A New Perspective on Integrating Functional and Logic Languages

John Darlington Yi-ke Guo Helen Pull
Department of Computing
Imperial College, University of London
180 Queen's Gate London SW7 2BZ U.K.
E-mail: jd, yg, hmp@doc.ic.ac.uk

February 1992

Abstract

Traditionally the integration of functional and logic languages is performed by attempting to integrate their semantic logics in some way. Many languages have been developed by taking this approach, but none manages to exploit fully the programming features of both functional and logic languages and provide a smooth integration of the two paradigms. We propose that improved integrated systems can be constructed by taking a broader view of the underlying semantics of logic programming. A novel integrated language paradigm, Definitional Constraint Programming (DCP), is proposed. DCP generalises constraint logic programming by admitting user-defined functions via a purely functional subsystem and enhances it with the power to solve constraints over functional programs. This constraint approach to integration results in a homogeneous unified system in which functional and logic programming features are combined naturally.

1 Introduction

During the past ten years the integration of functional and logic programming languages has attracted much research. An extensive survey and classification of their results can be found in [GLDD90]. Traditionally this integration is performed by attempting to integrate the respective semantic logics of functional and logic languages in some way, resulting in a "super logic language". The conventional understanding is that a logic program defines a logical theory and computation is attempting to prove that a query is a logical consequence of this theory. Taking this view, integration is regarded as enhancing the original logic to cope with functional programming features and results in a new logic programming system. In section 2 we survey the main results of this approach. It seems to us that this approach fails to deliver all the features of both functional and logic programming. The main source of inadequacy appears to stem from the respective "intended semantics" assumed for logic and functional languages. It is this intended semantics which we question, motivating our search for a new way of approaching the problem of integrating functional and logic languages.

We show in later sections that if we regard functional programming as defining a higher-order value space, we can extend the conventional constraint logic programming (CLP) framework by using a functional programming language to define the domain over which relations are defined. Thus we combine functional programming with a general CLP framework rather than with the conventional Prolog-like system. We call the resulting language paradigm **Definitional Constraint Programming (DCP)**. We claim that DCP provides a uniform and elegant integration of functional, constraint and logic programming, while preserving faithfully the essence of each of these language paradigms.

In section 3, constraint systems and constraint programming are investigated at a very general level. A constraint logic programming model is then presented in section 4 as a particular constraint programming paradigm. Section 5 presents constraint functional programming (CFP) as a framework which superimposes a solving capability on the functional programming paradigm. The definitional constraint programming paradigm is developed in section 6. We discuss future work in section **??** and make some concluding comments in section **??**.

2 Background and Motivation

From the traditional view of logic programming, integrating functional and logic languages is viewed as enhancing the original logic to cope with functional programming features. Most approaches take first-order equational logic as the semantic logic of functional languages and combine it with Horn clause logic. A comprehensive presentation of the theory of Horn clause logic with equality may be found in [GM87] and [Yuk88]. This shows that for every theory in Horn clause logic with equality, its initial model (called the least Herbrand E-model in [Yuk88], and the least Herbrand model in [Sny90]) always exists. Crucially, the initial model is the intended model of a logic programming system, since, according to the Herbrand theorem, the model is complete with respect to solving a query. For a Horn clause with equality program Γ and a

query $\exists x_1, \ldots, x_n\ A_1, \ldots, A_n$ where A_i is an atom or equation, a computational model must verify $\Gamma \models \exists x_1, \ldots, x_n A_1, \ldots, A_n$ by computing an answer substitution θ such that $\Gamma \models \forall(\theta A_1 \wedge \ldots \wedge \theta A_n)$. Such models integrate SLD-resolution with some form of equational deduction such as paramodulation. A complete computational model was proposed recently by Snyder et. al. [Sny90] as a goal directed inference system. Systems which aim to support the full power of Horn clause logic with equality include Eqlog [GM84], which exploits fully the order-sorted variation of the logic, SLOG [Fri85] in which a completion procedure is used as the computational model, and Yukawa's system [Yuk88] which uses an explicit axiomatization of equality.

The computational difficulties of constructing a practical programming language based on the full Horn clause logic with equality leads us to conclude that this approach is not appropriate. Alternative languages overcome these problems by imposing syntactic and semantic restrictions on the paradigm. They all aim either to restrict the use of, or to weaken, defined equality. An example of the first approach is Jaffer and Lassez's **Logic Programming Scheme** [JL86], in which the equality part of a program is defined separately from predicate definitions. A program uses a first-order equational sublanguage to define abstract data types over which a definite clause subprogram is imposed. Operational models are based on SLD-resolution together with an E-unification procedure which solves equations over the equality defined by the equational subprogram.

Another way to restrict the computational explosiveness of general equational deduction is to use equational clauses as directed rewrite rules. A full discussion may be found in [DO88]. Narrowing [Hul80] (resp. conditional narrowing [DO88]) is employed to solve equations in a rewriting system (resp. conditional rewriting system). Many languages have been developed along this line, e.g. RITE [DP86], K-Leaf [EGP86]. They represent enhanced Prolog systems in which a "rewrite" relation is defined over the Herband space. Syntactic restrictions guarantee the confluency of this rewrite relation so that equational logic can mimic first order functional programming. In the case of K-Leaf, the Herbrand space is enhanced to include partial terms, thus the lazy evaluation of functional languages may be modeled.

These endeavours have led to the development of several very successful languages and have significantly enriched the state of the art of declarative language design, semantics and implementation. However, we believe that the benefits of this combination are arguable and question how much is gained by enhancing a first-order logic by weakening a higher-order logic. Moreover, even with only first-order equational logic added, the inefficiencies of equational deduction mean that the resulting system is far from practical. This approach to language integration results in a sophisticated theorem prover, which we find unsatisfactory. We suggest, therefore, some fundamental rethinking on the purpose of integrating functional and logic languages.

In fact, the conventional assumption that a logic program defines a logical theory has been criticized in many circumstances because: "there is no reference to the models that the theory is a linguistic device for" [Mes89]. A logical theory may have many models, however when we are programming we always have a particular intended model in mind. This alternative school of thought regards a program as a linguistic description of the intended model; but the model itself is primary. For a Horn clause program,

its least Herbrand model is taken as the intended model. Therefore, if a program is regarded as a linguistic description of this model, the canonical denotation of a program is not a first-order theory but a set of relations over the Herbrand space. This view of logic programming has also been taken by researchers wishing to extend Prolog-like systems. Hagiya and Sakurai [MT84] present a formal system for logic programming based on the theory of iterative inductive definitions. A similar approach is taken by Hallnas and Schroeder-Heister to develop the framework of **General Horn Clause Programming** [AEHK89]. Paulson and Smith proposed an integrated system in which a logic subprogram is regarded as an inductive definition of relations [PS89].

This definitional view of logic programming suggests the flexibility to define Horn clauses over arbitrary domains. Relations become constraints over the domain of discourse, which coincides with the general framework of **Constraint Logic Programming** [Smo89]. In this paper, we take this idea one step further by using a functional programming language to define the domain over which relations are defined. A novel definitional constraint programming system is induced in which functions and relations are used together to define constraint systems.

3 Constraint Programming

In this section, we present a framework for constraint programming which has it origins in the seminal work of Steele [Ste80]. From the mathematical point of view, constraints are associated with well-studied domains in which some privileged predicates, such as equality and various forms of inequalities, are available. Relations formed by applying these predicates are regarded as constraints. A constraint may be regarded as a statement of properties of objects; its denotation is the set of objects which satisfy these properties. Therefore, constraints provide a succinct finite representation of possibly infinite sets of objects. We present a simple definition of constraint systems to capture these characteristics.

3.1 Constraint System

Definition 3.1 (Constraint System) *A constraint system is a tuple* $< \mathcal{A}, V, \Phi, I >$ *where*

- $\mathcal{A}$ *is a set of values called the* **domain** *of the system.*
- V *is a set of* **variables**.
- Φ *is a set of* **constraints**.
  We define an $\mathcal{A}$*-valuation as a mapping* $V \rightarrow \mathcal{A}$*, and* $\mathcal{VA}$ *as the set of all* $\mathcal{A}$*-valuations. A computable function* $\mathcal{V}$ *is used to assign to every constraint* ϕ *a finite set* $\mathcal{V}(\phi)$ *of variables, which are the variables constrained by* ϕ*.* $Val_{\mathcal{A}}$ *denotes the set of all* $\mathcal{A}$*-valuations.*
- I *is an interpretation which consists of a solution mapping* $[\![]\!]^I$*, mapping every basic constraint* ϕ *to* $[\![\phi]\!]^I$ *, a set of* $\mathcal{A}$*-valuations called the* **solutions** *of* ϕ *and* $[\![]\!]^I$

is **solution closed** *in the sense that*

$$\forall \alpha \in [\![\phi]\!]^I \; \forall x \in \mathcal{V}(\phi) \;\; \beta(x) = \alpha(x) \Rightarrow \beta \in [\![\phi]\!]^I$$

We now present some examples of constraint systems. The most familiar constraint system in the context of programming languages is perhaps the Herbrand system which is a constraint system over finite labelled trees.

Example 3.1.1 (Herbrand System) *Let Σ be a set of ranked signatures of function symbols and V be a set of constant symbols treated as variables. $T(\Sigma)$ is the ground term algebra consisting of the smallest set of inductively generated Σ-terms. A Herbrand system is a constraint system $< T(\Sigma), V, \Phi, I >$ where Φ consists of all term equations of the form $t_1 = t_2$ for $t_1, t_2 \in T(\Sigma, V)$, where $T(\Sigma, V)$ is the free term algebra, and $[\![t_1 = t_2]\!]^I = \{\alpha \mid \alpha\, t_1 \equiv \alpha\, t_2\}$ where $\equiv$ denotes the identity of two terms.*

Example 3.1.2 (Herbrand E-System) *Let Σ, V be as above and E an equational theory over $T(\Sigma, V)$. Then $T(\Sigma)/E$ denotes the quotient term algebra consisting of the finest Σ-congruences over $T(\Sigma)$ generated by E. The constraint system $< T(\Sigma)/E, V, \Phi, I >$ is called the Herbrand E-System where Φ consists of all term equations of the form $t_1 = t_2$ for $t_1, t_2 \in T(\Sigma, V)$ and $[\![t_1 = t_2]\!]^I = \{[\alpha]_E \mid [\alpha\, t_1]_E = [\alpha\, t_2]_E\}$, where $[t]_E$ stands for the equivalence class of t in $T(\Sigma)$ and $[\alpha]_E : V \rightarrow T(\Sigma)/E$ stands for the corresponding equivalence class of ground term substitutions $\alpha : V \rightarrow T(\Sigma)$.*

Constraint systems on various term structures can be regarded as cases of the following general definition of an algebraic constraint system.

Example 3.1.3 (Algebraic Constraint System) *Let A be an algebra equipped with a set of operators Σ and a set of predicates Π. Then the algebra is associated with a constraint system $S_A : <| A |, V, \Phi, I >$ where $| A |$ is the carrier of the algebra and evey constraint in Φ is of the form $p(e_1, \ldots, e_n)$ where every e_i is an A-expression and $p \in \Pi$ is an n-ary predicate in the algebra. $[\![p(e_1, \ldots, e_n)]\!]^I = \{\alpha \mid A, \alpha \models p(e_1, \ldots, e_n)\}$. Examples of algebraic constraints are constraints over term algebras, constraints over arithmetic expressions and constraint systems in boolean algebra.*

Following the idea of associating constraint systems with algebras, predicate logic can be viewed from the constraint system perspective.

Example 3.1.4 (Predicate Logic) *Suppose Σ is the first order signature of symbols, the well-formed Σ-formula are constraints, $\mathcal{V}(\phi) \subseteq V$ are free variables in ϕ with V the set of free (unquantified) variables. $\mathcal{A}$ is given by a Σ- structure (algebra) over which symbols are interpreted. With respect to a particular interpretation, I can be given as $[\![\phi]\!]^I = \{\alpha \mid \mathcal{A}, \alpha \models \phi\}$.*

For any constraint system, the solution of a constraint c can be restricted to a set of variables in c. Given a finite set of variables $W \subseteq_f V$, a valuation with respect to W becomes a partial mapping defined as follows :

$$\alpha_{|W}(x) = \begin{cases} \alpha(x) & x \in W \\ \bot & \text{otherwise} \end{cases}$$

The solutions of a constraint ϕ with respect to W are defined as:

$$[\![\phi]\!]^I_{|W} := \{\alpha_{|W} \mid \alpha \in [\![\phi]\!]^I\}$$

A constraint ϕ is **consistent** in a constraint system iff $[\![\phi]\!]^I \neq \emptyset$. A consistent constraint $\phi \in \Phi$ is **valid** iff $[\![\phi]\!]^I = \mathcal{VA}$. We use the word `true` to denote a valid constraint and `false` to denote an inconsistent constraint. Given a set of constraints, the set inclusion relation of solutions introduces a preorder over constraints that reflects the richness of the information they possess. A constraint ϕ_1 is a W-refinement of a constraint ϕ_2, $\phi_2 \leq_{|W} \phi_1$, iff $[\![\phi_1]\!]^I_{|W} \subseteq [\![\phi_2]\!]^I_{|W}$. ϕ_1 is a refinement of ϕ_2, $\phi_2 \leq \phi_1$, iff $[\![\phi_1]\!]^I \subseteq [\![\phi_2]\!]^I$. The preorder introduces an equivalence relation between constraints. A constraint ϕ is equivalent to a constraint ϕ', denoted $\phi \equiv \phi'$ iff $[\![\phi_1]\!]^I = [\![\phi_2]\!]^I$.

We consider some fundamental operations over constraints.

Definition 3.2 *Let ϕ_1, ϕ_2 be two constraints. Then their conjunction, $\phi_1 \wedge \phi_2$, is a constraint with $\mathcal{V}(\phi_1 \wedge \phi_2) = \mathcal{V}(\phi_1) \cup \mathcal{V}(\phi_2)$ and $[\![\phi_1 \wedge \phi_2]\!]^{\mathcal{I}} = [\![\phi_1]\!]^{\mathcal{I}} \cap [\![\phi_2]\!]^{\mathcal{I}}$; the constraint implication, $\phi_1 \rightarrow \phi_2$, is a constraint with $\mathcal{V}(\phi_1 \rightarrow \phi_2) = \mathcal{V}(\phi_1) \cup \mathcal{V}(\phi_2)$ and $[\![\phi_1 \rightarrow \phi_2]\!]^{\mathcal{I}} = \{\mathcal{VA} \perp [\![\phi_1]\!]^{\mathcal{I}}\} \cup [\![\phi_2]\!]^{\mathcal{I}}$.*

The definition of binary constraint conjunction can be extended to the conjunction of a set of constraints. A finite set of constraints is called a **goal** when it is interpreted as the conjunction of all its element constraints. A constraint implication $\phi_1 \rightarrow \phi_2$ is always valid whenever $[\![\phi_1]\!]^{\mathcal{I}} \subseteq [\![\phi_2]\!]^{\mathcal{I}}$ in which case we say that ϕ_1 entails ϕ_2, denoted $\phi_1 \vdash \phi_2$. It is obvious that $\phi_1 \vdash \phi_2 \iff \phi_2 \leq \phi_1$.

Definition 3.3 *If ϕ is a constraint and $x \in \mathcal{V}(\phi)$, then the existential quantification, $\exists x.\phi$, is a constraint with $\mathcal{V}(\exists x.\phi) = \mathcal{V}(\phi) \perp \{x\}$ and $[\![\exists x.\phi]\!]^{\mathcal{I}} = \{\alpha \in \mathcal{VA} \mid \exists \beta \in [\![\phi]\!]^{\mathcal{I}}, \alpha_{\mathcal{V}(\phi)-x} = \beta_{\mathcal{V}(\phi)-x}\}$; the negation of ϕ, $\neg\phi$, is a constraint with $\mathcal{V}\neg\phi = \mathcal{V}\phi$ and $[\![\neg\phi]\!]^I = \mathcal{VA} \perp [\![\phi]\!]^I$.*

Existential quantification provides a means of hiding, by projecting away, information about quantified variables. A constraint system is said to be closed with respect to an operator iff the constraint obtained by applying the operator is always in the system.

3.2 Constraint Solving

The computational task of a constraint system is to solve constraints. This is a constructive procedure which not only verifies that the solution set is non-empty, but transforms it to an equivalent, more informative form, from which solutions are easily derived. Such a form is called a **solved form**. As suggested by Smolka in [Smo91], constraint solving can be modeled by a rewriting system which simplifies a constraint to its equivalent solved form. Since, in the programming context, we are interested in solving goals, rewriting is applied to a set (more precisely, multiset) of constraints. Therefore, to express constraint solving in terms of rewriting we use multiset transformation systems.

Definition 3.4 (Constraint Solver) *A constraint simplification rule is a multiset transformation rule $G \rightarrow G'$ where G, G' are multisets of constraints (goals) such that $[\![G']\!]^I \subseteq [\![G]\!]^I$. A constraint solver C is a multiset transformation system containing a set of constraint simplification rules which is* **solution preserving**, *i.e. for n simplification rules of the form $G \rightarrow G'_i$ with the same left hand side M (up to renaming), we have :*

$$[\![G]\!]^I = \bigcup_{i=1}^{n} [\![G'_i]\!]^I$$

We call the relation $\stackrel{c}{\rightarrow}$ a one step simplification and $\perp \stackrel{c}{\rightarrow}^*$ a simplification derivation. A solved form of a goal G is a goal G' such that G' is a normal form with respect to the constraint solver.

The set $\mathcal{SF}_G$ of all solved forms of a goal G is complete iff

$$\forall \alpha \in [\![G]\!]^I . \exists G' \in \mathcal{SF}_G \text{ such that } \alpha \in [\![G']\!]^I$$

For a one step simplification $M \rightarrow M'$, it is obvious that simplification is sound.

Lemma 3.4.1 (Soundness of Simplification) *For one step simplification $G \stackrel{c}{\rightarrow} G', [\![G']\!]^I \subseteq [\![G]\!]^I$.*

The following proposition is also straightforward.

Lemma 3.4.2 *For any goal G which is not in normal form and $\alpha \in [\![G]\!]^I$, there exists a one step simplification $G \stackrel{c}{\rightarrow} G'$ such that $\alpha \in [\![G']\!]^I$.*

To model precisely the idea of simplification and its completeness, the familiar methodology of term rewriting systems is adapted. We require a complexity measure of a goal G with respect to a solution α, $|(G, \alpha)|$. We say that a constraint solver is **well-founded** iff

$$\forall \alpha \in [\![G']\!]^I \; G \perp \stackrel{c}{\rightarrow}^* G' \Rightarrow |(G, \alpha)| \geq |(G', \alpha)|$$

For any well-founded solver, a solved form is always reachable for a consistent goal G for a particular solution α. Therefore, it is always possible to enumerate a simplification derivation such that $G \perp \stackrel{c}{\rightarrow}^* G'$ with G' in solved form and $\alpha \in [\![G']\!]^I$.

Lemma 3.4.3 (Completeness of Simplification) *If a constraint solver $\mathcal{C}$ is well-founded, then for every consistent goal G, there is a set, $\mathcal{SF}_G$, which is the complete set of solved forms of G.*

The well-foundness of a constraint solver does not suggest that the complete set of solved forms for a goal is finite. It would be helpful to consider only finite sets of solved forms. For this we need the notion of compactness of constraint systems.

Definition 3.5 (Compactness) *A constraint system is compact iff for every finite set of constraints G :*

$$G \vdash G_1 \iff \exists G_2 \subseteq_f G_1 \text{ such that } G \vdash G_2$$

For any compact constraint system, a stronger completeness condition holds for any well-founded solver.

Lemma 3.5.1 *Let S be a compact constraint system and $\mathcal{C}$ be a solver for S. If $\mathcal{C}$ is well-founded, for any goal G, G' and $G \vdash G'$, there are a finite number of derivations $G' \perp \xrightarrow{c}{}^{*} G'_i$ such that G'_i is in solved form and $G \vdash \bigvee_{i=1}^{m} G'_i$.*

This lemma shows that, in a compact constraint system, any information contained by a goal constraint can always be processed by a finite amount of computation.

A constraint solver is deterministic when the simplification system is confluent. A simplification rule is deterministic in the solver if no other rule in the system has the same left hand side. For a well-founded deterministic solver, a consistent goal has a unique solved form. Constraint solving by a non-deterministic solver can be regarded as a reduction procedure which simplifies a disjunction of goals by rewriting it into an equivalent one. A constraint solver is terminating iff there is no infinite simplification derivation $G \rightarrow G_1 \rightarrow \ldots$. A well-founded constraint solver is decidable iff any unsatisfiable constraint can be simplified to `false`. Thus, a complete constraint solver is a decision procedure for the satisfiability of constraints.

3.3 Constraint Programming

Constraint programming is a declarative programming paradigm in which the task of programming is to define a constraint system and the task of computation is to solve the constraints. Therefore, the declarative semantics of a constraint program is given by determining the domain of discourse and defining the denotation of each constraint as its solution set. Its operational semantics is given by the constraint solver of the system which can be presented as a rewriting system which must be sound with respect to the declarative semantics and preferably complete. A sufficient condition for completeness is well-foundness of the solver. This notion of constraint programming is a generalization of the approach of Steele [Ste80] and Lassez [LM89]. Here the constraint system is assumed to be "built-in" and therefore, "programming" simply means imposing constraints.

When designing a constraint programming language it is essential to develop a systematic way to define constraints and a generic way to construct a solver for each defined constraint system. We give two examples of this generalized definition of constraint programming: Horn Clause Logic Programming and Equational Logic Programming.

Example 3.5.1 (Horn Clause Logic Programming) *A Horn clause program Γ defines a constraint system $< T(\Sigma), V, \Phi, I >$ where $T(\Sigma)$ is the ground term algebra for the signature Σ of function symbols (Herbrand space) and Φ consists of all positive literals and is closed under renaming, conjunction and existential quantification. I interprets constraints (defined predicates) as relations in the least Herbrand model M_Γ of the program :*

$$[\![p(t_1, \ldots, t_n)]\!]^I = \{\alpha : V \rightarrow T(\Sigma) \mid M_\Gamma, \alpha \models p(\alpha\, t_1, \ldots, \alpha\, t_n)\}$$

SLD resolution is a well-founded constraint solver which simplifies each consistent goal G into a disjunction of idempotent substitution equations: $\exists X. \bigvee_i S_i$.

This view of Horn clause logic programming is consistent with its traditional presentation. The major divergence is that we take the definitional view of logic programs. In another words, it is a linguistic specification of the intended model of the program, the least Herbrand model. This divergence results in some subtle differences in the properties of programs. For example, the completeness condition of constraint solving may not hold for all models of a program. Therefore it is not true that $\Gamma \models G \iff \bigvee_i S_i$ in the above example. To get this result a completion procedure must be applied to programs.

Example 3.5.2 (Equational Logic Programming) *An equational program E is a constraint system for solving equations in the quotient term algebra which it defines. A general E-unification procedure is its constraint solver. Following Gallier and Synder's result [Sny90], such a procedure exists and can be represented as multiset transformation system. Moreover, it is also well-founded. Therefore, it is a complete solver which simplifies an equational goal to a (possibly infinite) set of idempotent substitution equations.*

These two logic programming systems show two different ways to construct constraint systems in terms of logical formulas. A constraint system may be defined by a Horn clause logic program using recursive definition rules to define constraints over a fixed underlying domain. By contrast, in the case of equational logic programming, the form of constraints is fixed as equations over terms. An equational logic program forms a Herbrand E-system by defining the domain of discourse along with the interpretation of constraints (i.e. the equality in the domain). The former approach may be seen as a "relational extension" of a basic constraint system comprising the predefined fixed domain of discourse together with some "built in" constraints. In section 4, a systematic framework is constructed for such an extension. On the other hand, an equational logic program can be understood as defining an abstract data type with equations as constraints. This method of defining constraint systems may be called "domain construction" for some fixed constraint relation. In section 5, we propose a way to use functional programs to define the domain of discourse for solving constraints.

These two approaches may be combined to form a powerful constraint programming system in which both the domain of discourse and constraint relations are user-definable. The logic programming scheme [JL87], in which a program is regarded as a relational extension of the Herbrand E-constraint system defined by the equational subprogram, takes this route, although this was not the original semantics of the scheme. We believe that the constraint programming perspective provides a simpler and more intuitive semantic treatment of the scheme. Moreover, if we instantiate the underlying constraint system of the general CLP framework by constraint systems constructed over functional programs we have a powerful, general purpose, definitional constraint programming model which unifies functional and logic programming. This is the main result derived from our generalized view of constraint programming.

4 Constraint Logic Programming

As mentioned in the previous section, from the definitional view of logic programming, a constraint system can easily be integrated into a logic programming system. The

resulting **constraint logic programming** system is a definitional logic system which allows a predefined underlying constraint system to be extended by defining relations as new constraints. This formalism, which was first proposed by Höhfeld and Smolka in [Smo89], is more general than the proposal of Jaffer and Lassez [JL87], in which CLP is modelled within the traditional logic programming school. Many restrictions that are imposed on the underlying constraint system in the latter approach, such as the requirement that it is effectively axiomatized in first-order theory, are unnecessary within the definitional framework.

Let $C :< \mathcal{A}, \mathcal{V}, \Phi_c, I_c >$ be a constraint system closed under conjunction, renaming and existential quantification. Given a signature $\mathcal{R}$ as a family of user-defined predicates indexed by their arities, a constraint logic program Γ over C is a set of **constrained defining rules** of the form :

$$\mathcal{P} \leftarrow c_1, \ldots, c_j, B_1, \ldots, B_m$$

$\mathcal{P}$ is an $\mathcal{R}$-atom of form $p(x_1, \ldots, x_n)$ where $p \in \mathcal{R}_n$ is an n-ary user-defined predicate, $c_i \in \Phi_c$ and the B_i are atoms.

An interpretation I of Γ over C is defined by interpreting each predicate symbol $p \in \mathcal{R}$ as a relation p^I over $\mathcal{A}$. An ordering over interpretations is defined by the set inclusion of the relations. That is, for any interpretations I, I', $I \leq I'$ iff $p^I \subseteq p^{I'} \subseteq A^n$ for any n-ary predicate $p \in \mathcal{R}_n$. Thus, the set of all interpretations forms a complete lattice. With respect to a given interpretation, I, any $\mathcal{R}$-atom, $p(x_1, \ldots, x_n)$, can be regarded as a constraint whose solutions are given by :

$$[\![p(x_1, \ldots, x_n)]\!]^I = \{\alpha \mid (\alpha(x_1), \ldots, \alpha(x_n)) \in p^I\}$$

The solution set of a conjunction of atoms, a goal, is the intersection of the solutions of its elements. A defining rule $\mathcal{P} \leftarrow c_1, \ldots, c_j, B_1, \ldots, B_m$ is valid in an interpretation I iff $[\![p(x_1, \ldots, x_n)]\!]^I \supseteq \bigcap_{i=1}^{j} [\![c_i]\!]^{I_c} \cap \bigcap_{i=1}^{m} [\![B_i]\!]^I$

A model of Γ over C is an interpretation in which all rules of the program are valid. The set of all models of a CLP program is closed under intersection and union, therefore, the minimal model exists. We take the minimal model as the intended model of a program. This may be constructed by the standard iteration procedure which computes the inductive closure of relations generated by the clauses in Γ, given as:

$$\begin{aligned} I_0 &= \emptyset \\ I_{n+1} &= \bigcup_{p \in \Pi} p^{I_{n+1}} \end{aligned}$$

where $p^{I_{n+1}}$ defines the denotation of predicate p at the $n+1$th iteration step :

$$p^{I_{n+1}} = \{(\alpha(x_1), \ldots, \alpha(x_n)) \mid \alpha \in \bigcap_{i=1}^{j} [\![c_i]\!]^{I_c} \cap \bigcap_{i=1}^{m} [\![B_i]\!]^{I_n}\}$$

for all $p(x_1, \ldots, x_n) :\perp c_1, \ldots, c_j, B_1, \ldots, B_m \in \Gamma$. The sequence $I_0, I_1, \ldots, I_k \ldots$ is a chain in the interpretation lattice. The limit of the chain, $I_\Gamma = \cup_{n=0}^{\infty} I_n$, is the minimal model of the program Γ and can be computed as the least fixed point of the iteration procedure. This is presented as the following theorem:

Theorem 4.0.1 *Let $C :< \mathcal{A}, \mathcal{V}, \Phi_C, I_C >$ be a constraint system and Γ a constraint logic program over C. The sequence of interpretations I_n represents a chain in the complete lattice of interpretations of Γ. The limit of the chain is the minimal model of Γ over C.*

In this least model semantics of a CLP program the underlying constraint system is extended to a new constraint system via user-defined constraints. We call this a **relational extension** of a constraint system.

Definition 4.1 (Relational Extension) *Let* Γ *be a constraint logic program and* $\mathcal{R}$ *be the signature of user-defined predicates in* Γ. Γ *constructs the constraint system :*

$$\mathcal{R}(C) :< \mathcal{A}, \mathcal{V}, \Phi_c^{\mathcal{R}}, I_c^{\mathcal{R}} >$$

as a **relational extension** *of the underlying constraint system* $C :< \mathcal{A}, \mathcal{V}, \Phi_c, I_c >$ *by extending* Φ_c *to accommodate user-defined relations over* $\mathcal{A}$. *That is,* $\Phi_c^{\mathcal{R}} = \Phi_c \cup \Phi_{\mathcal{R}}$ *where* $\Phi_{\mathcal{R}}$ *contains all* $\mathcal{R}$*-atoms and*

$$\begin{array}{lcl} [\![\phi]\!]^{I_c^{\mathcal{R}}} & = & [\![\phi]\!]^{I_c} \\ [\![p(x_1, \ldots, x_n)]\!]^{I_c^{\mathcal{R}}} & = & \{\alpha \mid (\alpha(x_1), \ldots, \alpha(x_n)) \in p^{I_\Gamma}\} \end{array}$$

where I_Γ *is the minimal model of* Γ *over* C.

A solver for a relationally extended constraint system can be constructed by integrating SLD-resolution with the constraint solver of the underlying constraint system to give **constrained SLD-resolution**. Constrained SLD-resolution rewrites a goal of the form $G = G_c \cup G_{\mathcal{R}}$, where $G_{\mathcal{R}}$ is a finite subset of atoms in $\Phi_{\mathcal{R}}$ and G_c is a finite subset of Φ_c, to its solved forms. This model can be represented by the following multiset transformation rules.

Semantic Resolution: $\frac{G : \exists X < G_{\mathcal{R}} \cup \{p(s_1, \ldots, s_n)\} \cup G_c >}{G' : \exists X \cup Y < G_{\mathcal{R}} \cup \{B_1, \ldots, B_m\} \cup G_c \cup \{c_1, \ldots, c_k\} \cup \{x_1 = s_1, \ldots, x_n = s_n\} >}$

where $\forall Y.p(x_1 \ldots x_n) :\perp c_1, \ldots, c_k, B_1, \ldots, B_m$ is a variant of a clause in a program Γ.

Constraint Simplification: $\frac{G : \exists X < G_{\mathcal{R}} \cup G_c >}{G' : \exists X < G_{\mathcal{R}} \cup G_c' >}$

if $\exists X.G_c \perp\!\!\!\rightarrow_c \exists X.G_c'$ and $\perp\!\!\!\rightarrow_c$ is the simplification derivation realised by the solver in the underlying system.

Finite Failure: $\frac{G : \exists X G_{\mathcal{R}} \cup G_c}{\texttt{false}}$

if $\exists X.G_c \perp\!\!\!\rightarrow_c \texttt{false}$.

In this model, semantic resolution generates a new set of constraints whenever a particular program rule is applied. The unification component of SLD-resolution is replaced by solving a set of constraints via the underlying solver. Whenever it can be established that the set of constraints is unsolvable, finite failure results.

For example, the following CLP program [Col87]:

InCap ([], 0)
InCap (i:x, c) :⊥ *InCap* (x, 1.1*c - i)

can be used to compute a series of instalments which will repay capital borrowed at a 10% interest rate. The first rule states that there is no need to pay instalments to repay zero capital. The second rule states that the sequence of *N+1* instalments needed to repay capital c consists of an instalment i followed by the sequence of N instalments

which repay the capital increased by 10% interest but reduced by the instalment i. When we use the program to compute the value of m required to repay *\$1000* in the sequence *[m, 2m, 3m]*, we compute the solved form of the goal constraint: *InCap ([m, 2m, 3m], 1000)*. One execution sequence is illustrated below, in which $\rightarrow_R$ denotes a semantic resolution rewrite step:

$InCap\ ([m, 2m, 3m], 1000)$
$\rightarrow_R\ InCap\ (x,1.1c\text{-}i),\ x=[2m, 3m],\ i=m,\ c=1000$
$\rightarrow_R\ InCap(x',1.1c'\text{-}i'),\ x=i':x',\ c'=1.1c\text{-}i,$
$\quad x=[2m, 3m],\ i=m,\ c=1000$
$\rightarrow_c\ InCap\ (x',1.1c'\text{-}i'),\ i'=2m,$
$\quad x'=[3m], i=m,\ c'=1100\text{-}m$
$\rightarrow_R\ InCap(x'',1.1c''\text{-}i''),\ x'=i'':x'',1.1c'\text{-}\ i'=c''$
$\quad ,\ i'=2m,\ x'=[3m], i=m, c'=1100\text{-}m$
$\rightarrow_c\ InCap\ (x'',1.1c''\text{-}i''),\ x''=[],\ i''=3m, i'=2m,$
$\quad i=m, x'=[3m],\ c'=1100\text{-}m,\ c''=1210\text{-}3.1m$
$\rightarrow_R\ x''=[], 1.1c''\text{-}i''=0,\ i''=3m,\ i'=2m,\ i=m,$
$\quad x'=3m,\ c'=1100\text{-}m,\ c''=1210\text{-}3.1m$
$\rightarrow_c\ 1.1(1210\text{-}3.1m)=3m$
$\rightarrow_c\ m=207+413/641$

Constrained SLD-resolution is a sound solver for a relationally extended constraint system and, as proved in [Smo89], it is also well-founded. Therefore, any consistent goal can be simplified to a set of solved forms. Let $\Phi\downarrow\subseteq\Phi$ be the set of all solved forms for Φ. Then it is easy to show that $\Phi_c^{\mathcal{R}}\downarrow\subseteq\Phi_c\downarrow$. Therefore, from the completeness of the model, for any goal G with $\bigvee_{i\geq 1} G_c^i$ as its solved forms, given a goal $G'_c\subseteq_f\Phi_c$ containing only basic constraints, $G'_c\vdash G\Rightarrow G'_c\vdash\bigvee_{i\geq 1} G_c^i$. Moreover, if the underlying constraint system is compact, then $G'_c\vdash\bigvee_{i=1}^n G_c^i$ for some n, i.e. the model has the stronger completeness of section 3.2.

5 Constraint Functional Programming

Constraint functional programming (CFP) is characterized as functional programming, enhanced with the capability to solve constraints over the value space defined by a functional program. An intuitive construction of this language paradigm is presented below.

5.1 Informal CFP

A data type D in a functional program, Γ, can be associated with a constraint system C_D. C_D may contain privileged predicates over D. A CFP system may be formed to extend the constraint solver so that any D-valued expression, which may involve user-defined functions, can be admitted in constraints. A D-valued expression must be evaluated to its normal form with respect to Γ to enable the constraint solver to handle that value.

We give a simple example of this paradigm. We assume a constraint system over lists in which atomic constraints are equations asserting identity over finite lists. A unification algorithm is used as the basic solver for the system. Given a functional program defining the function $++$ which concatenates two lists and the function *length* which computes the length of a list:

data $[alpha] = [] \mid alpha : [alpha]$
functions
$++ :: [alpha] \times [alpha] \rightarrow [alpha]$
$length :: [alpha] \rightarrow Num$

$[] ++ z = z$
$(x{:}y) ++ z = x : (y{+}{+}z)$

$length\ [] = 0$
$length\ (x{:}y) = 1 + length\ y$

An extension to the basic solver may be used to solve the constraint:

$$l1 ++ l2 = [a1, a2, \ldots, an], length\ l1 = 10$$

to compute the first 10 elements of a the list $[a1, a2, ..., an]$. The solver must apply the function definitions of $++$ and *length* and must guess appropriate instances of the constrained variables. We will show that this procedure itself may be modelled by some new constraints generated during rule application.

Solving constraints over a functional program significantly enhances the expressive power of functional programs to incorporate logic programming features. This idea was central to the **absolute set abstraction** construct which was originally proposed in [DAP86, DG89] as a means to invoke constraint solving and collect solutions. Using the absolute set abstraction notation, the above constraint may be represented as the set-valued expression:

$$\{\ l1 \mid l1 ++ l2 = [a1, a2, \ldots, an], length\ l1 = 10\ \}$$

Reddy's proposal of "Functional Logic Programming" languages [Red86] also exploits this solving capability in functional programs. However, his description of functional logic programming as functional syntax with logic operational semantics fails to capture the essential semantic characteristics of the paradigm. The constraint programming approach, as we will show in the following, presents a concise semantical and operational model for the paradigm.

5.2 Functional Programming Languages

We assume a functional language that is strongly typed, employs a polymorphic type system and algebraic data types, and supports higher-order functions and lazy evaluation. Examples of such languages are Miranda [Tur85] and Haskell [Com90]. To investigate constraint solving we put aside the statical features of a functional language such as its type system, and concentrate on its dynamic semantics. We use a kernel functional language with recursion equation syntax for defining functions. We assume

variables ranged over by x and y, a special set of functional variables (identifiers) ranged over by f and g, constructors ranged over by d, constants ranged over by a and b, patterns ranged over by t and s and expressions ranged over by e. A pattern is assumed to be linear, i.e. having no repeated variables. Data terms comprise only constants, constructors and first-order variables. The following syntax defines this tiny functional language:

$$\begin{array}{lll} \mathit{Program} & ::= & \mathit{Decl}\ \texttt{in}\ \mathit{Exp} \\ \mathit{Decl} & ::= & ft = e \\ & & \mid \mathit{Decl}; \mathit{Decl} \\ \mathit{Exp} & ::= & x \mid a \mid e_1\, e_2 \mid e_1\ op\ e_2 \\ & & \mid\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ \mathit{Pattern} & := & x \mid a \mid dt_1, \ldots, t_n \end{array}$$

The language can be regarded as sugared λ-calculus and a program as a λ-expression. The program shown above is an instance of this formalism in which the data statement introduces a list structure with a nullary constructor [] and a binary constructor :, and functions *length* and ++ which are defined by recursion equations.

The semantics of a functional program is given in the standard way [Sco89]. The semantic domain D of the program is an algebraic CPO which is the minimal solution of the domain equation :

$$D = B_{\perp} + C(D) + D \to D$$

D contains the domain $B_{\perp}$ of basic types (real numbers, boolean values et. al. lifted by $\perp$ which denotes undefinedness), the domain $C(D)$ for constructed data structures which consists of partial terms ordered with respect to the monotonicity of constructors and the domain $D \to D$ of all continuous functions. A subdomain A of $C(D) : A = B_{\perp} + C(A)$ is distinguished as the domain of data terms in the language (which is defined by the eq-type of ML [Mil84]). We use $\mathcal{T}$ to denote all complete objects of A.

For a functional program, the semantic function $\mathcal{P}[\![]\!]$ computes the value of the program in terms of the function $\mathcal{D}[\![]\!] : \mathit{Decl} \to (\mathit{Var} \to D) \to (\mathit{Var} \to D)$ which maps function definitions to an environment which associates each function name with its denotation. The function $\mathcal{E}[\![]\!] : \mathit{Exp} \to (\mathit{Var} \to D) \to D$ maps an expression together with an environment $\eta : \mathit{Var} \to D$ (a D-valuation) to an element of D.

5.3 Evaluating Nonground Expressions

Conventional functional programming involves evaluating a ground expression to its unique normal form by taking a program as a rewriting system. To superimpose a solving capability on the functional programming paradigm, we consider first the extension of functional programming to handle non-ground expressions. The meaning of a non-ground expression is a set of values corresponding to every correctly typed instantiation of its free variables. Narrowing has been proposed as the operational model for computing all possible values of a nonground expressions [Red84]. In the theorem proving context, enumerating narrowing derivations provides a complete E-unification procedure for equational theories defined by convergent rewriting systems. This use of

narrowing must be refined for the functional programming context. Due to the lazyness of functional languages, only those narrowing derivations whose corresponding reduction derivations are lazy should be enumerated. This notion of **lazy narrowing** is mentioned by Reddy in [Red84]. A lazy narrowing procedure, **pattern-driven narrowing**, is proposed by Darlington and Guo in [DG89] for evaluating absolute set abstractions. A similar procedure was independently developed by You for constructor based equational programming systems [You88]. Here we present a lazy narrowing model following the constraint solving approach. The model is central to the CFP paradigm.

Consider reducing a non-ground expression of form fe by a defining rule $ft = e'$. The environment η should be enhanced to satisfy $\mathcal{E}[\![e]\!]\eta = \mathcal{E}[\![t]\!]\eta$, i.e. η is a solution of the **rewriting constraint** $e = t$. This equality is the so called semantic equality since it is determined by the identity of denotations of components. It is not even semidecidable since it involves verifying the equivalence of partial values. However, since in our problem t is always a linear pattern, a semidecidable solver exists.

Definition 5.1 *The solved form of a rewriting constraint $e = t$ is of the form $\{x_1 = t_1, \ldots, x_n = t_n, y_1 = e_1, \ldots, y_m = e_m\}$ where the $x_i \in \mathcal{V}(e)$ are output variables and the $y_i \in \mathcal{V}(t)$ are input variables. The equation set $\hat{\delta} : \{x_1 = t_1, \ldots, x_n = t_n\}$ is an output substitution equation and $\hat{\theta} : \{y_1 = e_1, \ldots, y_m = e_m\}$ is an input substitution equation. The substitutions δ and θ corresponding to $\hat{\delta}$ and $\hat{\theta}$ are called output substitutions and input substitutions respectively.*

The constraint solver presented below simplifies a rewriting constraint to its solved form. Solving a rewriting constraint realises the bidirectional parameter passing mechanism for narrowing an outmost function application. The algorithm is called **pattern-fitting** [DG89].

Substitution: $\{x = r\} \cup G \Rightarrow \{x = r\} \cup \rho G$
where $\rho = \{x \mapsto r\}$.

Decomposition: $\{de = dt\} \cup G \Rightarrow \{e = t\} \cup G$

Removing: $\{a = a\} \cup G \Rightarrow G$

Failure: $\{d_1 e_1 = d_2 e_2\} \cup G \Rightarrow$ `false`
if $d_1 \neq d_2$.

Constrained Narrowing: $\{fe = ds\} \cup G \Rightarrow \{r = ds, e = t\} \cup G$
where $ft = r \in \Gamma$

Lemma 5.1.1 *The pattern-fitting algorithm is a complete solver for simplifying a rewriting constraint to its solved form.*

For any rewriting constraint $e = t$, a solved form corresponds to a pattern-driven narrowing step $fe \leadsto_\delta \theta e'$ with respect to a defining rule $ft = e'$ where δ is the output substitution and θ is the input substitution associated with the solved form. A pattern-driven narrowing derivation is defined in a standard way by composing the output substitutions of each of its component steps. Note that a one step pattern-driven narrowing

derivation contains many narrowing steps due to the need to solve rewriting constraints. Each narrowing step is demand driven and affects an outermost function application. Therefore, we have the following theorem:

Theorem 5.1.1 *For any expression e and term t, if $e \leadsto_\delta t$, then the corresponding reduction derivation $\delta e \rightarrow^* t$ is always a lazy derivation. Such a reduction derivation is called a standard reduction in [Hue86]. Enumerating pattern-driven derivations is optimal and complete in the sense that any other derivation is subsumed by a pattern-driven derivation.*

We conclude that pattern-driven narrowing provides a realisation of lazy narrowing. Lazy narrowing extends functional programming with the capability to find for which values of variables in a nonground expression the expression evaluates to a given value. Thus, it introduces the essential solving feature to functional languages. However, on its own it is not enough because "built in" predicates may exist in functional languages, for example equality and various boolean valued primitive functions, for which a dedicated constraint solver is required. If we integrate lazy narrowing with a constraint solver over data terms, the solver is then extended to allow general expressions containing user-defined functions. Therefore, querying a functional program becomes possible. This enhanced functional programming framework may be formalized as the paradigm of constraint functional programming.

5.4 Formalizing CFP

We assume a constraint system $C_\mathcal{T} : (\mathcal{T}, V, \Phi_c, I_c)$ over first-order values, where V is the set of variables over first-order types and Φ_c are constraints consisting of privileged predicates $\mathcal{R}$. Computing the truth value of a ground relation of data terms with respect to $\mathcal{R}$ is decidable. Thus, a predicate ω in $\mathcal{R}$ can always correspond to a boolean valued function f_ω in the language. A functional program may be applied to $C_\mathcal{T}$. This introduces a new syntactic category in the functional program for constraints :

$$\mathit{Constraint} \quad ::= \quad \omega(e_1, \ldots, e_n) \mid \mathit{Constraint}, \mathit{Constraint}$$

where $\omega(x_1, \ldots, x_n) \in \Phi_c$. We use c to range over constraints.

Constraints in $C_\mathcal{T}$ are now enriched to admit general expressions defined by the functional program. A constraint system is **admissible** if it is closed under negation. In the following, we assume the underlying constraint system is admissable. A CFP program is an extension of a functional program with the syntax:

$$\mathit{Program} \quad ::= \quad \mathit{Decl}\ \texttt{in}\ e \mid \mathit{Decl}\ \texttt{in}\ c$$

The semantic function $\mathcal{C}[\![\,]\!] : \mathit{Constraint} \rightarrow \mathcal{P}(\mathit{Env})$ maps constraints to their solution sets:

$$\begin{array}{lcl} \mathcal{C}[\![c_1, c_2]\!] & = & \mathcal{C}[\![c_1]\!] \cap \mathcal{C}[\![c_2]\!] \\ \mathcal{C}[\![\omega(e_1, \ldots, e_n)]\!] & = & \{\eta_{|\cup \mathcal{V}(e_i)} \mid \mathcal{T} \models \omega(\mathcal{E}[\![e_1]\!]\eta, \ldots, \mathcal{E}[\![e_n]\!]\eta)\} \end{array}$$

This semantics reveals constraint solving over a functional language as "computing the environments" in which expressions, when evaluated, satisfy constraints.

The constraint solving mechanism is formed by integrating the solver of $C_{\mathcal{T}}$ with lazy narrowing, thus enhancing $C_{\mathcal{T}}$ to handle constraints in the more general universe constructed by a functional program. A scheme for such an integration is presented below. We use the pair (G, C) to represent a goal $G \cup C$ in which C contains rewriting constraints and G contains constraints from the underlying constraint system.

Constrained Narrowing: $\frac{(G \cup \{\omega(...,fe,...)\}, C)}{(G \cup \{\omega(...,r,...)\}, C \cup \{e=t\}}$)
where $ft = r \in \Gamma$.

Simplification 1: $\frac{(G,C)}{(G',C)}$
if $G \xrightarrow{c} G'$ where $\xrightarrow{c}$ is a simplification derivation computed by the under lying solver.

Simplification 2: $\frac{(G,C)}{(G,C')}$
if $C \xrightarrow{pt} C'$ where $\xrightarrow{pt}$ stands for a simplification derivation computed by the solver of rewriting constraints.

Failure: $\frac{(G,C)}{\texttt{false}}$
if $G \xrightarrow{c} \texttt{false}$ or $C \xrightarrow{pt} \texttt{false}$

Substitution 1: $\frac{(G, C \cup \{x=e\})}{(\rho\, G, C \cup \{x=e\})}$
where $\rho = \{x \mapsto e\}$ and $x \in \mathcal{V}(G)$ and $C \cup \{x = e\}$ is in solved form.

Substitution 2: $\frac{(G \cup \{x=t\}, C)}{(G, \rho\, C \cup \{x=t\})}$
where $\rho = \{x \mapsto t\}$ and $G \cup \{x = t\}$ is in solved form.

Positive Accumulating: $\frac{(G, C \cup \{f_\omega e = \texttt{true}\})}{(G \cup \{\omega(e)\}, C)}$
If $\omega(x) \in \Phi_c$.

Negative Accumulating: $\frac{(G, C \cup \{f_\omega e = \texttt{false}\})}{(G \cup \{\neg\omega(e)\}, C)}$
if $\omega(x) \in \Phi_c$ and the constraint system is admissable.

An initial goal takes the form $(G, \{\})$. Its solved form is of the form (G_n, C_n) where G_n is in solved form with respect to the underlying solver and $\mathcal{V}(G) \notin \mathcal{V}(C)$ and C are solved form rewriting constraints.

The soundness of lazy narrowing guarantees that the enhanced solver is sound. However, it is not in general complete because a functional program may define some boolean-valued functions which have no corresponding constraints in $C_{\mathcal{T}}$. This problem is similar to that of solving "hard constraints" in general constraint programming. Some ways exist to resolve this problem such as the "waiting-resuming" approach in which the solving of a hard constraint is delayed until its variables are sufficiently instantiated [JL87], or by defining special simplification rules for such constraints. However, for a program in which all boolean-valued functions are consistent with the underlying constraint system, the scheme provides a complete enhanced solver.

The scheme provides a generic model to enhance a constraint system to solve constraints in functional languages. In [Pul90], Pull uses unification on data terms as the underlying solver and combines it with lazy narrowing to solve equational constraints in lazy functional languages. In [JCGMRA91], a more general constraint system over data terms is adopted in which disunification is also exploited to deal with negative

equational constraints. This model can be regarded as an instantiation of the scheme by providing unification and disunification as the "built-in" solvers.

CFP represents a constraint programming system of the "domain construction" approach of section 3.3. This means that constraints appear only as computational goals; it is not possible to define new constraints in the system. However, the framework significantly enhances the expressive power of both functional programs and the basic constraint system. Moreover, since a CFP program provides a constraint system in which defined functions behave as operators in some algebra, it is perfectly reasonable to define relations over the system following the philosophy of general constraint logic programming. Therefore, CFP is a "building block" for deriving a fully integrated **Definitional Constraint Programming** system in which both constraints and the domain of discourse are user-definable.

6 Definitional Constraint Programming

We are now in a position to present a unified definitional constraint programming (DCP) framework. A DCP program defines a constraint system by defining its domain of discourse and constraints over this domain. As discussed above, CFP and CLP exhibit, respectively, the power to define domains, and the power to define constraints. Therefore we would expect the unification of these two paradigms to result in a full definitional constraint programming system.

We start by superimposing a functional program onto a privileged constraint system. As shown in the previous section, the functional program defining functions ++ and *length* can be queried to compute the initial segment of a given list. A further abstraction is possible if we take this CFP enriched constraint system as the underlying constraint system for a CLP language. Thus, CFP queries can be used to define relations as new constraints. For example we can define the relation *front*:

$$front\ (n, l, l1)\ \ :\!\perp\ \ \ l1\ ++\ l2\ =\ l,\ length\ \ l1\ =\ n$$

to compute the initial segment with length n of an input list l. This systematic integration of CFP and CLP results in a definitional constraint programming system and therefore, can be expressed by the formula $DCP = CLP(CFP)$.

It is straightforward to construct the semantic model of a DCP program. The semantics for its functional component are traditional functional language semantics. The intended model of the relational component is its least model. This may be constructed by computing all ground atoms generated by the program using the "bottom up" iterative procedure presented in theorem 4.0.1 and taking the functionally enhanced constraint system as the underlying constraint system. In terms of the semantic functions defined above the denotation of a defined predicate p in a program Γ can be computed by enumerating the inductive closure of Γ as follows :

$$\begin{aligned} p^0 &= \emptyset \\ p^{I_{n+1}} &= \{\alpha(x_1, \ldots, x_n) \mid \alpha \in \textstyle\bigcap_{i=1}^{n} \mathcal{C}[\![c_i]\!] \cap \bigcap_{j=1}^{m} [\![B_j]\!]^{I_{n+1}}\} \end{aligned}$$

for each $p(x_1, \ldots, x_n) :\!\perp\ c_1, \ldots, c_n, B_1, \ldots B_m \in \Gamma$. $[\![B]\!]^I$ maps B to all solutions of B under the interpretation I for the predicates in B. That is :

$$[\![p(e_1, \ldots, e_n)]\!]^I = \{\eta \mid (\mathcal{E}[\![e_1]\!]\eta, \ldots, \mathcal{E}[\![e_n]\!]\eta) \in p^I$$

Compared with other functional logic systems, this general notion of constraint satisfaction permits us, not only to define equational constraints over finite data terms, but also to introduce more general domain specific constraints. Moreover, partial objects as introduced by lazy functional programming are admissible for constraint solving in the system as approximations of complete objects. This gives uniform support for laziness in a fully integrated functional logic programming system.

The computational model of the DCP paradigm is simply the instantiation of the underlying constraint solver in constrained SLD-resolution to the CFP solver. Soundness and completeness are a direct result of the properties of these two components.

Clearly then, DCP represents a supersystem of both these paradigms. Both the CLP *InCap* program and the CFP query which computes the initial segment of a list are valid DCP programs and queries. Moreover, the expressive power of each of these individual paradigms is enhanced in the DCP framework. We will demonstrate this with reference to some programming examples.

The "built-in" solver manipulates only first-order objects. In any correctly-typed DCP program, a function-typed variable will never become a constrained variable. Thus, higher-order functional programming features safely inherit their intended use in functional computation without introducing computability problems. The following examples illustrate some of the attractive programming features of this rich language paradigm.

The quicksort algorithm is defined below as a relation which uses difference lists (which appear as pairs of lists *(x, y)*) to perform list concatenation in constant time. The partitioning of the input list is specified naturally as a function, while the ordering function is passed as an argument to the quicksort relation. Within the semantics of DCP, such a functional parameter can be treated as special constant in relation definitions. A primitive function *apply* is assumed which is responsible for the application of such function names to arguments.

```
functions
partition : (alpha → alpha → boolean) × alpha × [alpha]
          → ([alpha], [alpha])
relations
quicksort : (alpha → alpha → boolean) × [alpha]
          × ([alpha ],[alpha])

partition (f, n, m : l) = if f (n, m)
                  then (m : l1, l2) else (l1, m : l2)
                   where (l1, l2) = partition (f, n, l)
partition (f, n, []) = ([], [])

quicksort (f, n : l, (x, y)) :⊥
   partition (f, n, l) = (l1, l2),
      quicksort (f, l1, (x,  n : z)), quicksort (f, l2, (z, y))
quicksort (f, [], (x, x))
```

The relation *perms* below shows an interesting and highly declarative way of specifying the permutations problem in terms of constraints over applications of the list concatenation function $++$.

relations
$perms : [alpha] \times [alpha]$
$perms\ (a : l,\ (l1\ ++\ (a : l2))\ :\bot\ (l1\ ++\ l2) = perms\ l$

The final example shows how the recursive control constructs of higher-order functions may be used to solve problems in the relational component of a DCP language. We use a *reduce* function over lists, together with the "back substitution" technique familiar in logic programming, to find the minimal value in a list and propagate this value to all cells of the list. This is shown via the relation *propagatemin* below, which uses the standard list *reduce* function to find the minimum value, y, in the input list and construct a list, $l1$, which is isomorphic to the input list, in which each element is a logical variable x.

relations $propagatemin : [Int] \times [Int]$
$propogatemin\ l\ l1\ :\bot$
$reduce\ (f\ x,\ l,\ (MaxInt,\ nil)) = (y,\ l1\ ),\ x\ =\ y$
where $f\ z\ n\ (m,\ l2\ ) = (min\ (n,\ m\ ),\ z : l2\ )$

These examples show that as well as being a systematic and uniform integration of constraint, logic and functional programming with a sound semantics, the DCP paradigm displays a significant enhancement of programming expressive power over other integrated language systems. We believe that this pleasing outcome is a direct result of our strenuous effort to identify clearly the essential characteristics of the component language paradigms and to preserve them faithfully in the DCP language construction. We have defined a concrete DCP language, Falcon [GP91]. Many Falcon programming examples appear in [DGP91].

References

[AEHK89] M. Aronsson, L-H Eriksson, L. Hallnas, and P. Kreuger. A Survey of GCLA: A Definitional Approach to Logic Programming. In *Proc. of the International Workshop on Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science, Springer Verlag*. Springer, 1989.

[Col87] A. Colmerauer. Opening the Prolog III universe. *Byte, July*, 1987.

[Com90] Haskell Committee. Haskell: A non-strict, purely functional language. Technical report, Dept. of Computer Science, Yale University, April 1990.

[DAP86] J. Darlington, Field. A.J., and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 37–70. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[DG89] J. Darlington and Y.K. Guo. Narrowing and Unification in Functional Programming. In *Proc. of RTA 89*, pages 292–310, 1989.

[DGP91] J. Darlington, Y.K. Guo, and H. Pull. A new perspective on integrating functional and logic languages. Technical report, Dept. of Computing, Imperial College, December 1991.

[DO88] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *Proc. of the FGCS '88, ed. by ICOT*, 1988.

[DP86] N. Dershowitz and D.A. Plaisted. Equational programming. *Machine Intelligence (Mitchie,Hayes and Richards, eds.)*, 1986.

[EGP86] C. Moiso E. Giovannetti, G. Levi and C. Palmidessi. Kernel Leaf: An experimental logic plus functional language - its syntax, semantics and computational model. ESPRIT Project 415, Second Year Report, 1986.

[Fri85] Laurent Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceeding of the 2nd IEEE Symposium on Logic Programming, Boston*, 1985.

[GLDD90] Y. Guo, H. Lock, J. Darlington, and R. Dietrich. A classification for the integration of functional and logic languages. Technical report, Dept. of Computing, Imperial College and GMD Forchungsstelle an der Universitat Karlsruhe, March 1990. Deliverable for the ESPRIT Basic Research Action No.3147.

[GM84] Joseph A. Goguen and Jose Meseguer. Equality, types, modules, and (why not?) generics for logic programming. *Journal of Logic Programming*, 2:179–210, 1984.

[GM87] Joseph Goguen and Jose Meseguer. Models and equality for logical programming. In *Proc. of TAPSOFT 87*, volume 250 of *Lecture Notes in Computer Science, Springer Verlag*. Springer, 1987.

[GP91] Y.K. Guo and H. Pull. Falcon: Functional And Logic language with CONonstraints–language definition. Technical report, Dept. of Computing, Imperial College, February 1991.

[Hue86] G. Huet. Formal structure for computation and deduction. Technical report, Dept. off Computer Science, Carnegie-Mellon University, May 1986.

[Hul80] Jean-Marie Hullot. Canonical forms and unification. In *5th Conf. on Automated Deduction*. LNCS 87, 1980.

[JCGMRA91] M.T. Hortala-Gonzalez J Carlos Gonzalez-Moreno and Mario Rodriguez-Artalejo. A Functional Logic Language with Higher Order Logic Variiables. Technical Report, Dpto. de Informatica y Automatica UCM, 1991.

[JL86] Joxan Jaffar and Jean-Louis Lassez. Logical programming scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming*, pages 441–467. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Prod. of POPL 87*, pages 111–119, 1987.

[LM89] J-L. Lassez and K. McAloon. A constraint sequent calculus. Technical report, IBM T.J. Watson Research Center, 1989.

[Mes89] Jose Meseguer. General logics. Technical Report SRI-CSL-89-5, SRI International, March 1989.

[Mil84] Robin Milner. A proposal for Standard ML. In *ACM Conference on Lisp and Functional Programming*, 1984.

[MT84] M.Hagiya and T.Sakurai. Foundation of Logic Programming Based on Inductive Definition. *New Generation Computing*, 2(1), 1984.

[PS89] L.C. Paulson and A.W. Smith. Logic Programming, Functional Programming and Inductive Definitions. In *Proc. of the International Workshop on Extensions of Logic Programming*, volume 475 of *Lecture Notes in Computer Science, Springer Verlag*. Springer, 1989.

[Pul90] Helen M. Pull. *Equation Solving in Lazy Functional Languages*. PhD thesis, Dept. of Computing, Imperial College, University of London, November 1990.

[Red84] Uday S. Reddy. Narrowing As the Operational Semantics of Functional Languages. In *Proc. of Intern. Symp. Logic Prog. IEEE'*. IEEE, 1984.

[Red86] Uday S. Reddy. Functional Logic Languages, Part 1. In J.H. Fasel and R.M. Keller, editors, *Poceedings of a Workshop on Graph Reduction, Santa Fee*, number 279 in Lecture Notes in Computer Science, Springer Verlag, pages 401–425, 1986.

[Sco89] Dana Scott. Semantic domains and denotational semantics. Lecture Notes of the International Summer School on Logic, Algebra and Computation, Marktoberdorf, 1989. to be published in LNCS series by Springer Verlag.

[Smo89] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Vom Fachbereich Informatik der Universitat Kaiserlautern, May 1989.

[Smo91] Gert Smolka. Residuation and Guarded Rules for Constraint Logic Programming. Research Report RR-91-13 DFKI, 1991.

[Sny90] W. Snyder. *The Theory of General Unification*. Birkhauser, Boston, 1990.

[Ste80] G.L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD thesis, M.I.T. AI-TR 595, 1980.

[Tur85] David A. Turner. Miranda: A non-strict language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*, LNCS 201, pages 1–16, 1985.

[You88] Jia-Huai You. Outer Narrowing for Equational Theories Based on Constructors. In Timo Lepistö and Arto Salomaa, editors, *15th Int. Colloqium on Automata, Languages and Programming*, LNCS 317, pages 727–741, 1988.

[Yuk88] K. Yukawa. Applicative logic programming. Technical Report LP-5, Logic programming Laboratory, June 1988.