The Unification of Functional and Logic Languages ---Towards Constraint Functional Programming

John Darlington Yi-ke Guo
Department of Computing
Imperial College, University of London
180 Queen's Gate London SW7 2BZ U.K.

Abstract:
We will present in this paper constraint functional programming as a new declarative programming paradigm. A constraint functional programming (CFP) system is an integrated declarative programming system in which functional programming can be achieved by evaluating expressions on the computational domains. At the same time, logic programming can also be achieved by solving constraints over these domains. CFP is developed from the notion of absolute set abstraction, a language construct in functional languages acting as the " logic programming facility". CFP systematically unifies functional programming and logic programming from a fundamental semantic level and so is expected to provide great expressive power which comes from the natural merging of the solving and evaluating in an integrated system. Taking advantage of the domain-based programming style of functional programming, problem solving in CFP is directly performed on the intended domain of discourse in contrast to traditional logic programming which works only on the Herbrand universe. This property also allows some algorithmic constraint solving techniques to be exploited and restricts the search space in the computation to make the system be efficiently implementable.

1. Introduction

A constraint over a computation domain is a declarative statement of the relationships between objects in the domain and is also a computational device for enforcing these relationships. In the pioneering work of Steel[Stee80], the constraint model of computation was introduced into the area of declarative programming. Research on constraint programming is becoming more and more active in the area of language design. Due to the proximity of logic programming to constraint-based programming, extensive work has been conducted in introducing constraint solving techniques into the logic programming world[JaLaMa86]. These investigations led to a new logic programming system and coined the name "Constraint Logic Programming (CLP)". CLP views a logic programming language as a constraint language on the domain of discourse. Different logic languages can be obtained by employing different underlying computational domains. Some newly designed logic programming languages such as prologII[Colm82], prologIII[Colm87], CLP(R)[JaLaMa86] et.al fall within this framework.

Is it possible and also necessary to introduce constraint-based computation into the functional programming paradigm? This paper presents positive answers to these questions and proposes Constraint Functional Programming (CFP) as a new declarative programming paradigm to unify functional and logic languages.

In CFP, the basic components of a problem are stated by functions. Functions are defined by constrained defining rules that express the functional relations among objects which satisfy certain constraint relations on the computational domain. Constraint solving via these rules acts as a basic model. CFP inherits functional programming by expression evaluation using the rules together with the constraint solving mechanism which solve constraints to make the rules applicable.

Since the constraint computation model provides a natural solving mechanism, constraints act as the " logic programming facility" and are smoothly integrated into functional programming with an uniform semantic base. Since both pattern matching and unification are but two special cases of constraint solving mechanism, the CFP computation model subsumes the models for functional and logic programming and offers a superior expressive power to both. The domain specific constraint solving mechanism provides an implementation more efficient than the conventional search based problem solving on the Herbrand universe.

In the next section, we present the motivation and background of our research. A formal framework for constraint functional programming is presented in section 3. The operational model for CFP languages together with proofs of its soundness and completeness are developed in section 4. In section 5, a concrete constraint functional programming language, CHOPE (Constraint Hope) , is defined to illustrate the language aspects and the novel features of CFP. Some CFP examples are presented in CHOPE . Concluding remarks are given in section 6.

2. Motivation and Background

The unification of functional and logic languages can be achieved in various ways with different degrees of semantic clarity and mathematical rigorousness. Some recent work has been concentrated on extensions to functional languages. The main issue lies in integrating "logic programming facility" into pure functional languages to achieve the additional power. Some proposed languages incorporate the logical variable by means of conventional unification[Lind85]. These languages can be categorized into Guarded Functional Programming framework in which the guard for applying a defining rule contains predicates defined by a seperate logic programming component (e.g. Prolog). Kieburtz[Kieb87] proposed such a language and defined the execution mechanisms which are simply the interactions between reduction and SLD-resolution. The semantic features of the amalgamated language was discussed. The designated semantics for functional programming is a special form of equational logic defined over the domain containing all constructor terms. Predicate logic on this domain serves as the semantics for logic programming component. It implies that computation can only be performed on a single symbolic domain. Many proposed languages such as K-Leaf[Giov86], Guarded Term ML[Lock88] share this semantic property. However, this symbolic domain property contradicts the essential notion of functional programming.

A different approach was proposed by Darlington in [Darl86] which suggested a new language construct, called absolute set abstraction, as the basic facility for logic programming in a functional programming framework. An absolute set abstraction has the form { E(x,y) with y | P(x,y) }, where x stands for the bound variables and y for the free variables. It denotes the set of elements by evaluating E(x,y) for all y's satisfying the constraints P(x,y). A typical use is formed in defining the function "split" that returns all possible ways of decomposing a given list in terms of the append function.

```
dec split: list alpha -> set(list alpha x list alpha)
---split(l) <= {l1,l2 with l1,l2|append (l1,l2)= l}
```

In this example, the RHS of the function definition contains logical variables. Set abstraction defines the scope of logical variables and gives them the set_valued interpretations. Regarding logic variables as dummies over the computational domain and instantiating them by solving constraining equations, absolute set abstraction introduced for the first time the notion of constraint into functional programming. Constraints provide a mean to express the logic relations on the various computational domains. It is the research on the semantic and pragmatic issues of absolute set abstraction [DG89] that led to our new CFP system.

In constraint functional programming, functions are defined on objects which satisfy a certain constraint relations over the intended domains. Taking advantage of the domain-based feature of functional programming, the constraints on some computational domains can be naturally defined and their satisfiability can also be computed on these domains by some built-in solver together with a general goal reduction procedure. A model-theoretic semantics is established to give the declarative meaning of programs. Constraint functional programming language will have a greater expressive power w.r.t. conventional functional/logic programming languages. The functional programming style is still in use, the extra expressive power for general logical inference and non-deterministic computation comes from the integrated inference system with constraint solving.

3. Constraint Functional Languages

In this section, we establish the language framework for constraint functional programming and also the semantic model.

We start with defining, abstractly, constraint languages by regarding constraints as statements constraining their variables. Based on this formalization, we then extend constraint languages to the constraint language incorporating a functional language syntax. Therefore, functional languages, together with their underlying constraint languages, can be viewed totally as a constraint programming system. The constraint semantics and the functional semantics can be successfully merged within a uniform model

3.1 An Abstract Constraint Language

Syntax:

Given a computational domain D, a constraint language over D is a tuple: $< Var, \Phi, \vartheta, D>$ such that:

Var is a decidable, infinite set of variables.

Φ is a set of constraints expressing the relations between the objects in D. It can be defined as follows:

Given a set of pre-defined predicates Π_D over D, a relation $p_n(a_1,...a_n) \in \Phi$ where $p_n \in \Pi_D$ is an n-ary predicate predefined on D and each a_i is a variable or an object in D .

The empty conjunction $\varnothing \in \Phi$, if $\phi_1, \phi_2 \in \Phi$, $\phi_1 \& \phi_2 \in \Phi$.

ϑ is a mapping : $\Phi \rightarrow Var$ which defined as:

$\vartheta(\Phi)$ is a finite set of variables assigned to the constraint Φ to constrain their values, such that:

$\vartheta(p_n(a_1,...a_n)) = \{ a_i \mid a_i \text{ is variable} \}$

$\vartheta(\varnothing) = \varnothing$ and $\vartheta(\phi_1 \& \phi_2) = \vartheta(\phi_1) \cup \vartheta(\phi_2)$.

Semantics:

Each interpretation I, taking D as the computational domain, is a pair $(D, [\| \;\|]^I)$. By defining an I-assignment as a mapping $Var \rightarrow D$, and ASS^I is the set of all I-assignments, the interpretation function $[\| \;\|]^I$ is a a set-valued function : $\Phi \rightarrow P(ASS^I)$ such that :

$[\| p_n(a_1,...a_n) \|]^I = \{ \alpha \in Ass^I \mid \alpha(a_1,...a_n) \in p_n\}$

$[\| \varnothing \|]^I = ASS^I$

$[\| \phi_1 \& \phi_2 \|]^I = [\| \phi_1 \|]^I \cap [\| \phi_2 \|]^I$

All I-assignments in $[\| \phi \|]^I$ are the I-solutions of ϕ. A constraint ϕ is I-**satisfiable** *if* $[\| \phi \|]^I \neq \varnothing$. A special value "$\perp$" is used to denote an unsatisfiable constraint under the interpretation, i.e. $[\| \phi \|]^I = \perp$ iff $[\| \phi \|]^I = \varnothing$. A constraint ϕ is I-*valid if* $[\| \phi \|]^I = ASS^I$ and we can also say that I **satisfies** ϕ. For the declarative meaning of the language, we are only interested in the interpretation such that all the constraints in Φ are valid under that interpretation. These interpretations are called the model of Φ. That is, I is a model of Φ iff $\forall \phi \in \Phi \; [\| \phi \|]^I = ASS^I$.

Given A as a set of interpretations, we can define the subsumption relation and the equivalence relation on the constraint set Φ as follows:

1) $\phi_1 \leq \phi_2 :\Leftrightarrow \forall I \in A, [\| \phi_1 \|]^I \subseteq [\| \phi_2 \|]^I$

2) $\phi_1 \equiv \phi_2 :\Leftrightarrow \forall I \in A, [\| \phi_1 \|]^I = [\| \phi_2 \|]^I$

Given a set of variables V (called the solution variable set), the I-solutions for V in constraint ϕ are defined as :

$[\| \phi \|]^I{}_V := \{ \alpha|_V \mid \alpha \in [\| \phi \|]^I \}$.

We can correspondingly define the subsumption relation and the equivalence relation over the constraint set Φ for a variable set.

1) $\phi_1 \leq_V \phi_2 :\Leftrightarrow \forall I \in A, [\| \phi_1 \|]_V{}^I \subseteq [\| \phi_2 \|]_V{}^I$

2) $\phi_1 \equiv_V \phi_2 \equiv_V \phi_2 :\Leftrightarrow \forall I \in A, [\| \phi_1 \|]_V{}^I = [\| \phi_2 \|]_V{}^I$

In practical use, for a constraint language over D, all the predicates in D have fixed interpretations. All the elements in D that make all the predicates true is obviously a model of the language.

Let Φ be a set of constraints and I be an interpretation . The solutions of Φ in I are defined as : $[\| \Phi \|]^I := \cup \{[\| \phi \|]^I \mid \phi \in \Phi\}$

3.2. First Order Functional language

Syntax:

A signature $\Sigma = <S, F>$ is composed of a set of sort symbols (S) and a set of function declarations (F). A function declaration has the form $f: s_1 \times s_2 \times ...... \times s_n \rightarrow s$ where n is the arity of the function f and s, $s_1, s_2 ......, s_n$ are the sort symbols. We define $Var = (X_s)_{s \in S}$ as the family of the countable infinite sets of variables for every sort S. The Σ-term of sort s $T_\Sigma(X)_s$, is the minimal set of first order terms containing X_s which satisfies the condition :

$f(t_1,....t_n) \in T_\Sigma(X)s$ if $f: s_1 \times .... \times s_n \rightarrow s \in F$ and $t_i \in T_\Sigma(X)_{s_i}$.

A rewrite rule is a pair of terms (l, r) written in the form: $l \rightarrow r$ where l, $r \in T_\Sigma(X)_s$ such that $l \notin X$ and $\vartheta(l) \supseteq \vartheta(r)$. A term rewriting system R is a finite system of rewrite rules .

The term rewriting relation $\rightarrow_R$ associated with R is the smallest precongruence on $T_\Sigma(X)$ containing R which satisfies the following property:

$s \rightarrow_R t$ *if there is a rule* : $l \rightarrow r$ in R, for a subterm s_i in s, there exists a Σ-substitution θ such that $s_i = \theta(l)$ and $t = s[s_i \leftarrow \theta(r)]$.

For building functional languages over the Σ-terms, Σ should be partitioned into two disjoint parts, $\Sigma = \Sigma_d + \Sigma_c$. We call the function symbols in F_c the free constructors which are responsible for constructing the data structures, and those in F_d the defined functions, which act as the functions in the languages. The language may contain some predefined basic computational domains, such as integers, real numbers, boolean values et.al, so S could be partitioned further as $\Sigma_{val} + \Sigma_d + \Sigma_c$, where Σ_{val} is the signature for the predefined domains.

Therefore, given a signature Σ, we can abstractly define a first order functional language. Each function in a program is defined by a set of rewrite rules which are of the form: $Fd \rightarrow Exp$ where $F \in F_{d_n}$ is the principle function symbol to be defined and d is a tuple $(d_1,...,d_n)$ in which $d_i \in T_{\Sigma C}(Val,X)_{si}$ is a data pattern and no variable appears more than *once in* d. Exp is a term with the sort s, i.e. $Exp \in T_\Sigma(Val,X)_s$.

A program in the language is a set of rewrite rules in this form such that their LHS's are un-unifiable, i.e. there is no ambiguity for function definitions.

Semantics:

Let $\Sigma = <S, F>$ be a signature. An algebra is a tuple $A = (S^A, F^A)$, where S^A, F^A are the denotations for the sort and function symbols in Σ such that:

$S^A = (s^A)_{s \in S}$, is a family of non-empty carrier sets.

$F^A = (f^A)_{f \in F}$ is a family of functions $f^A: s_1{}^A \times s_2{}^A \times ...... \times s_n{}^A \rightarrow s^A$ for a function $f: s_1 \times s_2 \times ..... \times s_n \rightarrow s \in F$.

Given a Σ-algebra A , a valuation set (A-assignment) α is a family of mappings: $\alpha = (\alpha_s)_{s \in S}$, $\alpha_s : X_s \rightarrow s^A$

The interpretation $[\| \;\|]^A{}_\alpha = ([\| \;\|]^A{}_{\alpha,s})_{s \in S}$ is a family of mappings : $T_\Sigma(X)_s \rightarrow s^A$ which can be defined inductively on the structure of terms as:

$[\| x \|]^A{}_{\alpha,s} = \alpha(x) \;\; \forall x \in X_s$.

$[\| f(t_1,....,t_n) \|]^A{}_{\alpha,s} = f^A([\| t_1 \|]^A{}_{\alpha,s1}, ..., [\| t_n \|]^A{}_{\alpha,sn})$ for $f: s_1 \times .... \times s_n \rightarrow s \in F$.

An *interpretation* is a model of a functional program P, iff for all the rules $Fd \rightarrow Exp$ in P, $[\| F \|]^A{}_\alpha = [\| Exp \|]^A{}_\alpha$ for any A-assignment. Among all models of program P, the *initial* model I(P) gives the semantics. The universe of the model consists of one element for each P-congruence to act as its representative. Because of the confluency of functional programs, all P-equivalent ground terms have the same normal form. Therefore, the normal form algebra, which has the set of all P-normal forms as its universe, constructs the initial model by interpreting an operator by $f_P(t_1,....,t_n) = N(f(t_1,....,t_n))$ for all normal form t_i [Gogen80]. For a sufficiently complete program where every ground term is provably equal to a data value, this model captures the intended meaning of first order functional programming.

A Σ-algebra can be extended to a Σ-multialgebra by defining the denotations of function symbols as set-valued functions[Huss88].

For a signature $\Sigma = (\Sigma, F)$, a Σ-multialgebra A is a tuple , $A = (S^A, F^A)$; where S^A is a family of non-empty carrier sets defined as usual and $F^A = (f^A)_{f \in F}$ is a family of set-valued functions, such that for $f: s_1 \times s_2 \times ..... \times s_n \rightarrow s \in F$, $f^A: s_1{}^A \times s_2{}^A ...... \times s_n{}^A \rightarrow P^+(s^A)$, where $P^+(D) = \{N \mid N \subseteq D$ and $N \neq \varnothing\}$.

A multialgebra can be viewed as a relation system R where each set-valued function $f^A: s_1{}^A \times s_2{}^A ...... \times s_n{}^A \rightarrow P^+(s^A)$ is a relation f^R: $s_1{}^R \times .... \times s_n{}^R \times s^R$, that is:

$f^A(a_1, a_2..., a_n) = \{b \in S^A \mid (a_1, a_2..., a_n, b) \in f^R \}$

We write f(e)=a to mean $a \in f(e)$ when f is a multialgebra function.

Given a Σ-multialgebra A and a set of valuation, the interpretation $[\![\ ]\!]^A_\alpha = ([\![\ ]\!]^A_{\alpha,s})_{s \in S}$ is a family of mappings: $T_\Sigma(X)_s \rightarrow P^+(s^A)$, which map every Σ-term to its denotation in the multialgebra A. It can be defined inductively on the structure of terms as:

$$[\![\ x\ ]\!]^A_{\alpha,s} = \{ \alpha(x)\} \quad \forall x \in Xs$$

$$[\![\ f(t_1,\ldots,t_n)\ ]\!]^A_{\alpha,s} = \cup\{f^A(e_1,\ldots e_n) \mid e_i \in [\![\ t_i\ ]\!]^A_{\alpha,si}\}$$

$$= \{b \in s^A \mid (e_1,\ldots e_n, b) \in f^A \,\&\, e_i \in [\![\ t_i\ ]\!]^A_{\alpha,si}\}$$

3.3 Constraint Functional Languages

Given a signature Σ=(S,F) , We now present a construction, which takes the function symbols in Σ and a constraint language L, to construct a constraint functional language F(L) by defining functions over the constraint language.

Syntax:

Let L= (Var_L, Φ_L, ϑ, D) be a constraint language. A constraint functional language F(L)=($Var_{F(L)}$, $\Phi_{F(L)}$, $\vartheta_{F(L)}$, D) is defined as follows:

$Var_{F(L)} = Var_L = (X_s)_{s \in S}$

The constraints of F(L) can be defined inductively as follows:

1) $\Phi_L \subseteq \Phi_{F(L)}$

2) $f(x_1,\ldots,x_n) = y \in \Phi_{F(L)}$ $\quad x_i \in x_{s_i}$ and $y \in x_s$

for all $f:s_1 \times \ldots \times s_n \rightarrow s \in F$

3) If $\Phi_1 \in \Phi_{F(L)}$, $\exists x.\Phi_1 \in \Phi_{F(L)}$, where $x \in \vartheta_{F(L)}(\Phi_1)$

4)The empty conjunction $\varnothing \in \Phi_{F(L)}$, If $\Phi_1,\Phi_2 \in \Phi_{F(L)}$, then

1) $\Phi_1 \& \Phi_2 \in \Phi_{F(L)}$

2) $\Phi_1 :- \Phi_2 \in \Phi_{F(L)}$

The variables constrained by a constraint of F(L) are defined inductively as follows:

If $\phi \in \Phi_L$, $\vartheta_{F(L)}(\phi) = \vartheta_L(\phi)$;

$\vartheta_{F(L)}(\ f(x_1,\ldots,x_n) = y\ ) = (x_1,\ldots,x_n, y\ )$

$\vartheta_{F(L)}(\ \exists x.\Phi_1\ ) = \vartheta_{F(L)}(\ \Phi_1) - \{x\}$

$\vartheta_{F(L)}(\varnothing) = \varnothing$

$\vartheta_{F(L)}(\ \Phi_1 \& \Phi_2\ ) = \vartheta_{F(L)}(\ \Phi_1) \cup \vartheta_{F(L)}(\ \Phi_2)$

$\vartheta_{F(L)}(\ \Phi_1 :- \Phi_2\ ) = \vartheta_{F(L)}(\ \Phi_1) \cup \vartheta_{F(L)}(\ \Phi_2)$

In the remainder of the paper,we denote a L-constraint by ϕ_L(or simply ϕ) and F(L)-constraint by Φ for a CFP language F(L).

A constrained defining rule of f is an F(L) constraint of the form:

$f(x_1,\ldots x_n) = y :- \Phi$

where f is a Σ-function and Φ is a F(L) constraint.

The equation $f(x_1,\ldots x_n) = y$ is called " atomic equation" and will be abbreviated as $f(\vec{x})=y$ in the remainder of the paper. A well-formed F(L) program is a set of constrained defining rules.

For example, a functional program for defining the append function:

--- append(nil, x) <= x;

--- append(a :: x, y) <= a :: append(x, y) ;

can be viewed as the following F(L) program :

--- append(x, y) = z : - x = nil, z = y;

--- append(x, y) = z : - x = a :: x_1, z = a:: z_1, z_1 = append(x_1, y);

The underlying constraint language L for functional languages, following this example, can be simply defined as the first order matchings over the constructor terms.

A goal ϕ & E is a possibly empty conjunction L-constraints ϕ and atomic equations E. We use O(G) to denote the index set such that, for i $\in$ O(G), G/i is the ith atomic goal in G. Therefore evaluating the expression append([1,2],[3,4]) can be correspondingly regarded as solving the goal append(x,y) = z & x = [1,2] & y = [3,4] by the above F(L) language. If the underlying constraint language becomes first-order equations over constructor terms, the above program will be able to solve goals like append(x, y) = z & z = [1,2,3,4]. Thus the F(L) is extended to become a functional logic language. This approach will be exploited further by designing the semantic model.

Semantics:

A F(L) interpretation A is derived from an L-interpretation I, which is called the base of A. Σ-functions are interpreted as set valued functions in a multialgebra which takes D as the carrier set. The multialgebra interpretation for Σ-functions is necessary since we have shown that set valued functions are the key to integrate logic features and should be regarded as "first class citizens"[Darl86].

The semantic equations of F(L) are defined as :

$D^A = D^I = D$

$[\![\ \phi\ ]\!]^A = [\![\ \phi\ ]\!]^I \qquad$ if $\phi \in \Phi_L$

$[\![\ f(x_1,\ldots,x_n) = y\ ]\!]^A = \{\alpha \in ASS^A \mid \alpha(x_1,\ldots,x_n,y) \in f^A\}$

$[\![\exists x.\Phi_1\ ]\!]^A = \{\alpha \in ASS^A \mid \exists \beta \in [\![\ \Phi_1\ ]\!]^A\ \forall y \in \vartheta(\Phi_1) - \{x\}\ \alpha(y)=\beta(y)\}$

$[\![\ \varnothing\ ]\!] = ASS^A$

$[\![\ \Phi_1 \& \Phi_2\ ]\!]^A = [\![\ \Phi_1\ ]\!]^A \cap [\![\ \Phi_2\ ]\!]^A$

$[\![\ \Phi_1 :- \Phi_2\ ]\!]^A = \{ASS^A - [\![\Phi_2\ ]\!]^A\} \cup [\![\Phi_1\ ]\!]^A$

For a underlying constraint language L, its interpretation I is always fixed. We denotes all F(L) interpretations by A(I) which are extended from I as shown above and thus called I-based interpretations.

The following proposition is obvious.

Proposition 3.1:

$\forall A_1,A_2 \in A(I)$,$[\![\ \phi_L\ ]\!]^{A_1} = [\![\ \phi_L\ ]\!]^{A_2}$.

A partial order among A(I) can be defined as $\forall A_1,A_2 \in A(I)$, $A_1 \subseteq A_2$ iff $f^{A_1} \subseteq f^{A_2}$, for $\forall f \in F$.

For any two $A_1,A_2 \in A(I)$, the intersection $A_1 \cap A_2$, the union $A_1 \cup A_2$ are I-based interpretations obtained by intersecting or joining the denotations of functions respectively. i.e.

$f^{A_1 \cap A_2} = f^{A_1} \cap f^{A_2}$

$f^{A_1 \cup A_2} = f^{A_1} \cup f^{A_2}$

Proposition 3.2:

Let I be a L-interpretation, <A(I), $\subseteq$, $\cap$, $\cup$ > is a complete lattice.

For a F(L) program P, if an I-based interpretation $A \in A(I)$ is a model of P, then we have the following proposition.

Proposition 3.3:

Given F(L) program P,$\forall A \in A(I)$, A is a model of P iff for all constraint defining rules $E_i :- \Phi \in P$, we have: $[\![\ E_i\ ]\!]^A \supseteq [\![\ \Phi\ ]\!]^A$

In the following, we use $M(I)_p$ to denote all I-based models of P. We are now reaching the main result of this section. For any F(L) program, there exists the minimal model which determines the declarative semantics for the program by defining the unique minimal denotations for all function symbols.

Lemma 3.4:

Given a F(L) program P, $M(I)_p$ is its model set, then there is a minimal model $A_m \in M(I)_p$ which is the greatest lower bound of $M(I)_p$.

Proof(sketch):

We can prove that the intersection of models in $M(I)_p$, $\cap \{A_i \mid A_i \in M(I)_p\}$, is a model of P. Therefore $A_m = \cap\{A_i \mid A_i \in M(I)_p\}$, is the minimal model.

The function $T^f_{(P,I)}$: $P(ASS^I) \rightarrow P(ASS^I)$ maps from an I-based interpretation A to a new I-based interpretation A' for a function f : $T^f_{(P,I)}(A) = \{\alpha(\vec{x}, y) \mid$ there exists $f(\vec{x}) = y :- \Phi \in P \,\&\, \alpha \in [\![\Phi\ ]\!]^A\}$ and $T_{(P,I)}(A) = \cup \{T^f_{(P,I)}(A) \mid f \in F\}$

Proposition 3.5:

$T_{(P,I)}(s)$ is continuous on the lattice < A(I), $\subseteq$, $\cap$, $\cup$ >.

Using $T_{(P,I)}$, we can build the minimal I-based model for program P in the following way.

$T_{(P,I)} \uparrow 0 = \varnothing$

$T_{(P,I)} \uparrow (\alpha+1)$ = if α is a successor ordinal

then $T_{(P,I)}\ (T_{(P,I)} \uparrow \alpha)$

else $\cup\{T_{(P,I)} \uparrow \beta \mid$ for all $\beta < \alpha$ }

Lemma 3.6:

The least fixpoint lfp $T_{(P,I)}$ exists and lfp $T_{(P,I)} = T_{(P,I)} \uparrow \omega$ where ω is the smallest limit ordinal .

Proof(sketch):

By the fact that :$T_{(P,I)} \uparrow 0 \subseteq T_{(P,I)} \uparrow 1 \subseteq \ldots \subseteq T_{(P,I)} \uparrow n \subseteq \ldots$

and the continuity of $T_{(P,I)}$.

Theorem 3.7 (Minimal Model Theorem):

For any F(L) program P,

a) $A \in M(I)_p$, i.e. A is an I-based model of P , iff $T_{(P,I)}(A) \subseteq A$.

b) P has a least I-based model, which is equal to $A_m = T_{(P,I)} \uparrow \omega$.

Proof(sketch)
a) is straightforward

b) $A_m = \mathrm{glb}\{A : A \in M(I)_p\}$
$= \mathrm{glb}\{A : T_{(P,I)}(A) \subseteq A\}$
$= \mathrm{lfp}\ T_{(P,I)}$
$= T_{(P,I)} \uparrow \omega.$

It is obvious that $f^{A_m} = T^f_{(P,I)} \uparrow \omega$, for all function symbols in F. This indicates that the minimal model theorem represents the procedure for generating the unique minimal denotation for all the functions in the program. It is this result that bridges the gap between the declarative semantics and the operation model and reveals the most essential notion of the language.

This technique was used in logic programming community as a standard procedure to construct the Herbrand model from the Herbrand base[Lloy84]]. In his research on CLP scheme, Jaffar generalized this procedure to construct the minimal model from a unification complete computation domain[JaLaMa86]]. This idea was developed further by Höhfeld to construct minimal model for the constraint-based relational knowledge base [Höh88]. The semantic work we presented here is inspired by his work.

4. Operational Model

In this section, we present a complete interpreter for constraint functional programming. The interpreter is a non-deterministic algorithm for solving a goal by choosing atomic goals with don't care non-determinism and try all rules for reducing an atomic equations with don't know non-determinism. This algorithm can be considered as the generalization of the conditional narrowing method[Ders85] employed for solving equations over the conditional equational theory [Ders88].

4.1 The Interpreter

An F(L) interpreter consists of two kinds of operation on goals, which are goal rewriting and constraint solving, to reduce atomic equations and to solve L-constraints respectively.

For a CFP program P, a one-step goal rewriting $\stackrel{r}{\hookrightarrow}$ is defined as a binary relation over the set of goals by the following definition.

Goal Rewriting :

Given a goal G, $\exists\, i \in O(G)$, s.t. $G / i = E$, $G \stackrel{r}{\hookrightarrow}_{<P,i,V>} G' = G[i \leftarrow s']$, iff E is an atomic equation $f(\vec{x}) = y$, $\exists E' \text{:-} s' \in P$ is a variant of constrained defining rule for f such that $(V \cup V') \cap \vartheta(s') \subseteq \vartheta(E)$ and $V' = \vartheta(G) - \vartheta(E)$.

Constraint Solving :

Constraint solving is a binary relation of the goals such that for a goal G, $G \stackrel{c}{\hookrightarrow}_{<\varphi,i,V>} G' = G[i \leftarrow \phi'_L]$ iff $\exists i \in O(G)$ s.t. $G / i = \phi$ and $\phi' \equiv_{V \cup \vartheta(G - \phi_L)} \phi$.

By $\stackrel{r}{\hookrightarrow}^*, \stackrel{c}{\hookrightarrow}^*, \stackrel{r,c}{\hookrightarrow}^*$ we denote the transitive closure of $\stackrel{r}{\hookrightarrow}, \stackrel{c}{\hookrightarrow}$ and their combination.

Goal rewriting provides a very general means for all kinds of rule-based rewriting computation. The differences among them, characterized by the rule-appliability criterion such as pattern matching in rewriting, term unification in narrowing[Fay79], is now systematically abstracted and treated uniformly as constraint solving in the underlying constraint language. Therefore, the underlying constraint system determines the features of the various constraint functional languages which all take goal-rewriting as basic reduction operation. For example, Hope with Unification [Darl86] is a CFP language where the underlying constraint language is only the equations between constructor terms, which takes the word algebra $g(T_\Sigma)$ as its unique interpretation.

The underlying constraint language comes with a set of canonical constraints which are satisfiable for the fixed interpretation. A complete constraint solver is provided to solve every satisfiable conjunction of L-constraints to V-equivalent constraint in a canonical form, where V is a finite set of solution variables. So canonical constraints serves as the answer constraint We illustrate this by the following algorithm.

Constraint Solver $(\Phi_1, V) =$

if Φ_1 is consistent then (Φ_2, V) else $\perp$.

where $\Phi_1 \equiv_V \Phi_2$ and Φ_2 is in the canonical form

A general interpreter for a CFP language is illustrated below. It will be proved to be able to compute a complete set of answer constraints for a given goal. The interpreter solves a goal G to its answer constraint by rewriting atomic equations or solving L-constraints (with don't care non-deterministic choice) using the associated rules in P (with don't know non-deterministic choice) and the constraint solver provided by the underlying constraint system.

```
F(L)_Interpreter (P, $\phi$ & E, V)
<= if E= $\emptyset$
      then L-Constraint Solver ($\phi$, V)
    else
      case :
        1) $\phi$ is $\perp$
              then {$\perp$ }
        2) $\phi$ is in a solvable form
           then let $\phi'$ = L-Constraint Solver ($\phi$, V)
                in F(L)_interpreter(P, $\phi'$&E, V)}.
        3) else if $\exists k \in O(E)$, s.t. E / k = $f(\vec{x}) = y$ is rewritable
                then $\cup$ {F(L)_Interpreter(P, $\phi$ & $S_i$, V) | $E \xrightarrow{r}_{(p,k,V)} S_i$ }
                else {$\perp$}
```

It is obvious that the F(L)_interpreter solves a goal by goal-rewriting interleaved with some constraint solving steps. So the following propositions is straightforward :

Proposition 4.1

For all goal G,

a) if $G' \in$ F(L)_Interpreter(P,V,G), then $G \stackrel{r,c}{\hookrightarrow}^* G'$.

b) if $G' \in$ F(L)_Interpreter(P,V,G), then G' is either ϕ or $\perp$,where ϕ is a canonical L-constraint.

Proof(sketch):

a) is straightforward.

b) can be proved in terms of a recursive-path ordering.

We denote comp(P,V,G) as the set:

$\{\phi \mid \phi \in$ F(L)_interpreter (P,V,G) and $\phi \neq \perp\}$.

4.2 Soundness and Completeness

Soundness:

The soundness of the interpreter is that all the constraints ϕ_i computed by the interpreter for solving G are the approximations of G. That is:

$$\forall \phi_i \in \mathrm{comp}(P,V,G) \quad [\![\phi_i]\!]^A{}_{|V} \subseteq [\![G]\!]^A{}_{|V} \quad \text{for } \forall A \in M(I)_p$$

To prove this we start with proving the soundness of two basic operators, goal rewriting and constraint solving, in the interpreter :

Lemma 4.2

Let G_1, G_2 be two goals, and $G_1 \stackrel{r}{\hookrightarrow}_{<p,i,v>} G_2$ then $[\![G_1]\!]^A{}_{|V} \supseteq [\![G_2]\!]^A{}_{|V}$ where $A \in M(I)_p$

Proof (sketch):

$\exists\, i$, $i \in O(G)$ and $G / i = f(\vec{x}) = y$ and there exists a variant of rule $f(\vec{x}) = y$:- Φ in P , then $G_2 = G_1\, [i \leftarrow \Phi]$. For $\forall A \in M(I)_p$, let $\alpha \in [\![G_2]\!]^A{}_{|V}$, then $\exists \beta \in [\![G_2]\!]^A$ and $\beta_{|V} = \alpha$, $\beta \in [\![\Phi]\!]^A$. Since $f(\vec{x}) = y$:- Φ is valid in A, then $\beta(\vec{x}, y) \in f^A$, so $\beta \in [\![G_1]\!]^A$ and $\beta_{|V} = \alpha \in [\![G_1]\!]^A{}_{|V}$.

The following Lemmas are straightforward :

Lemma 4.3

For two goals G_1, G_2, if $G_1 \stackrel{r}{\hookrightarrow}^*_{<p,v>} G_2$ then $[\![G_1]\!]^A{}_{|V} \supseteq [\![G_2]\!]^A{}_{|V}$ for $\forall A \in M(I)_p$.

Lemma 4.4

Let G_1, G_2 be two goals and $G_1 \stackrel{c}{\hookrightarrow}_{p,i,v} G_2$ then $[\![G_1]\!]^A{}_{|V} = [\![G_2]\!]^A{}_{|V}$ for $\forall A \in M(I)_p$.

Theorem 4.5 (Soundness Theorem).

For every answer constraint ϕ computed by the F(L) _interpreter, i.e.

$$\forall \phi_i \in \mathrm{comp}(P,V,G) \quad [\![\phi_i]\!]^A{}_{|V} \subseteq [\![G]\!]^A{}_{|V} \quad \text{for } \forall A \in M(I)_p$$

Proof(sketch) :

Since $\phi_i \in$ comp(P,V,G), $G \stackrel{r,c}{\hookrightarrow}^*_{p,v} \phi_i$, the soundness is established following lemma 4.3 and lemma 4.4.

Completeness

The completeness of the interpreter states that for any possible L-constraint ϕ, which is approximate to a goal G, ϕ must be subsumed by all computed answer constraints by the interpreter. That is:

$$\forall \phi\ [\![\phi]\!]^A{}_{|V} \subseteq [\![G]\!]^A{}_{|V} \Rightarrow [\![\phi]\!]^A{}_{|V} \subseteq \cup\{ [\![\phi_i]\!]^A{}_{|V} \mid \phi_i \in \mathrm{comp}(P,V,G)\}$$

for $\forall A \in M(I)_p$.

To prove the completeness of the interpreter, we first prove the answer preserving property of goal rewriting with don't know non-deterministic rule choice.

Lemma 4.6

Let G be a goal, A_m the minimal model in $M(I)_p$, and $\exists\ i \in O(G)$, $G/i = E: f(\vec{x}) = y$. If $\alpha \in [| G |]^{A_m}_{|V}$, then there exists a variant of the defining rule of f, such that $G \rightarrow_{<p,i,v>} G'$ and $\alpha \in [| G' |]^{A_m}_{|V}$.

Proof(sketch):

Let $G = (f(\vec{x}) = y) \& G''$. Since $\alpha \in [| G |]^{A_m}_{|V}$, $\exists \beta \in [| G |]^{A_m}$ and $\beta_{|V} = \alpha$. Thus $\beta(\vec{x}, y) \in f^{A_m}$. By the construction procedure of A_m, $\exists$ i, s.t. $\beta(\vec{x}, y) \in f^i$ and a defining rule l : $f(\vec{x}') = y' :- \Phi \in P$, s.t. $\sigma \in [| \Phi |]^{A_{i-1}}$ and $\sigma(\vec{x}, y') = \beta(\vec{x}, y)$, therefore $G \xrightarrow{r} G' = \Phi' \& G''$ where $f(\vec{x}) = y :- \Phi'$ is the proper variant of the rule l.

From this Lemma, we can directly prove the following lemma which indicates that the interpreter can generate a complete answer constraint set for the minimal model of A.

Lemma 4.7

$[| G |]^{A_m}_{|V} = \cup \{[| \phi_i |]^{A_m}_{|V} \mid \phi_i \in comp(P,V,G)\}$

Theorem 4.8 (Completeness Theorem).

Given a goal G and a variable set V, $\forall A \in M(I)_p$

$\forall \phi\ [| \phi |]^{A}_{|V} \subseteq [| G |]^{A}_{|V} \Rightarrow [| \phi |]^{A}_{|V} \subseteq \cup \{[| \phi_i |]^{A}_{|V} \mid \phi_i \in comp(P,V,G)\}$

Proof (sketch) :

It is straightforward from lemma 4.7 and the fact that

$[| \phi |]^{A}_{|V} \subseteq [| G |]^{A}_{|V} \Leftrightarrow [| \phi |]^{A_m}_{|V} \subseteq [| G |]^{A_m}_{|V}$

The interpreter presented here is an abstract operational model of CFP systems in the sense that any computation, including expression evaluation, is systematically treated as constraint solving. The completeness and soundness result indicates that for any F(L) program, if the underlying constraint system comes with a set of canonical constraints, a goal can be correctly solved by the interpreter to a complete set of answer constraint in which each one is a canonical L-constraint.

5.CHOPE: A Constraint Functional Language

In this section, we illustrate the essential notions of CFP as well as its programming styles by defining a simple language CHOPE (Constraint Hope) which is extended from the functional language Hope[Burs80] by embedding some constructs for programming constraints. It is evaluated by reducing expressions to their values and by solving constraints.

A program in CHOPE is a set of function definitions which may involve constrained expressions which is the essential language construct for constraint programming. The syntax of the defining equations are shown as follows:

Defining Equations:

<equations> ::= <equation> {';' <equation> }+
<equation>::= <left hand side> '<=' <expression>
| <left hand side> '<=' <constrained expression>
<left hand side>::= '---' <function name> [<pattern>]

Constrained Expression:

<constrained expression::= <existential quantified expression>
| <basic constrained expression>
| <absolute set abstraction>
<existential quantified expression>::=
'∃'<variable tuple> '.' <expression>
<basic constrained expression>::=
<expression> <constraint declaration>
<absolute set abstraction>::=
'{' <basic constrained expression> '}

Constraint:

<constraint declaration>::=
{<variable declaration>} 's.t.' <constraints>
<variable declaration> ::= 'with' <variable tuple>
<constraints> ::= <constraint> {','<constraint>}+
<constraint>::=
<expression> <domain-specific relational symbol> <expression>
| <expression> '∈' <expression>

The semantics of CHOPE is consistent with the CFP framework. Any CHOPE program can be transformed to the standard CFP form without changing the semantics. Its operational model can be viewed as an optimization of the general CFP interpreter.

The type system, which is derived from HOPE, prescribes the computation domains which are constructed by data constructors together with elements of some basic domains. Assuming that the basic domains in CHOPE are integer number, real number, boolean number, the domain of computation can be abstractly defined as an algebra that consists of integer arithmetic term, real arithmetic term, boolean arithmetic term, and structured terms comprising of the combination of arithmetic terms and constructors. For numerical arithmetic terms, the usual arithmetic constraint : {<, >, =, ≠, ≤, ≥} are well-formed in CHOPE. For structured terms, the identity relation "=" and "≠" are the only two constraints for terms. For example, suppose age, salary and status are functions defining the age, salary and the status of a person, the following set of constraints :

```
{age(x)≥50,salary(x)>30000,status(x)=professor.}
```

are valid and constrain the people in the discourse domain to be the set of professors who are older than 49 and have a salary greater than $30000.

When a constrained expression is declared as set-valued, absolute set abstraction is used for denoting the set of expression results. The set membership '∈' is a basic constraint relation between a single-valued expression and a set-valued expression . For example, the following program evaluates the permutations of a given list in terms of absolute set abstraction:

```
dec Permu: list(alpha)->set(list(alpha));
--- Permu (nil) <= {nil};
--- Permu (a::L)
       <={u<>a::v with u,v | u<>v ∈ Permu (L)}
```

In the program, u<>v ∈ Permu (L) is the constraint which is satisfied when the result of the expression u<>v is the element of the set denoted by the permutation expression.

We present a couple of examples of constraint functional programming in CHOPE to illustrate the expressive power. All examples used are typical in the area of traditional logic programming, constraint programming and also database programming.

5.1 Programming with logical variables

Programming with logical variables is one of the essential features of the constraint functional programming. In the language, logical variables can be introduced into expressions by constraints and bound incrementally through constraint intersection. Therefore, some special features, such as incomplete data structure, "on-the-fly" assignment, familiar in traditional logic programming can now be easily performed.

An example is the well known program for address translation. That is, translating a list of definitions, Def(name), and references to defined names,Use(name), into a list of concrete assignments, Asgn(name, address), and references to assigned addresses Use(address), respectively in a single pass. E.g. the input list: [Def(a), Use(a), Use(b), Def(c), Def(b)] is to be translated into the output list: [Asgn(a,1),Use(1),Use(3),Asgn(c,2), Asgn(b,3)] .

E.g.1 Address Translation

```
data term == Def(char)++Use(char);
data out_term == Asgn(char#num) ++ Use(num);
dec member: alpha # list(alpha) -> truval;
dec trans:list(term) # num # (char # num) ->
list(out_term)
--- member(a, b::S) <= true s.t. a = b;
--- member(a, b::S) <= member(a,S) s.t.a ≠ b;
---trans(Use(a)::S,n,T)<=Use(i)::tran(S,n,T)
       with i s.t. member ((a,i),T)=true;
---trans(Def(a)::S,n,T)<=Asgn(a,n)::trans(S,n+1,T)
      s.t. member((a,n),T) = true;
---trans(nil, n, T) <= nil
```

In the program, the trans function is responsible for translating the input list. Following its first definition, the right hand side expression is constrained by the equation member ((a,i), T) = true. In the program, T is a logic variable introduced by the top level expression which is existentialy quantified and acts as an association table built up "on -the-fly" When evaluating the right hand side expression, its constraining equation is solved to constrain logic variable T or for logic variable i .

The second defining equation for trans is constrained by the equation member ((a,n), T) = true. In evaluation, the logic variable T or the instance of the logic variable i introduced by an expressions that matches the first equation may be bound through solving the equation. By reducing the expression ∃T.trans (input_list, 1, T), the intended output list will be incrementally constructed by traversing the input list once.

E.g. 2 Difference Structures :

Programming with difference lists is always regarded as a logic programming technique for pursuing more concise and efficient programs. The idea behind difference list is that any list may be considered as the difference between pairs of lists, i.e. a structure Diff(A_s,B_s), which takes A_s as head and B_s as tail can be defined by the type declaration :

```
data difference_list(alpha)== Diff (list (alpha) # list(alpha))
```

Hereinafter, we use an infix constructor / (pronounced differ) for representing a difference list.

The magic of difference list lies in the incompleteness of its component lists (i.e. they always have logic variable tails). Therefore, it may be used in the way of a queue with advantage of adding elements to its end. In order to introduce different structures into CFP, we face the problem of evaluating non-ground data structure. In CHOPE, non-ground data structures are interpreted by set abstraction. That is, a non-ground structure A has no meaning on itself while set {A} is meaningful and defines the values which may be computed incrementally by imposing more and more constraints on its logic variables. Therefore,a list [1,2,3] corresponds to a set of different lists denoted as {[1,2,3,x]/[x] | x∈ num}, or simply {[1,2,3,x]/[x]}. This interpretation was made by Reddy who regards the functional languages with non-ground data structure as F-languages[Reddy86]. Incorporating it into CFP framework is very natural since constraint model provides an elegant way for computing non-gound data structures. We illustrate the use of different structures by the following program which processes a list of commands for dealing with a FIFO queue.

The function Queue (S,Q) implements a command sequence which contains two kinds of commands w.r.t. two basic operations on a queue: 1) enqueueing an element by the command enqueue(X) and 2) dequeueing an element by dequeue(X) where X is the element concerned.

```
data commands == enqueue(alpha) ++ dequeue
data queue(alpha)== set_of difference_list (alpha)
dec Queue: list(commands) # queue(alpha) -> queue(alpha)
dec enqueueing: alpha # queue(alpha) -> queue(alpha)
dec dequeueing: queue(alpha) -> queue(alpha)
--- Queue(enqueue(X)::Ls, Q)<= Queue(Ls, Q')
              where Q' = enqueueing(X, Q);
--- Queue(dequeue::Ls,Q)<= Queue(Ls, Q')
              where Q' = dequeueing(Q);
--- Queue(nil, Q) <= Q;
--- enqueueing(X, Q)<= {Qh/Qt with Qh,Qt|Qh/[X::Qt] = Q',Q'∈Q};
--- dequeueing(Q)<= {Qh/Qt with X,Qh,Qt|[X::Qh]/Qt =Q',Q'∈Q};
```

Following this program, the evaluation of the expression ∃q.Queue(S,{q/q}) will implement the command sequence.

5.2. Constraint Programming

Constraint programming is one of the essential features of the system. It is based on constraint solving algorithms on the basic computation domains, such as the Gaussian elimination algorithm for solving linear equations on real domain and Boolean unification for solving constraints on the boolean domain. Constraints can therefore be handled in a uniform way over the input, execution and output of programs.

The example is modeling circuit calculation systems based on Ohm law:

E.g.3. Circuit Calculation

```
type Vot, Am, Ohm == real;
dec resistor: Vot x Am -> Ohm
dec par_cir: Vot x Ohm x Ohm -> Am;
dec ser_cir: Am x Ohm x Ohm -> Vot;
--- resistor(V,I) <= V/I;
--- par_cir(V,R1,R2)<= I1 + I2 with I1,I2 s.t.
resistor(V,I1)= R1,resistor(V,I2)= R2;
--- ser_cir(I,R1,R2)<= V1 + V2 with V1,V2 s.t.
      resistor(V1,I) = R1,resistor(V2,I) = R2;
```

More complicated systems can be built up by a multi-level hierarchy of components, e.g.

```
--- par-ser(I,R1,R2,R3,R4)<= V1+ V2 with V1,V2
      s.t.par_cir(V1,R1,R2) =I,par_cir(V2,R3,R4)=I;
```

we can evaluate the expression:

```
par_series(5, 10, 10, 10, 10)
```

and get the value = 50, and we can also compute a goal like:

```
I with I s.t. par_ser(I,10,10,10,10) = 50
```

and get the answer I = 5.

Suppose we know all the resistance values but not the voltage and current. By evaluating following set abstraction:

```
{I,V| par-ser (I,10,10,10,10) = V}
```

we get the relationship between the voltage and the current {I,V | V=10I} as our answer. So such a system is simply modeled by a CFP program. It can be flexibly used to design specialized solutions to map circuit problems.

5.3. Database Programming

It is well known that logic can be used to describe the static database and provides the deductive ability to the dynamic processing of database by its inference mechanism. By using logic uniformly for data description, data manipulation as well as data query, the conventional distinction between data base and program no longer applies. Since CFP provides a functional data description capability and also a powerful inference paradigm, CFP is amenable to database programming. An example is the following sales and purchases data base. It illustrates the multi-use of functions in CFP. This ability is the key for performing database querying. Also it is shown that constraints provide some basic means for dealing with the dynamic change of the database.

E.g 4. Sales and Purchases Database:

```
dec Sold: item x time -> quantity;
dec Purchases: item x time -> quantity;
dec Inventory: item x time -> quantity;
dec total-sold: time x time-> quantity;
dec total-bought: time x time-> quantity;
--- Inventory(item,date)<= total with startdate, startq  s.t.
    inventory(item,startdate) = startq, total = startq - total-
    sold(startdate,date)+total- bought(startdate, date);
--- total-sold(startdate, date)
      <= sum {x with x | sold(idem, time1) = x,
              startdate ≤ time1 ≤ date };
--- total-purchases(startdate, date)
      <= sum {x with x | purchases(idem,time1) = x,
              startdate≤ time1 ≤date };
```

where sum is defined on a set and calculates the sum of all the elements in the set.

In this database program, if an initial inventory such as : Inventory(Apples, 1.Jan.1987) <= 100 is provided to the data base, a query can be processed deductively by evaluating query expression such as: Inventory(Apples, 12th.Jan.1987) or by solving the goal such as :{days | sold(Apples, days) >100}. The dynamic change of the database, which happens in this example when an sale or purchase action is performed, can be simply dealt with by adding the functional event description to the database. For example, if some apples were sold at 3th.Jan.1987 and some were bought at 4th Jan 1987, the database is changed by putting the facts into the database such as :

```
sold(Apples, 3th.Jan.1987) <= 15;
purchases(Apples, 4th.Jan.1987) <= 18;
```

By introducing explicit time constraint for every time-varying relation, the whole database is maintained consistently without any side-effect normally caused by the database updating. Absolute set abstraction organizes these changes and processes the query by constraint satisfaction.

5.4 Extended Functional Programming

Many accounts of extending functional languages such as monolithic arrays[Arvind87], set-level functional programming [JaPl87], functional programming with temporal constraints[While88] can be uniformly treated within CFP context. This comes from the "universal" property of our CFP framework, that is, different language features can be characterized by different underlying constraint systems. We present two examples concerning array structure and set level programming to illustrate this claim.

E.g. 5 I-Structure

I-structure is a kind of monolithic array. An I-structure is an array composed of logic variables. Three constructs are adopted for allocating, storing, and reading from arrays. They are:

1. A=array(n), allocating an array of size and naming it as A.
2. A[i]=V, unifying value V with A[i] and storing the result.
3. A[i], selecting the ith element of A.

I-structure can not be produced as the result of evaluating a single expression but will be determined by binding the logic variables incrementally. This process fits well with our CFP model. The three constructs can be regarded as the underlying constraints over the array domain. That is, A=array(n) is the constraint which assert the size of A and A[i] =V are constraints which assert the equivalence between the ith element of A and V. Therefore, I-structure can be defined and computed within CFP framework which is embedded with such an underlying constraint system on arrays. This observation was also made by Jagadeesan et.al. in their research on the semantics of I-structure[JaPaPi89]. As an example, we can write a CHOPE program to solve the inverse permutation problem [Arvind87] in a way similar to using I-structure.

```
dec Vectorsize: vector(alpha) -> num;
     /* a primitive function on the array domain*/
dec Newvector: num -> set_of(vector(alpha))
dec Inverse: vector(alpha) -> vector(alpha);
---Newvector(n) <= {v with v| Vectorsize(v)=n};
---Inverse(P) <= let n=vectorsize(P)
                 in V with V s.t.
                 V ∈ Newvector(n),V(P(i))=i,i∈{1...n}.
```

E.g.6. Set Level Functional Program.

In [JaPl87], Jayaraman and Plaisted proposed a novel equational language for set-level functional programming. In the language, sets are treated as sets, instead of lists as in many functional languages[Turn85], to be consistent with their semantics. Each set is constructed in terms of constructor {} to denote empty set and two associative-commutative constructors $\cup$ and scons where $\cup$ is the set union and scon(h,t) $\equiv$ {h | t} $\equiv$ {h}$\cup$t. By taking set as the basic data structure, associative-commutative matching (a-c matching) is used for functional application. Many recursive function definition can be simplified by program on set level resulting concise codes. This approach can also be incorporated into CFP framework by adopting set structure together with the a-c matching, denoted as $=_{AC}$ over the structure as a constraint system. So set-level functional programming is available in a CFP framework .

```
dec diff: set(alpha) x set(alpha) -> set(alpha)
--- diff({},s) <= {};
--- diff(s,{}) <= s ;
--- diff(x,t) <= ∪ {if h ∈ t then {} else {h}
    with  h |{h|_} = AC x}.
```

6.Related Work and Conclusion

A language framework for constraint functional programming is presented in this paper. It consists of an underlying constraint language to express the constraint relations in the computational domain. Functions defined on the domain map the objects which satisfy the constraints. A model theoretical semantics is established with the multialgebra interpretations for the function symbols. A simple fixpoint characterization of the minimal model,which is always regarded as one of the elegant semantic feature of logic programming systems, is available in our CFP framework. We claim that CFP has a more expressive power compared with conventional logic programming systems.

Embedding constraints in declarative languages provides extra expressive power. As the counterpart of constraint functional programming, constraint logic programming performs logic inference over the domain of discourse instead of on a single Herbrand domain. This property offers some functional programming features of CLP. Besides the functional syntax and the reduction based operational model, CFP is still different from the CLP approach at the semantic level. In CFP, the well-designed type system capture the essential notion of computational domain , thus this concept is much more general than the one in CLP scheme. Some restrictions on the computational domain , such as "solution compactness" which is useful for the "closed word assumption" of the complete lógic inference system, are not necessary in the CFP framework. Because of the domain-based feature of functional programming, the underlying constraint language can be naturaly regarded as the relations over expressions with primitive functions on the data value domains and therefore can be formalized within the framework. Although CFP is only defined for first order functions, the promising feature of higher order functions can be performed at a higher level via transformation or compilation as in the Ideal system[Giov86].

Some researches on conditional term rewriting also aim to integrate functional and logic programming by means of conditional equational reasoning[Ders 85]. Although there are some obvious similarities between our CFP interpreter and those of conditional equational programming, the difference is that our semantics does not base on the equational logic. Therefore, CFP works at a much more general setting. Some requirements, which are always imposed on the operational model of equational programming such as confluent rewriting, termination, can be relaxed to provide greater programming convenience.

Research on CFP is just at its beginning. Much work remains. The semantic model we present here is very concise for defining a general language framework. However, when designing concrete CFP languages, we need much more work on the details of underlying constraint system and its interactions with functional programming world. It is also important to inherit a lot of well-designed work such as lazy evaluation, various type systems in functional programming research. The operational model presented here is also a general mechanism. In order to implement a CFP language, many optimizations should be made to achieve a reasonable performance. An obvious optimization is to make distinction between expression evaluation, which can be efficiently performed by reduction, and constraint solving. An experimental interpreter of CHOPE is under construction for further intensive research.

References

[Arvind87] Arvind, Nikhil,S.R.and Pingali,K.K.," Id Nouveau Reference Manual " CSG Report, MIT, Apr.1987.

[Burs 80] Burstall R. et.al. " Hope : an experimental applicative language" Proc. Lisp Conf. Stanford, 1980.

[Colm82] Colmerauer,A." Prolog and Infinite Trees" in Logic Programming, ed. K.L.Clark and S.A.Tarnlund, Academic Press, New York, 1982.

[Colm87] Colmerauer,A."Opening the Prolog III Universe" BYTE, July, 1987.

[Darl 86] Darlington, J. Field, A.J. and Pull, H." The unification of Functional and Logic languages" in Logic Programming: Functions, Relations and Equations, ed. Doug Degroot and G. Lindstrom, Prentice-Hall , 1986.

[Ders 85] Dershowitz, N. and Plaisted, D.A. "Equational programming" in Machine Intelligence, D.Michie, J.E. Hayes and J. Richards, eds.,1986.

[Ders 88] Dershowitz, N. and Okada, M.,"Conditional Equational Programming and the Theory of Conditional Term Rewriting" in Proc. of FGCS 88 , ed. by ICOT. 1988.

[DG89] Darlington, J, Yi-ke Guo "Narrowing and unification in Functional Programming" Proc.of RTA-89, LNCS 355, Apr.1989.

[Fay79] Fay, M. J., "First-order Unification in an Equational Theory " in 4th Workshop on Automated Deduction, 1979.

[Giov86] Giovannetti, E., Levi, G. and Moiso,C. and Palamidessi,C. Kernel LEAF: "An experimental logic plus functional language-its syntax, semantics and computational model," ESPRIT Project 415, Second year report 1986.

[Gogen80] Goguen,J.A."How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type implementations," Proc. of the First International Conference on Automated Deduction, July 1980.

[Höh88] Höhfeld, M. "Ein Schema für constraint-basierte relational Wissensbanken" Ph.D thesis,University of Kaiserslautern, FRG,Oct.1988. (in German)

[Huss88] Hussmann,H,"Nichtdeterministische Algebraische Spezifikationen (in German), Ph.D. thesis, Univ. of Passau, 1988.

[JaLaMa86] Jaffar, J., Lassez, J. and Maher, M. " Constraint Logic Programming" in Proc. of. 14th ACM symp. POPL, 1987.

[JaPaPi89] Jagadeesan, R., Panangaden, P. and Pingali, K." A Fully Abstract Semantics for a Functional Language with Logic Variables" TR 89-969 Dept. of Computer Science, Cornell University, May,1989.

[JaPl87] Jayaraman, B. and Plaisted, D.A.,"Functional Programming with Sets" in Proc. 3nd Int. Conf. on Functional LAnguages and Computer Architecture, Boston, Oct.1987.

[JaSi 86] Jayaraman, B. and Silberman,F., "Equations, Sets and Reduction Semantics for Functional and Logic Programming" in Proc. of ACM Conf. on LISP and Functional Programming, Aug.1986.

[Kieb 87] Kieburtz, R.B." Function + Logic in Theory and Practice" Draft manuscript, Feb.1987.

[Lind 85] Lindstrom, G. " Functional Programming and Logical Variable" in Proc. 20th ACM Symp. on POPL, New Orleans, 1985.

[Lloy84] Lloyd, J. W., " Foundations of Logic Programming" Springer-Verlag, 1984.

[Lock88] Lock, H.C.R." Guarded Term ML- A Functional Logic Programming Language" Proc. of Workshop on the Implementation of Lazy Functional Languages, Aug.1988.

[Reddy86] Reddy, U.S. " On the Relationship between Logic and Functional Languages" in Logic Programming: Functions, Relations and Equations, ed. Doug Degroot and G. Lindstrom, Prentice-Hall , 1986.

[Turn85] Turner, D.A. "Miranda: A Non-strict Functional Language with Polymorphic Types" In Proc. of Functional Programming Languages and Computer Architecture, ACM, 1985.

[Stee 80] Steele, G.L. " The Definition and Implementation of a Computer Programming Language Based on Constraints" Ph.D Thesis, M.I.T. AI- TR 595, 1980.

[While88] While, R.L. " Behavioral Aspects of Term Rewriting System" Ph.D. Thesis, Dept. of Computing, Imperial College, 1988.