as a Research Report, Ludwig-Maximilians-Universität München, 1993.

[8] J. Darlington, Y. Guo, H. Pull: A New Perspective in Integrating Functional and Logic Languages. Technical Report, Imperial College London, 1992.

[9] M.J. Fay: First-Order Unification in an Equational Theory. In Proc. 4th Workshop on Automated Deduction, pp. 161-167, Austin (Texas). Academic Press, 1979.

[10] M. Felleisen, D. Sitaram: Reasoning with Continuations II: Full Abstraction for Models of Control. In Proc ACM Conference in Lisp and Functional Programming, pp. 161-175, ACM Press, 1990.

[11] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, M. Wallace: Constraint Logic Programming - an Informal Introduction. Summer School in Logic Programming, Zürich, LNAI 636, pp. 3-35, Springer Verlag, 1992.

[12] M. Hanus: Efficient Implementation of Narrowing and Rewriting. In Proc. International Workshop on Processing Declarative Knowledge, pp. 344-365. LNAI 567, Springer Verlag, 1991.

[13] M.C.B. Hennessy, E.A. Ashcroft. A Mathematical Semantics for a Nondeterministic Typed λ-calculus. Theoretical Computer Science, 11, pp. 227-245, 1980.

[14] P. van Hentenryck: Constraint Satisfaction in Logic Programming. MIT Press, 1989.

[15] J. Jaffar, J.-L. Lassez: Constraint Logic Programming. In Proc. of the 14th ACM Symposium on Principles of Programming Languages, Munich (Germany), pp. 111-119. ACM Press, 1987.

[16] J. Jaffar, S. Michaylov, P. Stuckey, R. Yap: The CLP(R) Language and System. ACM Transactions on Programming Languages and Systems, pp. 339-395, 1992.

[17] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, M. Rodriguez-Artalejo: Graph-based Implementation of a Functional Logic Language. In Proc. ESOP 90, pp. 271-290. LNCS 432, Springer Verlag, 1990.

[18] Y. Lafont, B. Reus, Th. Streicher: From Continuation Semantics to Abstract Machines, unpublished manuscript, 1993.

[19] H.C.R.Lock: The Implementation of Functional Logic Programming Languages, Oldenbourg-Verlag 1993.

[20] F.J. Lopez-Fraguas: A General Scheme for Constraint Functional Logic Programming. In Proc. 3rd. International Conference on Algebraic and Logic Programming, Volterra (Italy), pp. 213-227. LNCS 632, Springer Verlag 1992.

[21] L. Mandel, A. Mück, Th. Streicher: A New Approach to Constraint Functional Logic Programming. Forthcoming Research Report, Ludwig-Maximilians-Universität München, 1993.

[22] D. Miller: A Logic Programming Language with Lambda Abstraction, Functional Variables and Simple Unification. Journal of Logic Programming, 1 (4), pp. 497-536, 1991.

[23] A. Mück: Compilation of Narrowing. In Proc. of the 2nd International Workshop on Programming Language Implementation and Logic Programming, pp. 16-29. LNCS 456, Springer Verlag, 1990.

[24] R. Paterson: A Tiny Functional Language with Logical Features, Declarative Programming, Sassbachwalden, Springer, 1991.

[25] U.S. Reddy: Narrowing as the Operational Semantics of Functional Languages. Proc IEEE Symposium on Logic Programming, pp. 138-151, 1985.

[26] D.H.D. Warren: An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, California, 1993.

[27] M. Wirsing: Algebraic Specification. In J. van Leeuwen (ed.): Handbook of Theoretical Computer Science, pp. 675-788, Elsevier Science Publishers, 1990.

[28] D. Wolz: Design of a Compiler for Lazy Pattern Driven Narrowing. In Recent Trends in Data Type Specifications, pp. 362-379. LNCS 543, Springer Verlag, 1990.