TR-662

Constraint Logic Programming and the Unification of Information

by

K.Mukai

June, 1991

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Constraint Logic Programming and the Unification of Information

Kuniaki Mukai

Institute for New Generation Computer Technology
and
Mitsubishi Electric Corporation

A doctor thesis submitted to
Department of Computer Science, Faculty of Engineering
Tokyo Institute of Technology
2-12-1 Ōokayama, Meguro-ku, Tokyo 152

April 1991

Abstract

Feature structure is a record-like structure of information used as a partial description of objects. The constraint logic programming scheme, CLP(X) (Jaffar and Lassez [39]), is a class of logic programming languages based on constraint solving, which is a generalization of the standard logic programming, where X is the parameter of the scheme. It has been an open problem to instantiate X with the feature structure to obtain a constraint logic programming which manipulates the partial information. This thesis reports the following three contributions to the problem.

● We propose a logic programming language CIL and show applications for linguistic analysis. CIL is an extension of Prolog with infinite record structures called PST (*Partially Specified Terms*) which is a generalization of the standard terms and record structures. Also PST is a generalization of Colmerauer's infinite trees. Besides PST, CIL has useful aspects for linguistics analysis. Among them are term constraints, arithmetic constraints, and Boolean constraints. Also CIL has a one-way unification as a primitive for the user to define his own constraints. CIL build-ins complex indeterminates, which are basic objects proposed in situation semantics by Barwise and Perry as building blocks for information and situations[1]. The study of CIL started around 1983. Several versions of CIL has been implemented by the author. The earliest version was implemented in Edinburgh Prolog on DEC-2060 machine. Then it was transplanted on PSI-II machine as a programming environment [8, 63] by some of his colleagues. This material appears in Mukai [55], Mukai and Yasukawa [64], and Sugimura, Miyoshi and Mukai [89].

● We introduce a class of algebras called record algebras as a mathematical model for the PSTs. A record algebra is a partial algebra with an operator domain. The algebra, R, is a commutative and idempotent partial monoid with respect to an operation called merge. The operator domain, G, is also a monoid called an access monoid which acts on R as a monoid from both sides, satisfying certain appropriate laws on the interaction between these actions and the merge. Elements of R and G are called records and accesses, respectively. Then we embed Herbrand universe H into a record algebra (R, G) so that the CLP over (R, G) is a generalization of the definite clause grammar on H. Moreover, we show that DAGs used in unification grammar formalisms are pictures of records and that records are denotations of DAGs. Thus we integrate unification grammars and the standard logic programming into the constraint logic programming over the record algebra. This material appears in Mukai [57] and Mukai [61].

● We propose a theory of feature structures based on P. Aczel's non-well-founded sets [1] (also called hypersets). A constraint language L_A over the class V_A of hypersets over A is proposed, where A is a class of atoms. L_A is a quantifier-free first-order language with equality, subsumption, disjunction, and negation. For a sublanguage L_{A_ω} of L_A and subclass V_{A_ω} of V_A, it is proved that V_{A_ω} is solution compact with respect to L_{A_ω} and L_{A_ω} is satisfaction complete with respect to Aczel's hyperset theory ZFCA[2] in the sense of the CLP schema. In fact, V_{A_ω} is the class of hereditarily finite hypersets in V_A, and L_{A_ω} is the class of constraints in L_A consisting of finitary terms. Two applications are given by restricting V_{A_ω} and L_{A_ω} in a natural way. One is an infinite tree unification in logic programming, and the other is a unification over feature structures for unification grammar formalisms. We remark for implementation that a

[1]CIL stands for Complex Indeterminate Language.

[2]ZFC minus the axiom foundation plus Aczel's anti foundation axiom.

partition refinement algorithm on transition nets finds the coarsest bisimulation relation which extends a given relation on hypersets. We show a basic fact that V_{A_ω} is compact with respect to L_{A_ω} if and only if A is empty, i.e., atomless. This material appears in Mukai [56, 59]. Then, introducing new class of functors called form-based and coalgebras, we prove a generalized unification theorem on coalgebras for class-valued functors which are form-based, conservative, and set-based. Finally, we prove that every countable set of algebraic equations with countably many variables is solvable in the complex number field if and only if so is every its finite subset. The theorem is a foundation of the Gröbner-base method working on perpetual process in the CLP.

Acknowledgments

The greatest thanks go to Professor Hozumi Tanaka, Tokyo Institute Technology, TIT, who supervised me and kept giving me advice during my thesis study. I am also grateful to Professor Masamichi Shimura, Professor Takuya Katayama, Professor Naoki Yonezaki, and Professor Motoji Saeki, TIT, for reviewing my thesis and giving me useful comments and criticism.

Soon after Institute of New Generation Computer Technology, ICOT, had launched, Kazuhiro Fuchi, Director of ICOT, gave a lecture there. He introduced Colmerauer's infinite trees, among other topics of logic programming. I was impressed by the lecture, in particular, the infinite tree. This thesis treats non-well-founded structures, and my interest in them was initiated from the lecture. I have been encouraged by Koichi Furukawa, Deputy of ICOT. I have greatly benefited from his perspective comments. His interest in my work on CIL around 1983 was a force for developing CIL further, and, accordingly, for this thesis. This thesis is related to situation theory and situation semantics, STASS. It was Toshio Yokoi, Director of EDR, who first gave my colleagues and me a suggestion to investigate STASS as a background of natural language semantics while he was the chief of our laboratory for natural language processing. His frontier-spirit encouraged my research.

I would like to thank Toshiyuki Okunishi, Toshiyuki Amanuma, and Takayuki Suzuki for their efforts to extend my prototype CIL system written in Edinburgh Prolog on DEC-2060 so that it works on the PSI machine. I discussed many times with Shozo Kondoh and Makoto Imamura on applications of CIL to situation-oriented discourse processing. Also I would like to thank Ryoichi Sugimura for his efforts to put the built-in record structure, PST, in wide use for natural language processing.

On the hyperset constraint in this work, I was received many useful comments and suggestions in various ways from the following people: Hideki Yasukawa, Kaoru Yoshida, Kazumasa Yokota, Makoto Imamura, Satoshi Tojo, Hideyuki Nakashima, Koiti Hasida, Satoshi Menju, William Rounds, and Jon Barwise. Peter van Emde Boas, Professor of Amsterdam University, gave me useful information about the compactness of algebraic equational constraints over the complex number field and Yōsuke Sato improved my proof of the theorem. Also Mark Stickel, SRI International, was so kind to spend time for proof reading this part during his stay at ICOT to give me comments, in particular, on English writing. I am thankful to all of them.

This thesis is based on published papers of mine. I would like to thank anonymous referees of those papers for useful comments and suggestions.

I was stimulated and informed through many conversations at the regular meeting on STASS held at ICOT: Hozumi Tanaka, Hidetoshi Shirai, Syun Tutiya, Yasuhiro Katagiri, Masahiko Sato, Yukiyoshi Kameyama, Masahito Kawamori, Satoshi Kobayashi, Jinsei Yamaguchi, Takafumi Sakurai, Hiroyuki Suzuki, Satoshi Tojo, Hideyuki Nakashima, Ikumi Imani, and Manabu Okumura. On language and meaning in general, I also have greatly benefited from conversations with Jon Barwise, Gerald Gazdar, Johan van Benthem, Per Martin-Löf, Franz Guenthner, Robin Cooper, Fernando Pereira, Susumu Hayashi, Kō Sakai, Yukihide Takayama, Ken Satoh, Jean-Louis Lassez, Joseph Goguen, Carl Pollard, and Tim Fernando.

It is my pleasure to express my thanks to Hidenori Itoh, Jiro Tanaka, and Phillipe Devienne for their hints and suggestions on research in general from fresh view points as well as for many pleasant lunch times with them.

I would like to express special thanks to ICOT for providing an invaluable research environment for this work. Using many opportunities supported by ICOT, I was able to have many fruitful communications with researchers from other research institutes. In fact, this thesis came out of such communications. I would like to thank general research managers of ICOT, Kyoji Kurozumi, Yoshihisa Ogawa, and Shunichi Uchida, for their efforts to keep high this research environment. Also I would like to thank Shunichi Uchida and Tsutomu Yoshioka for encouraging me to accomplish this thesis study.

I am grateful to my parent company, Mitsubishi Electric Corporation, in many ways. I enjoyed research and development there for about ten years. I met and worked with many excellent and good-natured people there. I am indebted to the following people for learning things in actual computer software developments: Masaru Sudo, Shoji Sekimoto, Yasuo Tachibana, Takeshi Sunohara, Kunio Iharada, Tetsuo Mizoguchi, Takasi Dasai, Junichi Sadamatsu, Akira Takano, Yuji Wada and Katsushi Suzuki.

I am thankful to Sonoko Tani for her helps during the preparation of this thesis. Takenobu Tokunaga, TIT, helped me on the thesis submission to TIT. I would like to thank him for his kindness.

Finally, I would like to thank my family, Sachie Mukai, Yumi Mukai and Mari Mukai for their supports as the background situation of my research work.

April 1991

Tokyo

Contents

Chapter 1

Introduction

The general aim of this work is to study information structures, constraints, and computations. The work is divided into the following three parts.

- CIL: a logic programming system which has built-in feature structures.
- Record algebra: a partial algebra model of feature structures.
- CLP(AFA): a constraint logic programming language over hypersets.

CIL is described in Chapter 5, the record algebra in Chapter 4, and CLP(AFA) in Chapter 3, in the reverse order of historical developments. These chapters are independent to each other, and each of them has its own introduction and preliminaries. So, in this introduction, we first describe quickly the contents of the following chapters, then, we explain a new perspective of information structures developed during this work.

A feature structure is a form of partial description of objects. It came out of the computational linguistics. It is an abstraction of record-like structures: the frame proposed by Minsky, semantic net in knowledge representation, record in programming languages, attribute-value pair list, association list, finite state automata, graph, first-order term, and so forth. The feature structure is used by the agent in various ways; information is conveyed, stored and processed in the form of feature structures. Also feature structures are used to classify objects and situations by the agent to pick up necessary information out of them for his real life. Thus, it is important to study the logic of feature structures for modelling information processing by the agent. In fact, in recent years, many logics of the feature structures have been proposed, though the mathematical model of the feature structures has not been fixed yet (Moss [54]).

In this thesis, we propose two constraint theories over feature structures. One is an algebraic model called record algebras, and the other is a set-theoretical one in the universe of hypersets. We are interested in computational theories of feature structures that can be built-in actual programming languages; we relate the structure to the constraint logic programming scheme, CLP(X) (Jaffar and Lassez [39]). The CLP is a class of logic programming languages based on constraint solving, and is a generalization of the standard logic programming, where X is the parameter of the scheme. By instantiating X with a constraint theory over a problem domain, D, a constraint logic programming language over D is obtained. The built-in constraint theory is called also a unification theory. In this thesis, we instantiate the parameter X of the CLP scheme with the two constraint theories over feature structures, i.e., a record algebra and a domain of hypersets, and thereby, we obtain a family of CLPs over feature structures. Our unification theory over hypersets extends Colmerauer's unification theory over infinite trees with equations ($=$) and unequations ($\neq$) (Colmerauer [21, 22, 23]).

The organization of this thesis is as follows. In Chapter 2, we prepare basic notions from Aczel's hyperset theory, situation semantics, and unification grammar. The CLP scheme is reviewed in the introduction of Chapter 3.

In Chapter 3, we describe the CLP over hypersets. First, a unification theory over hypersets are described. The solvability of hyperset constraints is characterized by a consistent and closed set of basic constraints. Then, semantics of the CLP over hypersets is described. It is proved that the fair SLD computation rule is sound and complete, and then, that the negation-as-failure rule is sound and complete. The semantics is described in the hyperset theory.

In Chapter 4, a CLP family over record algebras is described. The feature structures are formalized to be a record algebra, and are given a compact constraint theory. As an application of the record algebra, it is shown that unification grammars are Horn clause programs over a record algebra.

In Chapter 5, a CLP system, CIL, is described. Syntax of terms and programs, operational semantics, various built-in constraints and functions are described. CIL applies to linguistic analysis, which includes PST, discourse situations, mental state representation taken from situation semantics. Section 6 concludes this thesis.

Now, we turn to the above mentioned perspective on information structures; the perspective supports the hyperset-theoretical approaches proposed in this thesis. In general, a framework of information structures requires two kinds of things: generating devices of universes of objects and computations for objects. Here, we view the unification or constraint theory on the objects as an computational engine.

For example, take a logic programming [46], Prolog, as a framework of information structures. The universe for a Prolog program is the set of first-order terms inductively generated from function and constant symbols of the program, and computations are proof trees or derivation trees. The unifier of Prolog is the engine of the computation, namely, the resolution. There are two alternatives for a choice of universes: a well-founded universe and non-well-founded one. The former consists of the standard first-order terms, and the latter has Colmerauer's infinite trees. Let P be a Horn clause program, i.e., a set of Horn clauses. Then, let T_P be the transformation on the powerset of B_P induced by P, where B_P is the Herbrand base of P. The fixpoint semantics of the program P is defined as a special fixpoint of T_P. There are two alternatives for a choice: the maximum one and the minimum one. Similarly, operational semantics of a Horn clause program can be viewed as a form of inductive or coinductive definitions of proof trees or derivation trees. According to the two types of trees, i.e., well-founded one and non-well-founded one, again there are two alternatives for a choice: inductive definition and coinductive definition. The former has only finite-length derivations, and the latter has infinite-length ones as perpetual processes.

Thus, we observe that duality of well foundedness and non-well-foundedness is in the information structures: first order terms, infinite trees, records, proof trees, computation as state transitions, graphs, feature structures, sets, and so on. We add even the semantics to this list, because the semantics of a program is a structured object as a set consisting of ground atomic formulae or a derivation tree. By duality, we mean category theoretic duality [45]. Thus, we have the following table of dualities in information structures:

Dualities on Information Structures

component relation	well-founded	non-well-founded
definition method	induction	coinduction
operation	algebra	coalgebra
fixpoint	minimum	maximum
limit object	initial	final
set theory	ZFC	ZFCA

Each pair on the table forms a dual notion. Recall the unification on first-order terms and infinite terms differ only on the 'occur-check'; they form a dual pair. Assuming an appropriate metatheory, if one gets an object with some property, then he can get automatically another object with the dual property. For example, suppose that we have the minimum class such that $M \supseteq pow(M)$, then also we can have the maximum class such that $N \supseteq pow(N)$, where pow is the power class functor. In fact, M is the universe of well-founded sets, and N of non-well-founded ones.

The domain of the standard meta-theory, ZFC, of information structures is itself a well-founded universe. As the table suggests, the membership relation ($\in$) can not be used for the component relation on non-well-founded structured objects. In fact, non-well-founded objects can not have natural coding in ZFC; ZFC is not large enough for the duality. It is this point that the duality between first-order terms and infinite trees, for instance, is unclear in the existing models that uses ZFC as a meta-theory. For example, an infinite tree is defined as a function on a set of finite strings closed under prefix, and, on the contrary, the first-order terms are defined inductively [46]. The duality between the two universes is not clear from such definitions.

On the other hand, the universe of ZFCA is enough large to keep the duality; a final coalgebra theorem holds for the category of classes of hypersets as well as the initial algebra theorem for any set-based functor [1, 5]. Using these initial algebra and final coalgebra theorems, universes of generalized terms are defined as fixpoints of set-based class-functors.

Using the duality of ZFCA universe, we will find in Chapter 3, in particular, Section 3.6, a family of CLPs over structured objects. The outline of the finding story is as follows. First, we can see clearly that the universe of ZFCA is constraint-definable and compact with respect to the bisimulation theory [1] on the universe, where these two properties are what the CLP scheme requires. So, the problem is to generalize the bisimulation to that on generalized terms and to find a family of set-based functors, T, such that the final coalgebra, $J(T)$, for T is constraint-definable and compact with respect to the relativized bisimulations on the generalized terms. The family of CLP($J(T)$) for such T will have the same properties as those of the CLP over Colmerauer's infinite tree unification with $\neq$.

To find the family of T, the membership relation is generalized so that the 'argument-of' relation on standard terms, for example, is such a generalized one. Then, two properties on functors are defined: form-based and conservative. Finally, it is proved that the final coalgebra for a set-based, form-based and conservative class-functor satisfies the CLP requirements.

Thus, due to the non-well-founded axiom, ZFCA provides an powerful metatheory for various information structures, in particular, structured objects.

Chapter 2

Preliminaries

Each of the following chapters gives necessary preliminaries in the section after the introduction. Chapter 3 and Chapter 4 are mathematical and self-contained; every technical notion is defined, and every assertion is attached with a proof except the solution lemma [1] and the final coalgebra theorem [5]. So, we review only some background stuff for this thesis from the following: situation semantics and situation theory (STASS, for short), unification grammar, and hyperset theory. The **CLP scheme** is reviewed in Chapter 3.

We begin with **STASS**. The situation semantics [18] is a relational theory of meaning [90, 67, 37]; it can be seen a successor of possible world semantics. The standard version of situation semantics asserts the following [18]:

- The world consists of situations: a situation is a small portion of the world, which can stand in relations to others, i.e., situations are objects.
- The meaning of a sentence is a relation between situations: the truth of sentences are one of the attributes of sentences.
- Situation semantics treats attitudes, e.g., beliefs. The idea is to represent the mental state by a pair of a frame and setting.

Indexicals such as 'I', 'You', 'now', 'here' and demonstratives such 'that', 'this' are treated in situation semantics. In Chapter 5, an example CIL program treats these context dependent elements of language uses.

Situation theory [14, 26, 10, 11, 15, 31] is a meta theory of meaning, logic, and information. Cooper et al [24] is a collection of the state-of-the-art papers on the situation theory and its applications. The following are efforts towards mathematical foundations of situation theory: theory of relations (Plotkin [74, 75, 76]), theory of structured objects (Aczel [2, 3]), model of situation theory (Barwise [11]) based on Aczel's set theory, proposal of branch points of situation theories (Barwise [14, 9]). Situation theory has many kinds of objects as first-class citizens: sets, types, relations, propositions, state of affairs, and so on. Due to this rich universe of objects, STASS is related to other frameworks and meta-theories: property theory [20], Landman's model of information [44], Martin-Löf's type theory [50], Feferman's system [30], Montagovian approaches [52, 53, 67, 68, 91], etc.. However, the comparison of these frameworks is out of the scope of this thesis.

Partial assignment is an incomplete list of arguments, i.e., a partial function which assigns values to argument places; it plays an important role in semantics [66]. According to situation theory, each relation R is given a set $arg(R)$ of argument places. A partial assignment for R is a function f which is defined on a subset of $arg(R)$. Situation semantics asserts as a slogan that not only total but also partial assignments are first-class citizens in semantics. Based on this ontology, the important notion of dynamic interpretation of natural languages is captured as a process which generates and modifies partial assignments. It might be useful to think that the partial assignment is the memory state of a computer, and that the rule of the interpretation is a computer program. This is a relational view of meaning and an extension of the classical model theory. In classical model theory, satisfiability is a ternary relation between models, M, assignments f and formula ψ:

$$M, f \models \psi.$$

If the model and the formula are fixed, then we get a unary relation relation on assignments. Relational theory meaning extends the classical one, so that meaning of a sentence is a binary relation on assignments. This is also called dynamic interpretation (Kamp [42] and Barwise [13]). As these dynamic interpretation models suggest, the relational theory and operational semantics of computer languages are close to each other. Also frameworks of computer languages and natural languages are getting closer (Benthem [92, 93], Mukai and Yasukawa [65]).

Unification grammar is a constraint theory on phrase structures and feature structures [32, 78, 82, 78]. Phrase structures are ordered trees. Each node of a phrase structure is assigned a feature structure. Phrase structure rules describe constraints on phrase structures and feature structure assignment. For example, the following phrase structure rule

$$M \;\rightarrow\; H\ C \mid M!head = H!head$$

means a local structural constraint. First, let m be the mother node and let h and c be the two daughter nodes. Suppose that M, H, and C be feature structures assigned to the nodes m, h, and c, respectively. Then the above rule says that the 'head' component of M is the same as the 'head' component of H.

The semantics of a unification grammar is the set of phrase structures with a feature structure assignment which satisfy all the local constraints expressed by the grammar rules. Namely, it is a device which generates the set inductively or coinductively. The generating principle of the unification grammar is the same as that of a constraint logic program, which is a device to define the class of true propositions in the same way. Based on this observation, in Chapter 4, we will identify a unification grammar to be the corresponding Horn clause program over the record algebra.

We informally explain basic ideas of the **hyperset theory**. In the classical standard set theory, ZFC, it follows from the axiom of foundation that it is impossible to have an infinite sequence such as the following:

$$\cdots \in x_n \in x_{n-1} \in \cdots \in x_1 \in x_0.$$

Namely, the universe of ZFC set theory consists of only well-founded sets. P. Aczel began a non-well-founded set (= hyperset) theory with a motivation to replace the graph theoretical foundation of Milner's SCCS theory [51] by the non-well-founded sets. Aczel's set theory is the ZFC minus the foundation axiom plus the following anti-foundation axiom (AFA, for short): every graph has a unique decoration. In general, AFA is equivalent to the following solution lemma: every system of equations $(x = b_x)_{x \in X}$ has a unique solution, where X is a set of parameters and $(b_x)_{x \in X}$ is an indexed family of X-sets (parametric sets). Non-well-founded sets are also called *hypersets*. For example, the following family of equations have a unique solution:

$$x = \{a, x\},\ y = \{b, y\}.$$

The solution lemma of Aczel's will be used heavily in Chapter 3. In the universe of hypersets, the inductive definition is not available in general, since the induction is based on the well-foundedness of the membership relation.

Given a class-functor Γ, the coinductive definition means the largest fixpoint, $J(\Gamma)$, of Γ. As $J(\Gamma)$ exists for any set-based class-functor Γ, and $J(\Gamma)$ is the largest class M such that

$$M \subseteq \Gamma(M).$$

The coinductive definition is useful to introduce infinite structures. For example, the domain of infinite binary trees is defined to be the largest set B such that the following hold:

If $x \in B$ then either $x = \phi$ or $x = (y.z)$ for some $y, z \in B$.

In Chapter 3, coinductive meanings to infinite computations of Horn clause program will be given.

Aczel [1], and Aczel and Mendler [5] proved a final coalgebra theorem: every set-based functor has the final coalgebra. This theorem will be used in Section 3.6. A bisimulation relation is a equivalence relations on a graph which is compatible with the successor relation. It follows from AFA that a bisimulation relation on the class of hypersets is the identity relation.

Barwise [14] sees that the hereditary subset relation on sets can be a model of the (extensional) subsumption relation between feature structures. He proved a unification theorem that a constraint is solvable iff it extends to some consistent constraint closed under the bisimulation axioms and the subsumption axioms. This theorem is important in this thesis. In Chapter 3, we extend Barwise's unification theorem in two ways.

Using AFA, Aczel proposes also a theory of structured objects [2, 3] to provide mathematical tools for situation theory. For example, it is a generalization of the powerset operation in set theory and term generating operation in a signature of algebra. According to Aczel [3], situation theory is both set theoretical and syntactical in the sense that situations are set theoretical objects, and propositions and states of affairs are syntactical objects. As we have mentioned in the introduction, also we will generalize the membership and 'argument-of' relations to find a family of CLPs over structured objects including infinite trees.

Chapter 3

Constraint Logic Programming over Hypersets

A constraint language L_A over the class V_A of hypersets over A is proposed, where A is a class of atoms. L_A is a quantifier-free sublanguage of the first-order language with equality, subsumption, disjunction, and negation. For a natural sublanguage L_{A_ω} of L_A and subclass V_{A_ω} of V_A it is proved that V_{A_ω} is solution compact with respect to L_{A_ω}. A subtheory of L_{A_ω} is given that is satisfaction complete for positive constraints in the sense of the constraint logic programming (CLP) schema. In fact, V_{A_ω} is the class of hereditarily finite hypersets in the universe V_A of Aczel's hyperset theory ZFCA[1] and L_{A_ω} is the class of constraints in L_A that consists of finitary set terms. A characterization of the canonical programs in L_A is given in terms of extended bisimulation relations. Thus CLP(AFA) over the hypersets is obtained as a new instance of the CLP schema.

Furthermore, the CLP scheme is reconstructed and given a foundation within the hyperset theory. First a declarative semantics and an operational semantics of Horn clause programs over L_{A_ω} and V_{A_ω} are given based on coinductive definition. Then soundness and completeness of the two semantics are proved by showing what we call a *simulation* relation between the two semantic domains.

Two applications are given by restricting V_{A_ω} and L_{A_ω} in a natural way. One application is an infinite tree unification in logic programming. The other is a feature structure unification in unification grammar formalisms. It is pointed out that not only can the UNION-FIND algorithm be applied to the restricted cases as usual, but a partition refinement algorithm on transition nets can also be applied straightforwardly to find the coarsest bisimulation relation that extends a given relation on hypersets.

3.1 Introduction

Aczel [1] proposed the universe of hypersets (= non-well-founded sets) for modelling non-well-founded structured objects such as streams in Milner's SCCS [51]. He provided a powerful coinductive method like the inductive one in the standard universe of well-founded sets. Barwise and Etchemendy [16] and Barwise [14] apply Aczel's theory to circular situations and unification

[1]ZFC minus the axiom foundation plus Aczel's antifoundation axiom.

of feature structures respectively. In the universe of hypersets, any collection called a system of equations has a unique solution. This lemma is called the solution lemma. For example, $x = \{x\}$ has a unique solution, which is obtained by unfolding the right side of the equation unboundedly to get, intuitively $x = \{\{\{\cdots\}\}\}$.

Jaffar and Lassez [39] proposes a scheme CLP(X) as a generalization of standard logic programming, where X is a parameter for constraint languages. A constraint language consists of a syntax of constraints, a notion of solutions, and a set of constraint solving rules. Pure Prolog is an instance of the scheme with X being the constraint language of the standard theory of equality over the Herbrand domain. The CLP scheme requires X to be *solution compact* and *satisfaction complete* to assure that SLD fair computation is sound and complete. Moreover, the scheme assures soundness and completeness of the negation-as-failure rule (NAFR) [46] for *canonical* programs. The notions 'solution compact', 'satisfaction complete' and 'canonical', among others, are explained informally below and will be defined formally in a later section. By soundness we mean that (2) $\vee$ (3) $\Longrightarrow$ (1) for every goal g and by completeness we mean the converse, where

(1) g has a solution.

(2) There exists a finite success derivation for g.

(3) There exists an infinite derivation for g.

Also by soundness of NAFR we mean that (4) $\Longrightarrow$ (5) for every goal g and by completeness of NAFR we mean the converse, where

(4) g has no solution.

(5) Every derivation for g fails in a finite number of steps.

A program is called *canonical* iff for every success derivation the constraints appearing in the derivation have a solution. For example it will be shown that a program in CLP(L_A) explained below is canonical if every parameter appearing in the negative part or on the right side of subsumption ($\sqsubseteq$) constraints in the derivation will be bound to nonparameter terms in a finite number of steps in the derivation.

Remark In Stuckey [88], a canonical program, P, is defined to be a program such that the maximum semantics is reached within countable steps of iterations of the transformation T_P for P on the power set of the Herbrand base of P. (Also see Lloyd [46].) If we have not negative constraints, then it will be evident later that our program is always canonical in this sense. □

A constraint language X is *constraint definable* if for every element x in the domain of X there is a constraint c in X such that x is a unique solution of c. In other words, constraint definability means that every element is a limit of a sequence of finitarily representable elements. For example, as every real number is a limit of some sequence of rational numbers, the domain of real numbers is constraint definable w.r.t. (with respect to) $<$ and rational number constants. A constraint language X is called *compact* provided that for every set C of constraints C has a solution iff every finite subset of C does.

The solution compactness condition is divided into two parts. One half is the same as constraint definability above. The other half is that for any given finite constraint c each element which does not satisfy c can be covered with a finite constraint c' that shares no solution with c. Clearly it follows from these definitions that if X is constraint definable and compact then X is solution compact. However, the converse is not always true.

Roughly speaking, the satisfaction completeness condition of the CLP scheme states unifiability and solvability are equivalent. More precisely, it requires that the existence of solutions of given constraints is characterized by the syntactic rules of the constraint language. For example, standard unification language over the Herbrand domain, i.e. first-order term unification, is satisfaction complete because unifiability and solvability of constraints are equivalent. Recall that a given family of equations is unifiable iff there exists a congruence relation that extends the given family.

Based on these notions, the CLP scheme provides a foundation for logic programming to treat infinite objects such as irrational numbers through constraints [40] and perpetual processes [46]. We are interested in nonterminating programs that process non-well-founded structures such as streams. These applications need maximum semantics because the standard minimum semantics does not give meanings to programs that treat streams, for example.

So far, the complete Herbrand domain, i.e. the domain of infinite trees, and the domain of real numbers with a certain appropriate class of constraints have been counted as two major instances of the CLP scheme. In recent years logics of feature structures have been studied extensively. The reader is referred to Moss [54] as one of the most up-to-date works on a logical foundation of feature structures. Several authors, Rounds [79] for example, suggest that the domain of feature structures is a new important instance of the scheme. Barwise [14] shows that feature structures can be modeled as hypersets in V_A (the full universe of hypersets over A) with the hereditary subset relation as subsumption relation. Accordingly, the domain of hypersets should be examined as a bridge that connects unification grammar formalisms and constraint logic programming.

Guided by the CLP schema, first we propose a constraint language L_A over the class V_A of hypersets over A, where A is a class of atoms. L_A is a quantifier-free sublanguage of the first-order language with equality, subsumption, disjunction, and negation intended for application to feature structure unification. We then propose a natural sublanguage $L_{A\omega}$ of L_A so that $L_{A\omega}$ is the class of constraints in L_A consisting of finitary terms. We will show that $V_{A\omega}$ is solution compact w.r.t. $L_{A\omega}$. As negative constraints are allowed, $L_{A\omega}$ is not satisfaction complete w.r.t. $V_{A\omega}$. However, we give a complete characterization of solvable positive constraints. Moreover, a subclass of $L_{A\omega}$ which consists of constraints called canonical is proved to be satisfaction complete. The Essential points here are that the solution lemma [1] is just constraint definability and that the class of positive constraints in $L_{A\omega}$ is compact. We give more details for these points.

In fact, there are clear reasons why $V_{A\omega}$ and $L_{A\omega}$ satisfy the CLP criteria. Constraint definability is clearly satisfied because of Aczel's solution lemma, i.e. every element of $V_{A\omega}$ can be represented as a unique solution of a system of equations in $L_{A\omega}$. As will be seen in the proof of solution compactness of $L_{A\omega}$, the second half of the solution compactness is obtained from constraint definability and compactness of the positive constraints. Barwise [14] gives a characterization of solvability of *positive* constraints in terms of a certain kind of extensions of the given constraints. The extension is called a *simulation pair*. A simulation pair is a set c of equations and subsumptions such that c is closed under the rules for bisimulation and subsumption relations and that *every parameter in the constraint is bound to a set term*. We refer to this theorem as Barwise's unification theorem. We need to extend the theorem to the case in which constraints have free parameters. In fact, we will obtain Proposition 3.8 that asserts that every simulation pair with 'unbound parameters' can be extended to a simulation pair without unbound parameters. This is a key lemma in this chapter. We refer to this lemma as the *binding lemma.*

A complete characterization of solvable constraints is an open problem at this time: we give only a partial characterization. Barwise's unification theorem is a complete characterization for the class of the solvable positive constraints over hypersets. We introduce a class of canonical constraints explained above. The notion of canonical supports is an extension of that of simulation pairs to the case in which negative constraints are allowed. Soundness and completeness of both the SLD fair computation rule and the NAFR (negation-as-failure-rule) are proved for the class of programs such that the accumulated constraints on the whole of unfailing derivation always form a canonical constraint.

In an earlier stage of logic programming, Colmerauer [23] introduced the domain of infinite trees for modelling circularity. We try to extend infinite trees to hypersets to see how well the CLP schema and unification grammar formalisms can be reconstructed. One technical advantage of hypersets over infinite trees is that the theory can be constructed in an abstract way which is highly independent of the concrete structure of objects. Remember that the mathematical formulation of the complete Herbrand domain needs auxiliary notions such as path, metrics and limit [46], while in the universe of hypersets we can start immediately from the solution lemma.

We use Aczel's theory in two ways. One is to define the semantics of a program coinductively and its use as a domain of logic programming as described above. In the former, two classes are defined coinductively. One is a class of non-well-founded triples for the declarative semantics of the program and the other is the class of non-well-founded pairs for SLD-like fair computation trees for queries (= goals). Soundness and completeness results are proved by showing a relation called *simulation* between them. Indeed, the simulation relates solutions to computations in a hereditary and coinductive way.

Here are a few remarks on algorithm complexity which concern hypersets. First, one of the partition refinement algorithms proposed in Paige and Tarjan [69] can be applied to check the solvability of a given system of equations. Let n and m be the numbers of nodes and edges, respectively, appearing in the system, assuming that the system is represented suitably as a directed graph. Then there exists an $O(m \log n)$ time-complexity algorithm to compute the coarsest bisimulation for a given partition of the system. Second, the subdomain of records in V_{A_ω} has an efficient implementation for a theory of $=$ and $\neq$. In fact, these constraints can be solved with UNION-FIND technique [6].

This chapter is the foundation of the the logic programming system, CIL, described in Chapter 5. Also, the idea of CLP over hypersets and the results on unification over hypersets presented in this chapter have been used as the kernel of the knowledge representation language called Quixote [95].[2]

In addition to these results, we will prove that every countable system of algebraic equations is solvable in the complex number field if and only if so is every its finite subsystem. This fact gives a foundation of the meaning of the incremental algebraic constraint solver in perpetual

[2]Quixote is under development at icot, Tokyo.

process, in which an infinite number of variables and equations may occur in the limit of the computation. Also we will show that $V_{A\omega}$ is compact with respect to the class of $L_{A\omega}$ if and only if a is empty. This result suggests some difficulty in computational treatment of the subsumption constraints. As a demonstration of our theory of the hyperset constraint, we will show decidability of Colmerauer's infinite tree unification theory with unequations in a generalized form. See Section 3.6.

Although only a non-deterministic algorithm is shown, the well known union-find algorithm can be available if it is restricted to the case of terms or records. Namely, the set unification presented not only gives a theoretical perspectives but also has a good subproblem for which an efficient algorithm exists. The constraint language can be extended to have set constructors, $\cup$, $\cap$, $\setminus$ (set difference), *pow* (power set) so that it has a very similar characterization of sufficient condition of the solvability. It is interesting to give a more powerful set constructors. we will give a partial solution in Section 3.6 based on categorical notion giving an example of merge operation on non-well-founded record structures in a similar way of Aczel's treatment of Milner's process algebras.

We give semantics of the CLP over set constraints, using Aczel's set theory. The soundness and completeness result is formalized in terms of certain kinds of correspondence between computation trees defined operationally and solution trees defined declaratively. the computation is sound if every path of a computation tree compute a solution. For the converse, the computation is complete if for every solution there exists a computation path which compute the solution. these notions are formalized and proved in Aczel's set theory. The point is that Horn cla[illegible] program is a form of inductive and coinductive definitions over an appropriate given n[illegible] founded universe.

3.2 Preliminaries

By $x \subseteq y$ we means that x is a subset of y, including the case of $x = y$. also, by $x \sqsubseteq y$ we mean that x is a hereditary subset of y, including the case of $x = y$. We use Aczel's hyperset theory ZFCA [1] as a metatheory throughout the chapter. The following concepts are from Aczel [1]: coinductive definition, solution lemma, bisimulation relation [70]. The definition of solution compactness and satisfaction completeness are from Stuckey [88].

Let A be a possibly empty set of *atoms* and let X be an infinite set of *parameters* (=variables). V_A denotes the class of *hypersets* over A. $V_A[X]$ denotes the class of hypersets over $A \cup X$. Sets in $V_A[X]$ are called X-sets (over A). V_A is a subclass of $V_A[X]$. $V_{A\omega}$ denotes the class of hypersets in V_A that are hereditarily finite, i.e. finite at all levels. $V_{A\omega}[X]$ denotes the class of X-sets over A that are hereditarily finite. $V_{A\omega}$ is a subclass of $V_{A\omega}[X]$. An element of $V_A[X]$ is also called a *term*.

The *transitive closure* of a set $x \in V_A[X]$ is the least transitive set T in $V_A[X]$ such that $x \subseteq T$. *trans*(x) denotes the transitive closure of x. For example

$$trans(\{a, \{b, c\}\}) = \{a, \{b, c\}, b, c\}$$

where a, b, c are atoms. In general u *appears* in v when $u \in trans(\{v\})$.

Definition 3.1 A set is *rational* if it has the finite transitive closure. □

Clearly a rational set is hereditarily finite but the converse is not always true. To see this, let $\{A_i\}_{i\in N}$ be a family of finite sets of atoms such that A_i is a proper subset of A_j for $i < j$, where N is the set of nonpositive integers. It is clear that the solution for x_0 of the system $\{x_i = A_i \cup \{x_{i+1}\} \mid i \in N\}$ of equations is hereditarily finite but it has no finite transitive closure.

Definition 3.2 A *system of equations* is a collection of *equations* $x = b_x$ for $x \in X$, where X is a collection of parameters and $\{b_x\}_{x\in X}$ is a family of X-sets. □

Theorem 3.1 (Solution Lemma (Aczel [1])) *Every system of equations has a unique solution in V_A.*

Definition 3.3 For any A and X, a set u in $V_A[X]$ is *finitary* if u is hereditarily finite and well-founded. □

Let N be the set of natural numbers. Then the set $\{N\}$ is finite but neither finitary nor rational. The set Ω such that $\Omega = \{\Omega\}$ is finite. Moreover, it is hereditarily finite and rational, but not finitary.

Definition 3.4 We assume a collection of *nodes*. A *system* M is a collection of ordered pairs of nodes such that for any node a in the system, the successors of a form a set (not a proper class) which is denoted by a_M. □

Definition 3.5 A binary relation R on the system M is a *bisimulation* on M if $R \subseteq R^+$, where for $a, b \in M$

$$aR^+b \iff \forall x \in a_M \exists y \in b_M xRy \quad \& \quad \forall y \in b_M \exists x \in a_M xRy.$$

□

For the sake of convenience we represent a bisimulation relation R as the collection c_R of equations $a = b$ such that aRb. A collection c of equations is called a *bisimulation constraint* if $c = c_R$ for some bisimulation relation R. c can be a proper class. We make no distinction between the bisimulation relation and the bisimulation constraint.

3.3 Unification over Hypersets

3.3.1 Constraint Language L_A

In this subsection we introduce a constraint language L_A using the definition of Smolka [85] and Hoehfeld and Smolka [36].

Constraint Language

Definition 3.6 A *constraint language* is a quadruple $(X, C, \mathcal{V}, \mathcal{I})$, where

(1) X is a set of *parameters*(=variables[3]).

[3]We make no distinction between parameters and variables.

(2) C is a set of *constraints.*

(3) $\mathcal{V}$ is a function which assigns to each constraint, say ψ, a set $\mathcal{V}(\psi)$ of parameters.

(4) $\mathcal{I}$ is a set of *interpretations.* An *interpretation* is given as a pair (D, S) of a *domain* D and a function S such that for any constraint $\psi \in C$ S assigns a set $S(\psi)$ of partial functions from X into D. An element of $S(\psi)$ is called a *solution* of ψ. If the restriction of f to $\mathcal{V}(\psi)$ belongs to $S(\psi)$ then $f \in S(\psi)$.

□

Definition 3.7 A constraint language is *compact* if for any countable set c of constraints in the language, c is solvable iff every finite subset of c is solvable. □

We fix the syntax of constraint languages $L = L_A$ and L_{A_ω} by giving X and C. X is fixed to be an infinite set of parameters. A set u is *flat* if every element of u is either the empty set, an atom, or a parameter. A constraint is *flat* if ever set term in the constraint is flat. Without loss of generality for our purpose and for the sake of simplicity we use a convention that every term in the formal part of the chapter is flat unless mentioned explicitly. A special constant symbol $\emptyset$ is reserved to denote the *empty* set: $\emptyset = \{\}$.

Definition 3.8 An *atomic constraint* is an *equation* $(=, u, v)$ or a *subsumption* $(\sqsubseteq, u, v)$, where $u, v \in V_A[X]$. The equation and subsumption are written as $u = v$ and $u \sqsubseteq v$, respectively, as usual. □

Both u and v above are called *arguments* of the atomic constraint.

Definition 3.9 A *literal* is an atomic constraint or its negation $(\neg, u)$ written as $\neg u$, where u is an atomic constraint. A *clause* is a set of literals. □

Definition 3.10 A *constraint* is an element of M, where M is the least subclass of $V_A[X]$ such that M has every literal and is closed under $\neg$, $\bigvee$, $\bigwedge$, i.e. if $t \in M$, $u \subseteq M$ then $(\neg, t)$, $(\bigvee, u)$, $(\bigwedge, u) \in M$. □

The above three forms are written $\neg t$, $\bigvee u$, $\bigwedge u$ as usual, respectively.

Let c be a constraint in L_A. The *field* of c is the transitive closure of the set of arguments of atomic constraints appearing in c. *fld*(c) denotes the field of c. For example,

$$fld(\{x = a, y \neq \{b, c\}\}) = \{x, a, y, \{b, c\}, b, c\}$$

where x, y, a, b, and c are supposed to be not set terms. Clearly *fld*(c) is finite whenever c is *finitary.*

For a constraint c in L_A a term t is called a *term of constraint* c if $t \in fld(c)$. We assume that constraints are elements of $V_{A \cup S}[X]$, where S is the set of logical connectives and relation symbols, i.e $S = \{\bigvee, \bigwedge, \neg, \cdots, =, \sqsubseteq, \cdots\}$. But we need not assume that A and S are disjoint. A *positive constraint* is a constraint in which no negation sign $\neg$ appears.

Definition 3.11 $L_{A\omega}$ is the sublanguage of L_A whose constraint consists of all *finitary* constraints in L_A. □

As a convention, we write $u \neq v$ and $u \not\sqsubseteq v$ for $\neg(u = v)$ and $\neg(u \sqsubseteq v)$ respectively. We also write $u \vee v$ and $u \wedge v$ for $\bigvee\{u, v\}$ and $\bigwedge\{u, v\}$ respectively. We fix an interpretation $\mathcal{I}$ of L_A in the following way. The distinguished constant symbol $\emptyset$ denotes the empty set. Each constant symbol denotes itself. Let s be a set term and f be an assignment such that each parameter appearing in s is in $dom(f)$, i.e. $\mathcal{V}(s) \subseteq dom(f)$. The interpretation of s w.r.t. f, denoted by $s[f]$, is *inductively* defined to be the set $\{u[f] \mid u \in s\}$ as usual, where $x[f] = f(x)$ for x in $dom(f)$ and $a[f] = a$ for any atom a. Note that we have assumed that every term in $L_{A\omega}$ is well-founded. The interpretation of the equality symbol '=' in L_A is the identity relation in the domain V_A.

Definition 3.12 The interpretation of the subsumption relation symbol '$\sqsubseteq$' is the largest relation R on V_A such that if xRy then one of the following holds:

(1) $x = y \in A$, i.e. x and y are the same atom in A.

(2) x and y are sets in V_A and for each $u \in x$ there is some $v \in y$ such that xRy.

□

Remark The $\sqsubseteq$ relation is a preorder on V_A but not a partial order. In fact the following are true, which show a failure of the antisymmetric law for subsumption:

$$\{\emptyset, \{\emptyset\}\} \sqsubseteq \{\{\emptyset\}\}.$$

$$\{\{\emptyset\}\} \sqsubseteq \{\emptyset, \{\emptyset\}\}.$$

□

Remark Even in the case of ordered pairs we cannot reduce subsumption to pairwise subsumption. For example, though $(a, b) \sqsubseteq (b, a)$ is true by definition of subsumption, but also by definition neither $a \sqsubseteq b$ nor $b \sqsubseteq a$ are possible, where a and b are two distinct atoms. □

There are 'minimum' and 'maximum' elements in V_A w.r.t. $\sqsubseteq$. In fact $\emptyset \sqsubseteq x$ and $x \sqsubseteq \Omega_A$ for any set x in V_A, where Ω_A is defined by the equation $\Omega_A = A \cup \{\Omega_A\}$ for a set A.

Proposition 3.2 *$x \sqsubseteq \Omega_A$ for any set x in V_A.*

Proof Let $R = \{(x, \Omega_A) \mid x \in V_A\}$. R is a binary relation on V_A. Clearly R satisfies all clauses of definition of $\sqsubseteq$. As $\sqsubseteq$ is the largest such relations we get $R \subseteq \sqsubseteq$. □

Definition 3.13 $\models$ is a binary relation between assignments f and constraints c such that $\mathcal{V}(c) \subseteq dom(f)$, which is defined inductively on the structure of c:

(1) $\models_f u = v$ if $u[f] = v[f]$.

(2) $\models_f u \sqsubseteq v$ if $u[f] \sqsubseteq v[f]$.

(3) $\models_f \neg c$ if it is not the case that $\models_f c$.

(4) $\models_f \bigvee u$ if $\models_f d$ for some $d \in u$.

(5) $\models_f \bigwedge u$ if $\models_f d$ for all $d \in u$.

$\square$

Definition 3.14 An assignment f is a *solution* of a constraint c in L_A iff $\models_f c$. We also say that c is *solvable* or *satisfiable* when c has a solution in the domain. $\square$

Now we have fixed the constraint language L_A. It is easy to see that a constraint c is solvable iff some disjunct of the disjunctive normal form of c is solvable. So it suffices to study the solvability of clauses.

Constraint Rules

We define a list of constraint rules for the constraint theory in L_A. We take constraint rules to be condition on clauses in L_A. It is straightforward to see that the logical reading of the following rules is sound w.r.t. the constraint language L_A. In the definition, c is a clause being conditioned. x, y, z are any terms and u, v are set terms. So u and v are neither an atom nor parameter.

Definition 3.15 Constraint Rules

(1) If x is a term appearing in c then $x = x \in c$.

(2) If $x = y \in c$ then $y = x \in c$.

(3) If $x = y \in c$ and $y = z \in c$ then $x = z \in c$.

(4) If $u = v \in c$ then for each $x \in u$ there is $y \in v$ such that $x = y \in c$ and for each $y \in v$ there is $x \in u$ such that $x = y \in c$.

(5) If $x \neq y \in c$ then $y \neq x \in c$.

(6) If $x = y \in c$ and $x \neq z \in c$ then $y \neq z \in c$.

(7) If $u \neq v \in c$ then either there is some $x \in u$ such that $x \neq y \in c$ for all $y \in v$ or there is some $y \in v$ such that $x \neq y \in c$ for all $x \in u$.

(8) If x is a term appearing in c then $x \sqsubseteq x \in c$.

(9) If $x \sqsubseteq y \in c$ and $y \sqsubseteq z \in c$ then $x \sqsubseteq z \in c$.

(10) If $x = y \in c$ and $x \sqsubseteq z \in c$ then $y \sqsubseteq z \in c$.

(11) If $x = y \in c$ and $z \sqsubseteq x \in c$ then $z \sqsubseteq y \in c$.

(12) If $x \sqsubseteq y \in c$ and either x or y is an atom (in A) then $x = y \in c$.

(13) If $x \sqsubseteq \emptyset \in c$ then $x = \emptyset \in c$.

(14) If $u \sqsubseteq v \in c$ then for each $x \in u$ there exists some $y \in v$ such that $x \sqsubseteq y \in c$.

(15) If $x = y \in c$ and $x \not\sqsubseteq u \in c$ then $y \not\sqsubseteq u \in c$.

(16) If $x = y \in c$ and $u \not\sqsubseteq x \in c$ then $u \not\sqsubseteq y \in c$.

(17) If $u \not\sqsubseteq v \in c$ then there is some $x \in u$ such that $x \not\sqsubseteq y \in c$ for any $y \in v$.

□

In fact, these rules can be used as an effective method to solve finitary constraints as described later. However, they are intended only for theoretical exposition, not efficient computation. Record structures, for example, should be used for practical computation. The record structure will be discussed later in the chapter as an implementation issue.

Support

Recall that by a clause, i.e., a possibly infinite clause in L_A, we mean the conjunction of literals in the set. For example the clause $\{p, q, r\}$ means $p \wedge q \wedge r$.

Definition 3.16 A parameter x *is bound* in a clause c if c has an equation $x = b$ where b is a set (not a proper class), atom, or $\emptyset$. □

Definition 3.17 A clause c is *normal* if each parameter x appearing in c is bound in c. □

Intuitively, a normal clause is a 'grounded constraint', i.e., every parameter is fully instantiated.

Definition 3.18 A clause c is *canonical* if the following hold.

(1) For each negative literal $l \in c$ there is a finite normal subset c' of c such that $l \in c'$.

(2) For each positive literal $u \sqsubseteq x \in c$ x is bound in c, where x is a parameter.

□

Note that a normal clause is canonical but a canonical clause is not always normal. For example the clause

$$\{x \neq y, z \sqsubseteq x, y - \emptyset, x = \emptyset\}$$

is canonical but not normal because z is not bound in the clause.

Definition 3.19 A *presupport* is a consistent clause closed under the constraint rules. That is, it is a clause in L_A that has no complementary pair of literals and is closed under the above constraint rules (1)- (17) of the language L_A. □

Every bisimulation constraint and subsumption constraint is a presupport. Also "simulation pair" in Barwise's unification theorem [14] is a presupport, which is, roughly speaking, means the union of a bisimulation and a subsumption constraint.

Definition 3.20 A *support* s is a presupport in L_A such that there is no pair p and d that satisfies the following:

- p is a positive subset of s.
- $\neg d$ is a negative literal in s.
- $p \cup \{\neg d\}$ is canonical.
- There is a presupport s' in L_A such that $p \cup \{d\} \subseteq s'$.

□

Every positive presupport is a support by definition.

Example 3.1 Take a set s:

$$\{x \neq y, x = \{x\}, y = \{y\}, y \neq x, \{x\} = x, \{y\} = y, x = x, y = y, \{x\} = \{x\}, \{y\} = \{y\}\}.$$

Clearly s is a presupport. However s is not a support. To see this, consider the canonical subset

$$\{x \neq y, x = \{x\}, y = \{y\}\}$$

of s. It suffices to show a presupport that extends the clause $\{x = y, x = \{x\}, y = \{y\}\}$. In fact the clause s':

$$\{x = y, x = \{x\}, y = \{y\}, y = x, \{x\} = x, \{y\} = y, x = x, y = y, \{x\} = \{x\}, \{y\} = \{y\}\}$$

is immediately determined to be the presupport. □

Presupports and supports will be used heavily to characterize the satisfiability (=solvability) of constraints. We say a clause p is a *presupport of clause* c if p is a presupport such that $c \subseteq p$. We also say a clause c *has a presupport* p when p is a presupport of c. We say that p is a *small presupport* of a clause c if $fld(p) \subseteq fld(c)$, i.e., only terms of c appear in p.

Proposition 3.3 *For every finitary constraint c the following hold.*

(1) The set of small presupports of c is finite.

(2) The set of small supports of c is finite.

(3) If there exists a presupport of c then there exists also a small presupport of c.

(4) If there exists a support of c then there exists also a small support of c.

Proof Let Q be the set of literals l such that $\mathit{fld}(l) \subseteq \mathit{fld}(c)$. As c is finitary $\mathit{fld}(c)$ is finite. So Q must be finite. As every small presupport of c is a subset of Q, we get (1). Similarly, we get (2).

Now we prove (3). Let p be a presupport of c. Clearly $c \subseteq p$. As Q is finite, $p \cap Q$ is finite. It is routine to check that the clause $p \cap Q$ satisfies the definition of presupport. As $c \subseteq Q$ we get $c \subseteq p \cap Q$. Therefore, $p \cap Q$ is a small presupport of c. Similarly, we get (4). □

Proposition 3.4 *For every finitary clause c the existence of a presupport and support of c are decidable, respectively.*

Proof In general, for a given finite set B, the existence of a presupport p such that $\mathit{fld}(p) \subseteq B$ is decidable by an exhaustive search method.

Clearly p is a small presupport of c iff p is a support such that $\mathit{fld}(p) \subseteq \mathit{fld}(c)$. Since c is finitary, $\mathit{fld}(c)$ is finite. So it follows from the above general remark that the existence of a small presupport of c is decidable. Hence, from the above proposition 3.4, the existence of a presupport of c is decidable. Similar we can prove the case of a support. □

Motivating Examples

Generally speaking, some infinite computations fail to have a limit solution even if there is no conflict found in the accumulated constraints on the way of computation w.r.t. the builtin constraint rules. So it is necessary to find a subclass of possibly infinite computations that has a 'limit'. 'Canonical' supports characterize such a subclass of computations. More precisely, every computation has a limit whenever the limit constraints on the computation has a canonical support. In fact, as will be shown later in detail, the constraint sublanguage L_{A_ω} of L_A has the nice property that every canonical support is solvable in V_{A_ω} (theorem 3.11). We use several motivating examples from some constraint languages which leads to the theorem.

First, the standard equality constraint language L_H over Herbrand universe H is not compact, where H is the set of all first-order ground terms generated by given constants and function symbols. To see this take a set c of atomic constraints

$$x_1 = f(x_2), x_2 = f(x_3), \cdots, x_n = f(x_{n+1}), \cdots$$

where f is a unary function symbol. Every finite subset of c is satisfiable but c itself is not satisfiable in H. This example shows that there is a consistent constraint in L_H which is unsolvable in H. However it should be noticed that the compactness of the language L_{A_ω} over V_{A_ω} holds only w.r.t. positive constraints. In fact because of negative constraints L_{A_ω} is not compact. Consider the constraint c consisting of literals

$$x_1 \neq x_2, x_2 \neq x_3, \cdots,\ x_1 = \{x_2\}, x_2 = \{x_3\}, \cdots. \tag{3.1}$$

Clearly every finite subset of c is satisfiable. However, by the solution lemma the global solution of the equations is uniquely determined as

$$x_1 = x_2 = x_3 = \cdots = \Omega\ (= \{\{\{\cdots\}\}\}),$$

which does not satisfies any disequality $x_i \neq x_j$ in the constraint. Note that constraint (3.1) is normal but not canonical. Moreover, it is not the case that every finite normal support is solvable. To see this let $c = \{x = \{x\}, y = \{y\}, x \neq y\}$. c has a finite normal support. However, c is unsolvable because the unique solution for the equations is $x = \Omega$ and $y = \Omega$, which does not satisfy the unequation in c.

3.3.2 The Constraint Logic Programming Scheme

The CLP(X) scheme of Jaffar and Lassez [39] is a foundation for treating infinite objects, e.g., irrational numbers and infinite computations in logic programming. There are two well known instances for the parameter X. One is the domain H^* of infinite trees. The other is the domain $\mathcal{R}$ of real numbers. The CLP scheme assures soundness and completeness of the semantics including the negation-as-failure rule. Also the CLP scheme gives a duality (called *canonical*) between maximum and minimum semantics. That is, roughly speaking, the maximum model of the program can be reached by a countable monotone decreasing sequence of approximations as well as the minimum model by an increasing one. First of all, let us recall the definition of 'solution compactness' and 'satisfaction completeness'.

Definition 3.21 (Jaffar and Lassez [39, 88]) A many sorted structure $\mathcal{R}$ is called *solution compact* if the following hold.

(1) every element in $\mathcal{R}$ is the unique solution of a finite or infinite set of constraints.

(2) for every *finite* constraint c and choice of n there exists a finite or infinite family of *finite* constraints c_i containing n parameters $x_1, \cdots, x_n$ such that:

$$\mathcal{R}^n - \{(f(x_1), \cdots, f(x_n)) \mid f \text{ is } \mathcal{R}\text{- solution of } c\}$$
$$= \bigcup_i \{(g(x_1), \cdots, g(x_n)) \mid g \text{ is } \mathcal{R}\text{- solution of } c_i\}.$$

□

Intuitively speaking, the second condition of solution compactness says that for any given finite constraint c each element in the complement of c can be separated from c by a "finite covering". We shall treat only the case of $n = 1$, since the general case in which parameters $x_1, \cdots, x_n$ in the given constraint c are chosen is reduced to this simple case by adding the equation $x = (x_1, \cdots, x_n)$ to c in which the new parameter x is the only parameter chosen, where x is a new parameter. In what follows, $\tilde{\exists} c$ means the closed formula obtained by existentially quantifying the parameters of c. Similarly $\tilde{\forall} c$ means the universal closure of c.

Definition 3.22 We shall say that a theory $\mathcal{T}$ and structure $\mathcal{R}$ correspond if

(1) $\mathcal{R} \models \mathcal{T}$ ($\mathcal{R}$ is a model of $\mathcal{T}$) and

(2) $\mathcal{R} \models \tilde{\exists} c$ implies $\mathcal{T} \models \tilde{\exists} c$ for all constraints c.

□

In order to obtain negative information from a theory $\mathcal{T}$ we require a further condition.

Definition 3.23 (Jaffar and Lassez [39, 88]) A theory $\mathcal{T}$ is *satisfaction complete* if

$$\mathcal{T} \models \tilde{\forall}\neg c \qquad \text{whenever not} \quad \mathcal{T} \models \tilde{\exists} c$$

□

In other words, a theory $\mathcal{T}$ is *satisfaction complete* if whenever there is a model M of $\mathcal{T}$ such that c is unsolvable in M, then c is unsolvable in every model of $\mathcal{T}$. Satisfaction completeness roughly means that solvability is characterized within the theory on the domain. For example, standard unification theory over the Herbrand domain is satisfaction complete because, for any set c of equations, the solvability of c is equivalent to the syntactic condition that there exists a congruence relation that extends the given system of equations. Thus, the standard unification theory $\mathcal{T}$ is the set of clauses of first-order equations which represents a congruent equivalence relation over first-order terms with no clash between signatures.

3.3.3 Solvability of Hyperset Constraints

We relate ZFCA to the constraint logic programming scheme [39]. We show that the constraint language L_{A_ω} introduced above satisfies the criteria of the schema.

Normal Constraint

The following theorem is due to Barwise [14]. X and A are classes of parameters and atoms respectively. We repeat the proof given by Barwise using normal supports instead of "simulation pair" in Barwise

Theorem 3.5 (Unification Theorem (Barwise)) *Given any set p of equations and subsumptions over $V_A[X]$ the following are equivalent.*

(1) p has a normal support.

(2) p has a solution in V_A.

Proof (1)$\Longrightarrow$(2): Let q be a normal support of p. Since q is normal for each x in $\mathcal{V}(q)$ there exists a set term b_x such that $x = b_x$ is in q. Let $S = \{x = b_x \mid x \in \mathcal{V}(q)\}$. S is a system of equations. Clearly S is a subset of q. By the solution lemma, there is a solution f for S. Let B be the set of ordered pairs $(f(u), f(v))$ such that $u = v$ is in q. Let D be the set of ordered pairs $(f(u), f(v))$ such that $u \sqsubseteq v$ is in q. B and D are binary relations on V_A. Since p is a support, B and D satisfy all defining clauses of the bisimulation and subsumption relation on V_A, respectively. Since the maximum bisimulation on V_A is the identity relation [1], we obtain $B \subseteq \{(x, x) \mid x \in V_A\}$. Similarly D is a subset of the hereditary subset relation on V_A. Hence we obtain (2).

(2)$\Longrightarrow$(1): Suppose f is a solution for p in V_A. Let P be the set of equations $u = v$ such that u and v are in $\mathit{fld}(p)$ and $f(u)$ and $f(v)$ are the same element of V_A. Let Q be the set of subsumptions $u \sqsubseteq v$ such that u and v are in $\mathit{fld}(p)$ and $f(u)$ is a hereditary subset of $f(v)$ in V_A. Let $q = P \cup Q \cup \{x = f(x) \mid x \in \mathcal{V}(p)\}$. It is easily checked that $p \subseteq q$ and q has a normal support in L_A. □

Note that this theorem does not assure that any normal support in the language L_{A_ω} has a solution in V_{A_ω} even if every atomic constraint in the given support is finitary. We will discuss this later. By a *positive support* we mean a support without negative literals, i.e., consisting of only atomic constraints.

Lemma 3.1 (Binding Lemma) *Any positive support has a positive normal support.*

Proof Let p be a positive support. Let x be any unbound parameter of p, i.e., we suppose that if $x = u$ in p then u is a parameter. Let L_x be the set of terms u such that $u \sqsubseteq x \in p$. Similarly, let U_x be the set of terms u such that $x \sqsubseteq u \in p$. As p is a positive support, if $x = y$ is in p for some parameter y, then $L_x = L_y$ and $U_x = U_y$. Let B_x be the union of sets in L_x and

$$p' = p \cup \{u \sqsubseteq B_x \mid u \in L_x\} \cup \{B_x \sqsubseteq u \mid \ u \in U_x\} \cup \{B_x = B_x\} \cup \{B_x \sqsubseteq B_x\}.$$

As p is a positive support it is certain that p' satisfies all the constraint rules. For example, we can see that p' satisfies constraint rule (14) as follows. Let $y \in B_x$ and suppose that $B_x \sqsubseteq v \in p'$ for some set v. By definition of B_x $x \sqsubseteq v \in p$ and $y \in u$ and $u \sqsubseteq x \in p$ for some u. As p is a support $u \sqsubseteq v \in p$ and $y \sqsubseteq z \in p$ for some $z \in v$. Hence, p' satisfies constraint rule (14). Thus we can prove that p' is a support. Hence the reflexive closure p'' of $p' \cup \{x = B_x\}$ w.r.t. the equality $=$ is a support. Note that $\mathcal{V}(p) = \mathcal{V}(p'')$. By repeating this extension procedure for each free parameter in p we get a monotone increasing sequence of supports of p. It is easy to see that the union of supports in the sequence is a normal support of p. □

Example 3.2 Take a positive support $\{\{x\} \sqsubseteq x\}$. Then the above proof constructs a positive normal support $\{\{x\} \sqsubseteq x, x = \{x\}\}$ of the given support. □

I do not know whether 'positive' in the lemma can be dropped or not. In this case at least, the above proof does not work. To see this, consider the support $\{\{x\} \sqsubseteq x, x \neq \{x\}\}$. The normal constraint constructed by the method in the proof is $\{\{x\} \sqsubseteq x, x = \{x\}, x \neq \{x\}\}$, which is not a support.

Theorem 3.6 *Every positive support in L_A is solvable in V_A.*

Proof Let p be a positive support. p has a positive normal support p' by the binding lemma. As Barwise's unification theorem 3.5 can be applied to arbitrary positive supports p' has a solution in V_A. □

Theorem 3.6 together with Proposition 3.4 gives a decision procedure for subsumption problems. More precisely, given a set of positive equations and subsumptions, it is decidable whether the set has a solution or not.

Remark The distinction between atom and parameter is important. Consider the constraint $a = \{x\}$. If a is an atom then, by definition, there is no support for this constraint. In fact there is no solution in this case. Otherwise, if a is a parameter, clearly $a = \{x\}$ has a support and is in fact solvable. □

Remark Applying this theorem, I found an elegant decision procedure for a class of extensional subsumption problems over feature structures [60]. The procedure in [60] is a hyperset-theoretical positive answer to the problem class, which was first solved by Dörre (April 1990). On the contrary it was proved in [29] that the intensional version of subsumption problems is undecidable. □

Lemma 3.2 *Given a countable clause c of finitary literals, the following are equivalent.*

(1) Every finite subset of c has a support.

(2) c has a support.

Proof (2) $\Longrightarrow$ (1): Obvious.

(1) $\Longrightarrow$ (2): Suppose (1) is true. If c is finite, (2) is obviously true. So we assume c is infinite: $c = \{q_n \mid 0 \leq n\}$, where q_i are literals. Let S_n be the set of small supports of $\{q_0, \cdots, q_n\}$. It follows from the assumption that S_n is a nonempty finite set and that for any $n \geq 1$ and $p \in S_n$ there is a support $p' \in S_{n-1}$ such that $p' \subseteq p$. For each $n \geq 1$, let T_n be the set of all sequences $\{p_j\}_{0 \leq j \leq n}$ of supports such that $p_j \in S_j$ and $p_{j-1} \subseteq p_j$ ($j \neq 0$). Clearly, every T_n is finite and nonempty. Also, T_n and T_m are disjoint from each other for $n \neq m$. Let T be the union of the family $\{T_n\}_{n \geq 0}$. Let R be a binary relation on T such that $R(p, p')$ is true iff p is an initial segment of p', i.e., p' is an extension of p as a sequence. It is easy to see that (T, R) is a tree with finite branches at each node. Applying the standard argument of König's lemma to this tree, we can construct an infinite sequence $\{p_j\}_{0 \leq j}$ such that $p_j \in S_j$ and $p_{j-1} \subseteq p_j$ ($j \neq 0$). Let p be the union of the sequence: $p = \bigcup\{p_j \mid 0 \leq j\}$. It is easily checked that p is a support of c. □

Proposition 3.7 *The following are equivalent for any countable set p of finite atomic constraints:*

(1) Every finite subset of p has a support.

(2) p has a normal support.

Proof This proposition is a direct combination of the previous two lemmas. □ The normal support p in condition (2) may have infinite terms.

Proposition 3.8 *For any normal support p the following are equivalent.*

(1) p has a solution in V_A.

(2) For each negative literal $l \in p$, $p' \cup \{l\}$ has a solution in V_A, where p' is the set of positive literals in p.

Proof (1) $\Longrightarrow$ (2): Obvious.

(2) $\Longrightarrow$ (1): Suppose (2) is true. Let p' be the set of positive literals in p. As p is normal, p' is also normal and $\mathcal{V}(p) = \mathcal{V}(p')$. By the solution lemma, there is a unique solution f of p'. From the hypothesis, the uniqueness of f, and $\mathcal{V}(p) \subseteq dom(f)$, it follows that f is a solution of $p' \cup \{l\}$ for any negative literal l in p. Therefore, f is a solution of p. □

Lemma 3.3 *For a positive normal support p and an atomic constraint d such that $\mathcal{V}(d) \subseteq \mathcal{V}(p)l$, the following are equivalent.*

(1) $p \cup \{\neg d\}$ has a solution in V_A.

(2) $p \cup \{d\}$ has no solution V_A.

Proof As p is a normal positive support, there is a unique solution f of p in V_A such that $dom(f) = \mathcal{V}(p)$ and $\mathcal{V}(d) \subseteq dom(f)$.

(2) $\Longrightarrow$ (1): Suppose (2) is true. Then f does not satisfy d. Otherwise, f would be a solution of $p \cup \{d\}$ in $V_{A\omega}$, which contradicts (2). So f is a solution of $\neg d$. Hence, f is a solution of $p \cup \{\neg d\}$.

(1) $\Longrightarrow$ (2): Suppose (1) is true but (2) is not, i.e., there is a solution g of $p \cup \{d\}$. So g must satisfy d. On the other hand, it follows from (1) that f must satisfy $\neg d$. As f and g are solutions of p and $\mathcal{V}(d) \subseteq \mathcal{V}(p)$, f and g must coincide on $\mathcal{V}(d)$, which is a contradiction. Hence (2) must be the case. □ Lemma 3.3 and the binding lemma (Lemma 3.1) are a basis of a unification algorithm. To see this, let p and d be a finite normal support and an atomic constraint, respectively, such that $\mathcal{V}(d) \subseteq \mathcal{V}(p)$. Then it is decidable by a naive saturation method whether $p \cup \{\neg d\}$ has a solution or not by checking the solvability of $p \cup \{d\}$, which can be determined by Barwise's simulation pair theorem. This remark will be used in a later section for application to the usual term structures and record structures.

Lemma 3.4 (Compactness Lemma) *For a positive constraint $c = \{d_n \mid 0 \leq n\}$, the following are equivalent.*

(1) Each finite subset of c has a solution in V_A.

(2) c has a solution in V_A

Proof (2) $\Longrightarrow$ (1): Obvious.

(1)$\Longrightarrow$ (2): Suppose (1) is true. By the binding lemma, since every finite subset of c has a support, c has a normal support p. Then, by Barwise's theorem, there is a solution of p in V_A. □

The following lemma is a corollary of the compactness lemma (Lemma 3.4).

Lemma 3.5 *For a set c of atomic constraints in L_A, the following are equivalent.*

(1) There is no support of c.

(2) There is some finite set c' of c such that there is no support of c'.

Now we address the problem of compactness of $V_{A\omega}$ w.r.t. $L_{A\omega}$. Barwise's theorem is used again. Consider a countable family of subsumptions $\{a_i\} \sqsubseteq x$ for $i > 0$, where a_i are distinct atoms. Then every a_i must be an element of x. So x must be infinite. This means $V_{A\omega}$ is not compact in general w.r.t. subsumption. The theorem below gives more precise information about this problem. In the proof, we treat only positive constraint, which is a countable set of equations and subsumptions between *well-founded* elements of $V_{A\omega}[X]$. Note that each element of a constraint is finitely representable as a tree.

Theorem 3.9 *Let A be a set of atoms. $V_{A\omega}$ is compact w.r.t. the class of supports in $L_{A\omega}$ iff $A = \emptyset$.*

Proof Unlike the other parts of the chapter, set terms here may not be flat. The proof is divided into two cases, (1) and (2).

(1) Suppose $A \neq \emptyset$ and $a \in A$. Let Ω_a be the unique solution of $z = \{a, z\}$. Define a countable sequence b inductively as follows.

(1) $b_0 = \{\Omega_a\}$.

(2) $b_n = \{a, b_{n-1}\}$. $(n > 0)$

Define a countable sequence d inductively as follows:

(1) $d_0 = \{a\}$.

(2) $d_n = \{d_{n-1}\}$. $(n > 0)$

Let $e_n = \{d_n, b_n\}$ for $n \geq 0$. It is clear that if $d_n \sqsubseteq u$ then u must have the 'path' d_n by definition of d_n. If $u \sqsubseteq e_n$ and u has the path d_n then u cannot have any path $d_{n'}$ for $n' \neq n$. Hence for $n \neq n'$ if $d_n \sqsubseteq u \sqsubseteq e_n$ and $d_{n'} \sqsubseteq u' \sqsubseteq e_{n'}$ then $u \neq u'$. Consider the positive support s as the closure of the constraint

$$\{d_n \sqsubseteq x \mid n \geq 0\} \cup \{x \sqsubseteq e_n \mid n \geq 0\}$$

where x is a parameter. By Theorem 3.6 the support s has a solution in V_A. It is clear that every subset of s has a solution in $V_{A\omega}$. But by the remark above, any solution for x must have an infinite number of elements. So it is impossible for s to have a solution in $V_{A\omega}$. Therefore, $V_{A\omega}$ is not compact w.r.t. $L_{A\omega}$.

(2) Suppose $A = \emptyset$. For $u, v \in V_{A\omega}$, use König's lemma to easily prove that $u \sqsubseteq v$ iff the 'height' of u is less than or equal to that of v, where the height of non-well-founded sets is ∞ and the height of $\emptyset$ is 0. The height of other nonempty set is defined to be 1 + the maximum of the heights of members of the set.

Let s be a clause. Consider functions h that assign nonnegative integers including ∞ to each parameter x, y appearing in s in such a way that

- $h(y) < h(x)$ if $u \sqsubseteq x \in s$ and $y \in u$.
- $h(y) < h(x)$ if $u = x \in s$ and $y \in u$.
- $h(y) \leq h(x)$ if $y \sqsubseteq x \in s$.
- $h(y) = h(x)$ if $x = y \in s$.

where we use conventions $\infty < \infty$ and $n < \infty$ for finite integers n. Intuitively, $h(x)$ is the 'height' of a solution for x satisfying s. Let H be the set of such height functions on s and let h_s be the least function in H, which is defined so that $h_s(x)$ has the least integers in $\{h(x) \mid h \in H\}$. It is easy to see that $h_s \in H$. Define a sequence p inductively as follows.

- $p_\infty = \{p_\infty\}$, i.e., Ω.
- $p_0 = \emptyset$.
- $p_n = \{p_{n-1}\}$. $(n > 0)$

Consider $s' = s \cup \{x = p_{h_s(x)} \mid x \text{ is unbound in } s\}$. As s' is clearly a support, s' has a solution in $V_{A\omega}$, say f. It follows from the definition of f that the height of $f(u)$ is $h_s(u)$, for u appearing in s. For $u \sqsubseteq v \in s$, it follows from the remark above on the relationship between the subsumption and the height that $h_s(u) \leq h_s(v)$ and therefore $f(u) \sqsubseteq f(v)$. Hence, s is solvable in $V_{A\omega}$. Therefore, $V_{A\omega}$ is compact w.r.t. $L_{A\omega}$. □

Canonical Constraints

We treat a more general case of compactness in which negative information is involved. Note that a canonical constraint does not always have a solution. For example the constraint consisting of the literals

$$x \neq y, x = \{x\}, y = \{y\}$$

is a canonical support but clearly has no solution.

Theorem 3.10 (Canonical Compactness) *For a canonical set c of literals in $L_{A\omega}$, the following are equivalent.*

(1) c has a solution in $V_{A\omega}$.

(2) Every finite subset of c has a solution in $V_{A\omega}$.

Proof Let $c = p \cup p'$ where p and p' are the sets of positive and negative literals of c, respectively.

(2) $\Longrightarrow$ (1): Suppose (2) is true, i.e., every subset of c has a solution in $V_{A\omega}$. So every subset of p has a solution in $V_{A\omega}$. It follows from the compactness of positive supports that there is a solution f in $V_{A\omega}$ of p. We show that f is also a solution of p' in $V_{A\omega}$. As c is canonical, then for each negative literal d in p' there is a finite subset q of c that is a normal constraint containing d. By condition (2) q has a solution in $V_{A\omega}$, say h. As the set of positive literals in q is normal and is a subset of p, it follows from the uniqueness of a solution that f and h must coincide on each parameter of q. Thus f satisfies every negative constraint in p'. Therefore f is a solution of c in $V_{A\omega}$.

(1) $\Longrightarrow$ (2): Obvious. □

Theorem 3.11 (Canonical Support) *Every canonical support in $L_{A\omega}$ is solvable in $V_{A\omega}$.*

Proof Let $c = p \cup p'$ be a canonical support where p and p' are the positive and negative parts respectively. By Theorem 3.6, p has a solution in $V_{A\omega}$, say f. As c is canonical, f is defined on every parameter appearing in p'. Suppose that f satisfies $d \in p'$. Then $\{d\} \cup p$ has a support. As c is canonical there is a normal finite subset p'' of p such that $\{d\} \cup p$ is also normal. Hence,

$\{d\} \cup p''$ has a support, which contradicts that c is a support. So f is not a solution of d. Hence f is a solution of $\neg d$. Therefore f is a solution of c. □

By saying that a constraint c has a solution $x = t$ we mean that there is a solution f of c such that $f(x) = t$. Let c be a constraint in $L_{A\omega}$, x the distinguished parameter of c, and t a hyperset of $V_{A\omega}$. Then a *separating cover of t from c* is a clause s in $L_{A\omega}$ such that s has a solution $x = t$, but no common solution with c.

Theorem 3.12 (Solution Compactness) *$V_{A\omega}$ is solution compact w.r.t. $L_{A\omega}$.*

Proof Let c be a finite constraint in $L_{A\omega}$. By transforming c into disjunctive normal form we can assume without loss of generality that

$$c = c_1 \vee c_2 \vee \cdots \vee c_n$$

for some positive integer n. Let x be the distinguished parameter in c. Assume $t \in V_{A\omega}$ such that the constraint c has no solution for $x = t$.

Our goal is to find a finite separating cover s of t from c. If c is unsolvable in $V_{A\omega}$ we can take the constraint $\{\emptyset = \emptyset\}$, for example, as the finite separating cover of t from c. So we assume that c is solvable. Let E_t be a system of equations

$$x_0 = b_0, x_1 = b_1, \cdots, x_m = b_m, \cdots$$

that defines t as the unique solution for x_0. Note that, as $t \in V_{A\omega}$, clearly E_t can be constructed so that E_t is a constraint in $L_{A\omega}$. It suffices to show that for each $1 \leq i \leq n$, if c_i is solvable in $V_{A\omega}$, there is a finite subset s_i of E_t such that s_i is a separating cover of t from c_i because, if so, then it clearly follows that $s_1 \cup \cdots \cup s_n$ is a separating cover of t from c. So let i be any integer $1 \leq i \leq n$ such that c_i is solvable in $V_{A\omega}$. Let p and p' be the positive and negative parts of c_i respectively. We show that there is a subset of E_t that is a separating cover of t from $p \cup p'$. The proof is divided into two cases (1) and (2).

(1) Suppose that the constraint $\{x = x_0\} \cup E_t \cup p$ has no solution. Then there is some finite subset E' of E_t such that $\{x = x_0\} \cup E' \cup p$ has no solution, for otherwise, by the compactness lemma (Lemma 3.4), there must be a solution of $\{x = x_0\} \cup E_t \cup p$. But this means that $x = t$ is a solution of p, which contradicts the assumption. Now we have obtained a separating cover $s = \{x = x_0\} \cup E'$ of t from $p \cup p'$.

(2) Suppose that the constraint $\{x = x_0\} \cup E_t \cup p$ has a solution. We divide this case into two parts.

(2.1) Suppose that $E_t \cup p \cup \{d\}$ has a support for some negative literal $\neg d$ in p'. Then $p \cup \{d\}$ is a separating cover of t from $p \cup p'$.

(2.2) Suppose that $E_t \cup p \cup \{d\}$ has no support for any literal $\neg d$ in p'. Let $\neg d$ be any literal in p'. Then no solution of $\{x = t\} \cup p$ can be extended to that of $\{d\}$. So any solution of $\{x = t\} \cup p$ satisfies $\neg d$. Hence any solution of $\{x = t\} \cup p$ satisfies p'. So $x = t$ is an solution of $p \cup p'$. This is a contradiction. Therefore, the last case is impossible. This concludes the proof. □

The solvability of canonical constraints in $L_{A\omega}$ is characterized using a syntactic notion of the support in the following theorem:

Theorem 3.13 *Let $c = q \cup q'$ be a canonical constraint in $L_{A\omega}$, with positive part q and negative part q'. Then the following are equivalent.*

(1) c is solvable in $V_{A\omega}$.

(2) q has a normal support p such that $p \cup q'$ is a canonical support.

Proof Use Lemma 3.3. □ Now we establish the satisfaction completeness of unification theory as a direct consequence of the results obtained so far. Let $\mathcal{T}_{A\omega}$be the theory of ZFCA plus the theory of the constraint language $L_{A\omega}$, taking the latter theory to be the set of $\tilde{\exists} p$, where p is a canonical support.

Theorem 3.14 (Satisfaction Completeness) *For any canonical constraint c in $L_{A\omega}$*

$$\mathcal{T}_{A\omega} \models \tilde{\forall} \neg c \qquad \textit{whenever not} \quad \mathcal{T}_{A\omega} \models \tilde{\exists} c.$$

Proof We prove the contrapositive. Suppose that $\mathcal{T}_{A\omega} \models \tilde{\forall} \neg c$ is not the case. Then there must be a model of $\mathcal{T}_{A\omega}$ that satisfies c. Hence, by Theorem 3.13, c has a canonical support c'. Then, by the canonical support theorem (Theorem 3.11), c' must be solvable in any model of $\mathcal{T}_{A\omega}$. Hence, so is c. Therefore, we get $\mathcal{T}_{A\omega} \models \tilde{\exists} c$. □

We have proved the necessary properties required by the constraint logic programming scheme. Thus we have CLP($L_{A\omega}$), for which also we write loosely CLP(AFA).

3.4 Coinductive Semantics of Horn Clauses

We turn to the constraint logic programming (CLP) over the hyperset domain $V_{A\omega}[X]$, where sets A and X are assumed to be large enough to include all necessary atoms and parameters respectively. We start by defining two semantic domains. One domain consists of *computation trees* for the operational semantics and the other one consists of *solution trees* for the declarative semantics. Intuitively speaking, a computation tree is obtained from some solution tree by 'forgetting' information about solutions coded in the solution tree. Elements of the two domains are coded in hypersets in a more natural way than the usual representation by state transition nets.

For a given Horn clause program, the two semantic domains above are defined coinductively in the domain $V_{A\omega}$. Soundness and completeness are formalized as a relation between the computation trees and the solution trees. In fact, this relation is a 'forgetful' projection from solution trees into computation trees. This simulation relation is defined coinductively in a straightforward way.

Throughout this section, A and X denote sets of atoms and parameters respectively. It follows from this assumption that various constructions given in this section never form sets but never proper classes unless mentioned otherwise.

3.4.1 Horn Clause with Constraints

We introduce a class of Horn clauses with constraints over the constraint language L_{A_ω}. Let A and X be *sets* of atoms (= constants) and parameters, respectively, as before. Let Π be a set of *predicate symbols*. If $p \in \Pi$ and $x_1, \cdots, x_n$ are parameters, then $p(x_1, \ldots, x_n)$ is an *atomic goal*. A *goal* is a finite set of atomic goals. A *head* is a form $p(x_1, \ldots, x_n)$ where p is a predicate symbol and x_i $(i \neq j)$ are distinct parameters. A *constraint Horn clause* is a triple (h, c, g), where h is a head, c is a constraint, and g is a goal. We write

$$h : -c \mid b_1, \cdots, b_n$$

for $(h, c, \{b_1, \cdots, b_n\})$. A *program* is a *finite* set of constraint Horn clauses, each clause in it is called *a program clause*.

We use a simple running example to illustrate the idea of the coinductive semantics defined below. Let Ψ_0 be the program consisting of three constraint Horn clauses:

(C1) $bit(0)$.

(C2) $bit(1)$.

(C3) $stream(x)$:- $x = (b, y) \mid bit(b), stream(y)$.

Intuitively, the coinductive semantics of the program Ψ_0 is a function $\mathcal{I}_0$ such that $\mathcal{I}_0(bit) = \{0, 1\}$ and $\mathcal{I}_0(stream)$ is the set of streams over 0 and 1. Note that the standard least semantics of Ψ_0 is a function $\mathcal{I}'_0$ such that $\mathcal{I}'_0(bit) = \{0, 1\}$ but $\mathcal{I}'_0(stream) = \emptyset$.

3.4.2 Computation Trees

We introduce notions of a *state* and *computation tree* to give an operational semantics of the program. Computation trees only have information about steps of successful computations. In other words, they have no explicit information about failure branches.

Definition 3.24 A *computation state* (state for short) is a triple (c, g, ξ) of a constraint c, a goal g, and a set ξ of parameters such that c has a support and $\mathcal{V}(g) \cup \mathcal{V}(c) \subseteq \xi$. □

A set of parameters that is the last component of a state may be omitted when the context is clear. Given a goal g, the *initial state* is the state $(\emptyset, g, \mathcal{V}(g))$, where the empty set $\emptyset$ means the 'true' constraint that always holds. As the program Ψ is finitary and X is finite, the collection of states for the program Ψ forms a set.

Example 3.3 The triple

$$(\{x = (z, y), x = y\}, \{bit(z), stream(x), stream(y)\}, \{x, y, z, u\})$$

is a state. Note that u is a parameter that does not appear in the first two components of the triple. □

Definition 3.25 A *choice* γ *on* g is a function that assigns a program clause to each atomic goal in g. That is, $\gamma\colon g \to \Psi$, where Ψ is a program fixed in the context. □

Example 3.4 Let g be the goal $\{bit(z), stream(x), stream(y)\}$. The mapping γ which assigns (C1), (C2), (C2) to the atomic goals $bit(z)$, $stream(x)$, $stream(y)$, respectively, is a choice on g, where (C1) and (C2) are clauses in Ψ_0. □

Let S_g denote the set of choices on g. Then S_g is finite because both g and the program Ψ are finite.

Definition 3.26 Let $s = (c, g, \xi)$ and $s' = (c', g', \xi')$ be two states. Let γ be a choice on g. A triple (s, γ, s') is a *transition*, written as $s \xrightarrow{\gamma} s'$, if for each atomic goal $a \in g$ there is a 'renamed version' C_a: $h_a : -c_a \mid g_a$ of the clause $\gamma(a) \in \Psi$ such that

(1) $\mathcal{V}(C_a) \cap \mathcal{V}(C_{a'}) = \emptyset$. $(a \neq a')$

(2) $\xi \cap \mathcal{V}(C_a) = \emptyset$.

(3) $g' = \bigcup\{g_a \mid a \in g\}$.

(4) $c' = c \wedge \bigwedge\{\{a = h_a\} \wedge c_a \mid a \in g\}$.

(5) c' has a support.

(6) $\xi' = \xi \cup \bigcup\{\mathcal{V}(C_a) \mid a \in dom(\gamma)\}$.

□

Example 3.5

$$(\{x = y, x = (b, y)\}, \{stream(y)\}, \{x, y, b\}) \xrightarrow{\gamma_0}$$
$$(\{x = y, x = (b, y), u = y, u = (d, v)\}, \{bit(d), stream(v)\}, \{x, y, b, u, v, d\})$$

where γ_0 is the choice that assigns the clause (C3) of Ψ_0 to the atomic goal $stream(y)$. u, v, d are the new parameters generated for this transition. □

We formalize the set of all possible 'computations' for the program.

Definition 3.27 C_Ψ is the largest set M such that if $x \in M$ then $x = (s, b)$ for some state s and a partial function $b\colon S_g \to M$, where

(1) g is the goal component of the state s.

(2) $s \xrightarrow{\gamma} s'$ if $\gamma \in dom(b)$ and $b(\gamma) = (s', b')$ for some b'.

□

As the collection of states of the program is a set and the number of branches is finite at each node, C_Ψ for the program Ψ forms a set.

Definition 3.28 A binary order $\prec$ is the largest relation on $\mathcal{C}_\Psi$ such that if $x \prec y$ then the following hold for some s, b, b':

(1) $x = (s, b)$ and $y = (s, b')$.

(2) $dom(b) \subseteq dom(b')$.

(3) If $\gamma \in dom(b)$ then $b(\gamma) \prec b'(\gamma)$.

□

Proposition 3.15 *$(\mathcal{C}_\Psi, \prec)$ is a partial order structure.*

Proof Let R be a binary relation on $(\mathcal{C}_\Psi, \prec)$ such that xRy iff $x \prec y$ and $y \prec x$. It is easy to check that R is a bisimulation on $(\mathcal{C}_\Psi, \prec)$. So the preorder $\prec$ is a partial order. □

Definition 3.29 A *computation tree* is a maximal element of $(\mathcal{C}_\Psi, \prec)$. □

As will be shown later in Lemma 3.10, maximal computation tree in $(\mathcal{C}_\Psi, \prec)$ exists.

Definition 3.30 A minimal element of $(\mathcal{C}_\Psi, \prec)$ is called a *path*. □

3.4.3 Coinductive Semantics

Definition 3.31 An *interpretation* of the program is a function $\mathcal{I}$ which assigns a subset of $(V_{A\omega})^n$ to predicate symbols p, where n is the arity of p. Interpretations are partially ordered by pointwise inclusion. That is, $\mathcal{I} \leq \mathcal{I}'$ iff $\mathcal{I}(r) \subseteq \mathcal{I}'(r)$ for each predicate symbol of the program.

□

Definition 3.32 A *model* of the program Ψ is an interpretation $\mathcal{I}$ of Ψ such that for any $(a_1, \cdots, a_n) \in \mathcal{I}(p)$ there exists a program clause

$$p(x_1, \ldots, x_n) : -c \mid g$$

and assignment f such that

(1) $f(x_i) = a_i$ for $1 \leq i \leq n$.

(2) $V_{A\omega} \models_f c$.

(3) $(f(z_1), \ldots, f(z_m)) \in \mathcal{I}(q)$ for each atomic goal $q(z_1, \ldots, z_m) \in g$.

where m is the arity of the predicate symbol q. □

Since the Horn clause program Ψ works as a monotone operator T_Ψ on interpretations $\mathcal{I}$ w.r.t. $\leq$, it follows from the general theory of Aczel [1] that there exists a largest model, denoted by $\mathcal{M}_\Psi$, of the program Ψ. The largest model $\mathcal{M}_\Psi$ of the program Ψ is also called the *coinductive semantics* of the program. Note that $V_{A\omega}$ is a set. So interpretations are also sets. Hence $\mathcal{M}_\Psi$ is a also set.

3.4.4 Solution Trees

Definition 3.33 An assignment f is a *solution* of (c, g) if $V_{A\omega} \models_f c$ and $\mathcal{M}_\Psi \models_f g$, where c and g are a constraint and a goal, respectively, such that $\mathcal{V}(c) \cup \mathcal{V}(g) \subseteq dom(f)$. □

Definition 3.34 $\mathcal{A}_\Psi$ is the set of triples (f, c, g) such that f is a solution of (c, g). □

Definition 3.35 A binary relation $\longrightarrow$ is the largest relation on $\mathcal{A}_\Psi$ such that, if $(f, c, g) \longrightarrow (f', c', g')$ is defined, there is a choice γ on g such that for any $a \in g$ there is a 'renamed program clause' C_a: $h_a :- c_a \mid g_a$ of $\gamma(a)$ that satisfies the following.

(1) f' is an extension of f.

(2) $g' = \bigcup\{g_a \mid a \in g\}$.

(3) $c' = c \wedge \bigwedge\{\{a = h_a\} \wedge c_a \mid a \in g\}$.

(4) $\mathcal{V}(C_a) \cap \mathcal{V}(C_{a'}) = \emptyset$ $(a \neq a')$.

(5) $dom(f) \cap \mathcal{V}(C_a) = \emptyset$ for any $a \in g$.

We write $(f, c, g) \xrightarrow{\gamma} (f', c', g')$ indicating γ explicitly. □

Lemma 3.6 *If $s = (f, c, g)$ is in $\mathcal{A}_\Psi$ and g is not empty then $s \xrightarrow{\gamma} s'$ for some s' in $\mathcal{A}_\Psi$ and choice on g.*

Proof Let $a = p(y_1, \cdots, y_n)$ be an atomic goal in g. f is a solution of a by the hypothesis. Then by the definition of a solution, there is a program clause γ_a, a fresh copy h_a:- $c_a \mid g_a$ of it, and some extension f_a of f such that f_a is a solution of both (c_a, g_a) and $a = h_a$. Let

$$s' = (f', c \wedge \bigwedge\{\{a = h_a\} \wedge c_a \mid a \in g\}, \bigcup\{g_a \mid a \in g\})$$

where f' is an extension of any f_a for $a \in g$. So, by definition, we get $s \xrightarrow{\gamma} s'$, where γ is a choice on g such that $\gamma(a) = \gamma_a$ for $a \in g$. □

Definition 3.36 Given the program Ψ, $\mathcal{S}_\Psi$ is the largest collection $M \subset V_{A\omega}[X]$ such that if $x \in M$ then $x = (s, b)$ for some $s = (f, c, g) \in \mathcal{A}_\Psi$ and a function b such that the following hold.

(1) $dom(b)$ consists of choices on g.

(2) $ran(b) \subseteq M$.

(3) $s \xrightarrow{\gamma} s'$ if $b(\gamma) = (s', b')$ for some b'.

□

$\mathcal{S}_\Psi$ is a set for a similar reason to $\mathcal{C}_\Psi$'s.

Definition 3.37 $\prec$ is the largest binary relation on $\mathcal{S}_\Psi$ such that if $x \prec y$ then the following hold.

(1) $x = (s, b)$ and $y = (s, b')$ for some s, b, b'.

(2) $dom(b) \subseteq dom(b')$.

(3) If $\gamma \in dom(b)$ then $b(\gamma) \prec b'(\gamma)$.

□

$(\mathcal{S}_\Psi, \prec)$ is a partial order structure for a similar reason to $\mathcal{C}_\Psi$'s above.

Definition 3.38 A *solution tree* is a maximal element of $(\mathcal{S}_\Psi, \prec)$. □

It will be shown in Lemma 3.10 that there exist maximal solution trees in $(\mathcal{S}_\Psi, \prec)$.

Definition 3.39 A minimal element in $(\mathcal{S}_\Psi, \prec)$ is called a *path* . □

We illustrate a computation tree t for the goal $\{stream(x)\}$ of the program Ψ_0. t is an infinite binary tree. t has all the information about how each stream is generated by the program Ψ_0. The general form of states at nodes ν of t is a triple (c_ν, g_ν, X_ν), where

$$\begin{aligned} c_\nu &= \{x = (b_1, x_1), x_1 = (b_2, x_2), \cdots, x_{n-1} = (b_n, x_n), b_1 = \beta_1, \cdots, b_{n-1} = \beta_{n-1}\} \\ g_\nu &= \{bit(b_n), stream(x_n)\} \\ X_\nu &= \{x, b_1, x_1, \cdots, b_n, x_n\} \end{aligned}$$

where β_i is either 0 or 1. The other symbols are parameters. The state means that a finite stream $\beta_1\beta_2 \cdots \beta_{n-1}$ has been produced as an initial finite segment of some stream. The node ν has two successor nodes ν_0 and ν_1, whose states are

$$(c_\nu \cup \{x_n = (b_{n+1}, x_{n+1}), b_n = 0\}, \{bit(b_{n+1}), stream(x_{n+1})\}, X_\nu \cup \{b_{n+1}, x_{n+1}\})$$

$$(c_\nu \cup \{x_n = (b_{n+1}, x_{n+1}), b_n = 1\}, \{bit(b_{n+1}), stream(x_{n+1})\}, X_\nu \cup \{b_{n+1}, x_{n+1}\})$$

respectively.

An example of a solution tree for the same goal $\{stream(x)\}$ is a tree t' with a single path. Take the union of constraints on the single path to obtain a system of equations:

$$\{x = (b_1, x_1), x_1 = (b_2, x_2), \cdots, x_{n-1} = (b_n, x_n), \cdots, b_1 = \beta_1, \cdots, b_{n-1} = \beta_{n-1}, \cdots\}$$

Let f be the unique solution of the union. The solution tree has the following information at each node:

$$(f', c', \{bit(b_n), stream(x_n)\}, \{x, b_1, x_1, \cdots, b_n, x_n\}),$$

where f' is a restriction of f to the set $\{x, b_1, x_1, \cdots, b_n, x_n\}$, and c' is the set

$$\{x = (b_1, x_1), x_1 = (b_2, x_2), \cdots, x_{n-1} = (b_n, x_n), b_1 = \beta_1, \cdots, b_{n-1} = \beta_{n-1}\}.$$

Lemma 3.7 *If $x = (s, b)$ is a path such that $dom(b) \neq \emptyset$, then there are some γ, s', and b' such that the following hold:*

(1) $dom(b) = \{\gamma\}$.

(2) $b(\gamma) = (s', b')$ *is a path.*

(3) $s \xrightarrow{\gamma} s'$.

Proof It is obvious by the definition of a path. □

Lemma 3.8 *Let Ψ be a program and p be a predicate. Then the following are equivalent.*

(1) $(a_1, \cdots, a_n) \in \mathcal{M}_\Psi(p)$.

(2) There is a path $((f, \emptyset, \{p(x_1, \cdots, x_n)\}), b) \in \mathcal{S}_\Psi$ *for some b such that* $f(x_i) = a_i$ $(1 \leq i \leq n)$.

Proof (1)$\Longrightarrow$ (2): Suppose (1) is true. Then by repeated application of Lemma 3.6 we can make a sequence of possibly countably infinite length in $\mathcal{A}_\Psi$

$$s_0 \xrightarrow{\gamma_0} s_1 \xrightarrow{\gamma_1} \cdots \xrightarrow{\gamma_{n-1}} s_n \xrightarrow{\gamma_n} \cdots$$

where $s_0 = (f, \emptyset, \{p(x_1, \cdots, x_n)\}) \in \mathcal{A}$ and $f(x_i) = a_i$ $(1 \leq i \leq n)$. The path z_0 obtained by solving the following equations satisfies (2).

$$\begin{aligned} z_0 &= (s_0, \{(\gamma_0, z_1)\}) \\ z_1 &= (s_1, \{(\gamma_1, z_2)\}) \\ &\vdots \\ z_n &= (s_n, \{(\gamma_n, z_{n+1})\}) \\ &\vdots \end{aligned}$$

(2) $\Longrightarrow$ (1): Suppose (2) is true. Then f is a solution of $p(x_1, \ldots, x_n)$ by definition. Hence, $(a_1, \cdots, a_n) = (f(x_1), \cdots, f(x_n)) \in \mathcal{M}_\Psi(p)$. □

Lemma 3.9 $(\mathcal{S}_\Psi, \prec)$ *is chain complete.*

Proof Suppose we are given the following sequence of elements of $\mathcal{S}_\Psi$:

$$x_0 \prec x_1 \prec \cdots \prec x_n \prec \cdots.$$

Let us define a sequence X_i $(i \geq 0)$ inductively:

- $X_0 = \{x_i \mid i \geq 0\}$.
- $X_{i+1} = \{b(x) \mid \exists s\ (s, b) \in X_i, x \in dom(b)\}$.

Let $X = \bigcup\{X_i \mid i \geq 0\}$ and $\Lambda = \bigcup\{dom(b) \mid \exists s\ (s, b) \in X\}$. Let Λ^* be the set of finite sequences over Λ. Let us define a partial operation $\alpha \in \Lambda^*$ on $x = (s, b) \in X$ inductively:

- $x(\varepsilon) = x$.
- $x(\gamma\alpha) = b(\gamma)(\alpha)$

where ε is the empty sequence and $\gamma \in \Lambda$. Then consider the system of equations for $\alpha \in \Lambda^*$

$$y_\alpha = (a_\alpha, Y_\alpha)$$

such that

- there is some x_i in the given sequence such that $x_i(\alpha) = (a_\alpha, b)$ for some b.
- $(\gamma, y_{\alpha\gamma}) \in Y_\alpha$ iff there is some x_j such that $x_j(\alpha\gamma)$ is defined.

This system of equations is well defined though some $\alpha \in \Lambda$ may fail to have a corresponding equation. By the solution lemma, there is a unique solution of the system. Let y be the solution to y_ε. Now we show that y is the least upper bound of the family x_i $(0 \leq i)$. Suppose that z is an upper bound of the family, i.e., $x_i \prec z$ for all $0 \leq i$. By comparing the defining system of equations for z with that for y it follows that $y \prec z$. □

Lemma 3.10 *If x is an element of $\mathcal{S}_\Psi$ there is a maximal element x' in $\mathcal{S}_\Psi$ such that $x \prec x'$.*

Proof Let $X = \{z \in \mathcal{S}_\Psi \mid x \prec z\}$. By Lemma 3.9, any monotone family of elements of X has a limit with respect $\prec$. Hence, by Zorn's lemma, there exists a maximal element x' in X. □

3.4.5 Soundness and Completeness

Let $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$ denote the sets of solution trees and computation trees, respectively. We have remarked above that they are sets when the program Ψ fixed. We formulate soundness and completeness of the semantics of programs in terms of a correspondence between $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$. The correspondence will be called a *simulation*.

Before going into details, we introduce a notion of *hereditary projection* that will be used later. Let $\mathcal{T}(S, \Sigma)$ be the largest class T of sets such that if $x \in T$ then $x = (a, b)$, where $a \in S$ and b is a partial function from Σ to T. An element of T is called a *tree* over S and Σ. Let $X_i = \mathcal{T}(S_i, \Sigma)$ for $i = 1, 2$ and let f be a function from S_1 to S_2. Then there is a function F from T_1 to T_2 such that

$$F((a, b)) = ((f(a), F``b))$$

where $F``b = \{(z, F(b(z))) \mid z \in dom(b)\}$. We write f^* for F. We call f^* a *hereditary projection* if f is a projection.

Definition 3.40 A *simulation between* $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$ is a subset R of $\mathcal{S}_\Psi \times \mathcal{C}_\Psi$ such that if xRy then the following hold for some f, c, g, b, b':

(1) $x = ((f, c, g), b)$ and $y = ((c, g), b')$.

(2) If $dom(b) \neq \emptyset$ then $b(\gamma) R b'(\gamma)$ for *some* $\gamma \in dom(b) \cap dom(b')$.

$\square$

Lemma 3.11 *There is the largest simulation between $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$*

Proof Take the union of all simulations between $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$. $\square$ We say that w *simulates* p if wRp, where R is the largest simulation.

Lemma 3.12 (Simulation) *The following are equivalent.*

(1) x simulates y.

(2) There exist paths $p \prec x$ in $\mathcal{S}_\Psi$ and $q \prec y$ in $\mathcal{C}_\Psi$ such that p simulates q.

Proof Let R be the largest simulation between $\mathcal{S}_\Psi$ and $\mathcal{C}_\Psi$.

(2) $\Longrightarrow$ (1): Let S be the largest subset of $\mathcal{S}_\Psi \times \mathcal{C}_\Psi$ such that if xSy then there exist paths $p \prec x$ in $\mathcal{S}_\Psi$ and $q \prec y$ in $\mathcal{C}_\Psi$ such that p simulates q. It suffices to show that S is a simulation between $\mathcal{S}_\Psi \times \mathcal{C}_\Psi$. Suppose xSy. Then by the definition of S, there are paths $p \prec x$ in $\mathcal{S}_\Psi$ and $q \prec y$ in $\mathcal{C}_\Psi$ such that p simulates q. As p simulates q and both p and q are paths, we get $p = ((f, c, g), \{(\gamma, p')\})$ and $q = ((c, g), \{(\gamma, p')\})$ such that p' simulates q' for some f, c, g, γ, p', q'. Furthermore, as $p \prec x$, we get $x = ((f, c, g), b)$ for some b such that $p' \prec b(\gamma)$. Similarly, as $q \prec y$, we get $y = ((c, g), b')$ for some b' such that $q' \prec b'(\gamma)$. As p' simulates q' we get $b(\gamma) S b'(\gamma)$. Therefore, S is a simulation between $\mathcal{S}_\Psi \times \mathcal{C}_\Psi$.

(1) $\Longrightarrow$ (2) : For the converse suppose (1) is true. By the definition of xRy, there exists a sequence x_i, y_i ($i \geq 0$), where $x_0 = x$ and $y_0 = y$, such that

- $x_i R y_i$.
- $x_i = ((f_i, c_i, g_i), b_i)$.
- $b_i(\gamma_i) = x_{i+1}$.
- $y_i = ((c_i, g_i), d_i)$.
- $d_i(\gamma_i) = y_{i+1}$.

From this we get the following system of equations:

$$u_i = ((f_i, c_i, g_i), \{(\gamma_i, u_{i+1})\})$$

$$v_i = ((c_i, g_i), \{(\gamma_i, v_{i+1})\}).$$

By the solution lemma, we can let p and q be the solutions for u_0 and v_0 respectively. By definition of $\prec$ and R, it is easy to see that $p \prec x$, $q \prec y$, and pRq. $\square$

Let π be a projection such that $\pi((x, y, z)) = (y, z)$.

Definition 3.41 Given a path p of a computation tree in $\mathcal{C}_\Psi$, let c_p be the conjunction of constraint components of states on the path. The path p is *is canonical* if every disjunct component of c is canonical. A computation tree *is canonical* if all paths of the tree are canonical. □

Theorem 3.16 (Soundness) *For any canonical computation tree $y \in \mathcal{C}_\Psi$ there is a solution tree $x \in \mathcal{S}_\Psi$ such that x simulates y.*

Proof Let q be any path of a computation tree $y \in \mathcal{C}_\Psi$. We apply the canonical compactness theorem (Theorem 3.10) to select a global solution f to the path q. Make a family of solutions by restricting f to each step on the path of y. Write a system of equations

$$y_i = ((c_i, g_i), z_i)$$
$$z_i = \{(\gamma_i, y_{i+1})\}$$

for $i \geq 0$ such that $y_0 = q$, i.e., q is the solution to y_0. Then apply the solution lemma to the system of equations

$$x_i = ((f_i, c_i, g_i), w_i)$$
$$w_i = \{(\gamma_i, x_{i+1})\}$$

where f_i is the above-mentioned restriction of f to $\mathcal{V}(c_i) \cup \mathcal{V}(g_i)$ for $i \geq 0$. Let p be the solution to x_0. It is clear that $\pi^*(p) = q$. By Lemma 3.10, there is a solution tree x such that p is a path of x. Hence, by Lemma 3.12, x simulates y. □

Theorem 3.17 (Completeness) *For any solution tree $x \in \mathcal{S}_\Psi$ there is a computation tree $y \in \mathcal{C}_\Psi$ such that x simulates y.*

Proof Let R be the largest simulation. By definition of $\prec$, $xR\pi^*(x)$ is true. By Lemma 3.10 there exists a computation tree y in $\mathcal{C}_\Psi$ such that $\pi^*(x) \prec y$. Therefore, we get xRy. □

We turn to the soundness and completeness of the negation-as-failure rule (NAFR). Note that soundness and completeness are almost obvious because of the maximum semantics, provided only canonical programs are considered. An important point is that the formulation of NAFR needs ground goals, which may be infinite or irrational in the language L_{A_ω}, whereas actual computations require that constraints and goals in computation states be *finitary*. Thus a nontrivial aspect of the following Theorem 3.18 is that NAFR is given a meaning over a domain of infinite and irrational objects such as hypersets through implicit and finitary representations as approximations for those objects.

Theorem 3.18 (Negation As Failure) *Let g be a goal with parameters $x_1, \cdots, x_n$ that has the canonical computation tree $q = ((\emptyset, g), b)$ for some b. Let f be an assignment such that $f(x_i) = \xi_i \in V_{A_\omega}$ is not a solution of g. Then there is a finitary constraint c such that f is a solution of c and there is no computation tree for the goal $c \wedge g$.*

Proof For any path $p \prec q$ such that

$$\begin{aligned}
q_0 &= p = ((c_0, g_0), \{(\gamma_0, q_1)\}) \\
q_1 &= ((c_1, g_1), \{(\gamma_1, q_2)\}) \\
q_2 &= ((c_2, g_2), \{(\gamma_2, q_3)\}) \\
&\vdots
\end{aligned}$$

where $c_0 = \emptyset$, $g_0 = g$, and $\gamma_0 \in dom(b)$, there is some q_j in the path such that no extension of f satisfies c_j, for otherwise f could be extended to the solution of q. As every computation tree is hereditarily finite, by applying the standard argument of König's, lemma the set d of such constraints c_j must be finite:

$$d = \{c_1, \cdots, c_r\}$$

for some integer r. Then by the solution compactness theorem (Theorem 3.12) there exists a positive finite 'covering' constraint c of $(\xi_1, \cdots, \xi_n)$ such that c has no common solution with c_j above. It follows from the construction of the constraint c in L_{A_ω} that there is no computation tree for the goal $c \wedge g$. Otherwise, from the canonical computation tree q_0, we could also get a canonical computation tree that has the following path.

$$\begin{array}{rcl} q'_0 & = & ((c \wedge c_0, g_0), \{(\gamma_0, q'_1)\}) \\ q'_1 & = & ((c \wedge c_1, g_1), \{(\gamma_1, q'_2)\}) \\ q'_2 & = & ((c \wedge c_2, g_2), \{(\gamma_2, q'_3)\}) \\ & \vdots & \end{array}$$

By the canonical support theorem (Theorem 3.11), this implies that c and $\bigwedge\{c_i \mid i \geq 0\}$ have a common solution, which is also a solution of g. This is a contradiction.

□

3.5 Applications to Terms and Records

In this section, we make several more specific remarks about implementation issues. First of all, we point out that a partition refinement algorithm in Paige and Tarjan [69] can be used to compute bisimulation relations over hypersets. Secondly we take the domains of trees and records as two special subclasses of V_{A_ω}, respectively, by making the following embedding of standard constraint theories into L_{A_ω} in a natural way.

- Unification theory over (infinite) trees.
- Unification theory over (infinite) records.

3.5.1 Computing Bisimulations over Hypersets

Recall the definition of $V_{A_\omega}[X]$ for a class X of parameters and a class A of atoms: $V_{A_\omega}[X]$ is the largest class M such that $M \subseteq A \cup X \cup pow'(M)$, where $pow'(M)$ is the set of finite subsets of M. Let U be a transitive set of $V_{A_\omega}[X]$, i.e. $U \in V_{A_\omega}[X]$ and for all $x \in U$ if $y \in x$ then $y \in U$. We modify the notions of partition and bisimulation slightly for our purpose: a binary relation P on U is a *partition* of U if P is an equivalence relation on U such that the following hold:

- If aPx and $a \in U \cap A$ then $x = a$.
- For any parameter $x \in U$ there is a *set* $y \in U$ such that xPy.

A *bisimulation* R is a partition of U such that, for any sets $x, y \in U$, if xRy then the following hold:

- For any $u \in x$ there exists $v \in y$ such that uRv.
- For any $v \in y$ there exists $u \in x$ such that uRv.

Then the following two propositions are easy consequences of Paige and Tarjan [69]:

Proposition 3.19 *For a given partition P of U, there exists a coarsest bisimulation R that is a refinement of P.*

Let n, m be the numbers of nodes and edges, respectively, appearing in U, provided that U is represented suitably as a directed graph.

Proposition 3.20 *There exists an $O(m \log n)$ time-complexity algorithm to compute the coarsest bisimulation for a given partition P of U.*

3.5.2 Hyperterms

Let Σ and Δ be sets of *function symbols* and *argument places*, respectively. σ_0 is a distinguished function symbol in Σ. We assume that $N \subseteq \Delta$, where N is the set of positive integers. Each symbol $\sigma \in \Sigma$ is supposed to be assigned a *finite* subset $arg(\sigma)$ of Δ. A function symbol σ such that $arg(\sigma) = \emptyset$ is taken as a constant symbol as usual. A function b is a *partial assignment for* a symbol $\sigma \in \Sigma$ if $dom(b) \subseteq arg(\sigma)$. A function b is a *full assignment for* a symbol $\sigma \in \Sigma$ if $dom(b) = arg(\sigma)$.

Definition 3.42 $\mathcal{H}^*(\Sigma, \Delta, X)$ is the largest set $M \subseteq V_{(\Sigma\cup\Delta)\omega}[X]$ such that $M \subseteq X \cup \bigcup\{A_\sigma \mid \sigma \in \Sigma\}$, where $A_\sigma = \{(\sigma, b) \mid b\colon arg(\sigma) \to M \quad \text{(partial)}\}$ □

Elements of $\mathcal{H}^*(\Sigma, \Delta, X)$ are called *hyperterms over* (Σ, Δ, X). A hyperterm u is called a *subterm* of a hyperterm v if u appears in v. A *fully specified hyperterm* is a hyperterm u such that for every subterm (σ, b) of u, b is a *full* assignment for the symbol σ.

Definition 3.43 $\mathcal{H}(\Sigma, \Delta, X)$ is the smallest set $M \subseteq \mathcal{H}^*(\Sigma, \Delta, X)$ such that $X \cup \bigcup\{A_\sigma \mid \sigma \in \Sigma\} \subseteq M$, where $A_\sigma = \{(\sigma, b) \mid b\colon arg(\sigma) \to M \quad \text{(partial)}\}$ □

It is clear that every element of $\mathcal{H}(\Sigma, \Delta, X)$ is finitary and that $\mathcal{H}(\Sigma, \Delta, X)$ is the set of finitary hyperterms in $\mathcal{H}^*(\Sigma, \Delta, X)$.

We write $\sigma(b(1), b(2), \cdots, b(n))$ for hyperterms (σ, b) if $dom(b) = arg(\sigma) = \{1, \cdots, n\}$. Also, we write record structures $\{(a_1, b(a_1)), \cdots, (a_n, b(a_n))\}$ for hyperterms $(\sigma_0; b)$ if $\sigma_0 \in \Sigma$ is the distinguished function symbol and $dom(b) = \{a_1, \cdots, a_n\} \subset arg(\sigma_0)$. Thus we can see that the usual first-order terms and record structures are special cases of hyperterms.

3.5.3 Hyperterm Subsumption

Let Σ, Δ, X be the same as above.

Definition 3.44 A binary relation $\sqsubseteq$ on $\mathcal{H}^*(\Sigma, \Delta, \emptyset)$ is defined to be the largest relation $\sqsubseteq$ such that $x \sqsubseteq y \Longrightarrow x \sqsubseteq^+ y$ where $x \sqsubseteq^+ y$ iff there are some $\sigma \in \Sigma$ and partial assignments u, v for σ such that

- $x = (\sigma, u)$ and $y = (\sigma, v)$.
- $dom(u) \subseteq dom(v)$.
- For all $z \in dom(u)$ $u(z) \sqsubseteq v(z)$.

□

It is clear that the domain of hyperterms is ordered by $\sqsubseteq$. The relation $\sqsubseteq$ is called *hyperterm subsumption*. Recall that the set subsumption relation as a hereditary subset relation was not a partial order, but only a preorder on V_A. A decision procedure for hyperterm subsumption is given in Mukai [60].

3.5.4 Unification over Hyperterms

A *hyperequation* is an expression of the form $u \bowtie v$, where u and v are parametric hyperterms. A set of hyperequations is called a *solved form* if every hyperequation in the set has a parameter on the left side and no two equations have the same parameter on the left hand side. A hyperequation $u \bowtie v$ is a *conflict* if $u = (\sigma, b)$ and $v = (\sigma', b')$ for distinct function symbols σ and σ'.

The following algorithm is an extension of the standard unification to records. The input of the unification algorithm is a finite set of hyperequations. The output is a set of hyperequations. The unification algorithm proceeds as follows: Repeat applying the steps below to the input set of hyperequations until no step is applicable. When this process terminates, check whether there is a *conflict* or not. In what follows, x, y are parameters, $\sigma \in \Sigma$, and u, v,w are partial assignments for σ.

(1) If $x \bowtie x$ is in the set, remove it.

(2) If $x \bowtie y$ is in the set, replace all occurrences of y with x.

(3) If $u \bowtie x$ is in the set for a non parameter u, replace it with $x \bowtie u$.

(4) If $x \bowtie (\sigma, u)$ and $x \bowtie (\sigma, v)$ are in the set, remove it and add to the set all hyperequations in the set

$$\{x \bowtie (\sigma, w)\} \cup \{w(a) \bowtie u(a) \mid a \in dom(u)\} \cup \{w(a) \bowtie v(a) \mid a \in dom(v)\}$$

where w is a *parametric* (partial) assignment satisfying the following:

(4.1) $dom(w) = dom(u) \cup dom(v)$.

(4.2) $w(a) \in X$ are distinct new parameters for $a \in dom(w)$.

Definition 3.45 A set E of hyperequations is *unifiable* if the unification process for E terminates with no conflict. □

Definition 3.46 A hyperequation $u \bowtie v$ is *solvable* (in $H^*(\Sigma, \Delta, \emptyset)$) if the *equation* $\theta(u) = \theta(v)$ is solvable, where θ is the translation satisfying the following:

(1) $\theta(x) = x$ if x is a parameter.

(2) $\theta((\sigma, b)) = (\sigma, b')$, where $dom(b') = arg(\sigma)$, $b'(a) = \theta(b(a))$ for $a \in dom(b)$ and $b'(a)$ are new parameters for $a \in dom(b') \setminus dom(b)$.

□

A set S of hyperequations is *solvable* if the set $\{\theta(u) = \theta(v) \mid u \bowtie v \in S\}$ of equations is solvable.

The following two propositions can be proved without much difficulty.

Proposition 3.21 *The unification algorithm over hyperterms always terminates.*

Proposition 3.22 *For a set E of hyperequations, the following are equivalent:*

(1) E is unifiable.

(2) E is solvable.

3.6 Bisimulations on Generalized Terms

This section is independent to the other ones. In this section, two kinds of class-functors called *form-based* and *conservative*, are introduced. For the set-based and conservative functors T, three theorems so called *unification theorem*, *compactness theorem*, and *independence theorem* which concern bisimulation constraints on the final coalgebra for T are proved. Moreover, for the set-based and form-based functors T, a *constraint definability theorem* is proved for the final coalgebra for T.

These results include Colmerauer's independence theorem on the infinite tree unification with unequations as a special case with T being the standard term-forming functor. From the constraint definability and compactness on the final coalgebras, the final coalgebras for the functors can be domains of structured objects with evident restrictions for the constraint logic programming scheme proposed by Jaffar and Lassez.

Basic tools of this work is the final coalgebra theorem proved by Aczel [1] and and Aczel and Mendler [5]. Barwise's unification theorem on bisimulations and subsumptions are used. The restriction of the unification theorem to the bisimulations will be generalized in this section.

The objective of this section is to generalize the infinite tree unification in Colmerauer [23] to the bisimulation constraints on the final coalgebras for some class of endo-functors on the category of classes. We assume that the reader is familiar with an elementary part of both set theory and category theory [45] including the notion of a category, functor, (final) coalgebra for a functor, and mediating arrow. A distinction between class and set is important in this section, though not in a foundational sense but a concise expository reason. Barwise and Etchemendy [16] gives an excellent introduction to Aczel's hyperset theory. Also does Barwise and Moss [17].

Basic tools of this section is the final coalgebra theorem [1, 5] and the unification theorem in Barwise [14]. This work is a generalization of some part of Mukai [59], which applies the solution lemma to the constraint logic programming scheme. Barwise and Etchemendy [16] also gives an introductory description to the final coalgebra theorem as a part of introduction of the hyperset theory. Aczel [1] and Aczel and Mendler [5] proved a final coalgebra theorem that asserts that for every set-based endo-functor T on $\mathcal{C}$, there is a final coalgebra for T, where $\mathcal{C}$ is the superlarge category of classes of hypersets. An ordered pair (X, α) of a class X and a function $\alpha\colon X \to T(X)$ is called a *coalgebra* for T. For example, (X_0, α_0) is a coalgebra for *pow*, where $X_0 = \{x, y\}$, and $\alpha_0(x) = \{x, y\}$, $\alpha_0(y) = \{x\}$. The final coalgebra theorem says that there is a unique *mediating arrow* π such that $\pi(x) = \hat{\pi}(\{x, y\}) = \{\pi(x), \pi(y)\}$, and $\pi(y) = \hat{\pi}(\{x\}) = \{\pi(x)\}$, where $\hat{\pi} = pow(\pi)$ with *pow* as a functor.

$$\begin{tikzcd}
X \arrow[r, "\pi"] \arrow[d, "\psi"] & J(T) \arrow[d, equal] \\
T(X) \arrow[r, "T(\pi)"'] & T(J(T))
\end{tikzcd}$$

Several universes of familiar kinds of structured objects such as infinite and rational trees [23], finite automata, feature (=record) structures are constructed in a uniform way by specifying corresponding functors T. A well-founded universe and non-well-founded universe correspond to a minimum fixpoint $I(T)$ and maximum fixpoint $J(T)$ of the same functor T, respectively.

We treats the following end-functors on $\mathcal{C}$, *pow*, *pow*$'$, H^A, Σ^A, *map*$(A, -)$, *map*$'(A, -)$ with A being a signature, as main examples of set-based form-based, and conservative ones.

- $H^A(M)$ is the class of terms t such that prime function symbols of t is in A and arguments of t are in M.
- *pow*$'(M)$ is the class of finite sets of M.
- *map*(D, D') is the set of total functions from D into D'.
- *map*$'(D, D')$ is the set of partial functions from D into D'.
- $\Sigma^A(M)$ means the set of possible transitions from states in M with actions in A, and formally is the set $\{(\sigma, \delta) \mid \sigma \subseteq \{\lambda\}, \delta \in map'(A, M)\}$.

H^A is used as a Herbrand universe building functor. For example, suppose $A = \{f, g\}$ is a signature where f and g are unary operation symbols. H^A is set-based, form-based and conservative. $H^A(M) = \{f(m) \mid m \in M\} \cup \{g(m) \mid m \in M\}$. Then $I(H^A) = \emptyset$ but $J(H^A)$ is the set of infinite sequences of f and g. This functor H^A has similar properties to the power class functor *pow*. In fact we derive Colmerauer's infinite tree unification from general properties of *pow* in the form of set-based, form-based and conservative functors. The unary functor

$map'(A, -)$ is used as a common building functor for domains of records, finite automata, rational trees, with a slight modification for each class of domains. As a (deterministic) automata is simply a coalgebra for $map'(A, -)$, from this view point based on the final coalgebra theorem, a relationship is given between automata, rational trees, and regular languages.

Constraints are formalized as binary relations on coalgebras of these class functors. In particular, the notion of solutions in final coalgebras are defined as the mediating arrow from the coalgebra to the final one assured by the Aczel's final coalgebra theorem. More precisely, constraints are $\langle p, X\rangle$, where p is a binary relation on V and X is a collection of urelements of V. Elements of X behaves as parameters in bisimulation theory. Bisimulation is a constraint which means a kind congruence relation between parameterized terms. An general idea of the constraint theory proposed goes like this: for a given constraint c, first find a bisimulation closure c' of c, and then find a coalgebra $\langle X, \alpha\rangle$ for the given functor T in the closure c'. The final coalgebra theorem assures that the mediating arrow to the final coalgebra for T is a solution of c. Here is a simplified example: let $T = pow$ and $c = \langle p, X\rangle = \langle\{(x, \{x\})\}, \{x\}\rangle$. Then

$$c' = \langle\{(x, x), (\{x\}, \{x\}), (x, \{x\}), (\{x\}, x)\}, \{x\}\rangle$$

is a bisimulation closure of c, and $T(X) = pow(\{x\}) = \{\emptyset, \{x\}\}$. Let $\alpha: X \to T(X)$ so that $\alpha(x) = \{x\}$. Note that $\{(x, \alpha(x))\} \subseteq p$. So we get a coalgebra $\langle X, \alpha\rangle$. In fact, by the solution lemma, the solution of $x = \{x\}$ exists in $J(pow)$.

Barwise [14] gives a unification theorem for mixed constraints consisting of bisimulations and subsumptions on the class V of hypersets with urelements. His proof is based on Aczel's Solution Lemma [1]. V is the final coalgebra for pow the class functor and Solution Lemma is a special case of the final coalgebra theorem, though the former is a foundation of the whole hyperset theory. Mukai [62]— also see Section 3.6.6—treats external subsumption problems on feature structures by introducing merge operations on final coalgebras for record forming functors. Thus the infinite tree unification, feature unification, Barwise's hyperset unification theory are integrated as bisimulations on the final coalgebras for set-based, conservative, and form-based functors.

Using two simplified examples, we make a review on necessary aspects of the infinite tree unification for this section.

Example 3.6

$$\begin{aligned} x &= f(y, z) \\ y &= f(x, z) \\ x &\neq y \end{aligned}$$

This constraint is *unsolvable* in the domain of infinite trees. To see this, give arbitrary tree in the domain as a value to the unbound parameter z. Then as in general a solved form or system of equations without an unbound parameter has a unique solution, x and y are determined uniquely by z. Unfolding the right hand side of each equation unboundedly, we observe that x and y have the same value for any fixed value to z. Namely, any solution of the first two equations can not satisfy the unequation $x \neq y$. In other words, no solution of the equations satisfies the unequation. Hence there is no solution of the whole constraint. Notice that $\{x = f(y, z), y = f(x, z)\}$ and $\{x = f(y, z), y = f(x, z), x = y\}$ have the same singleton $\{z\}$ of unbound parameters, and moreover the same vacuous constraint i.e., $\{z = z\}$, though it is not explicit. In fact, solvability of constraints is determined by behaviors of the set of unbound parameters and equations between them. In this case, $\{x = f(y, z), y = f(x, z)\}$ and $\{x = f(y, z), y = f(x, z), x = y\}$ have the same constraint for the unbound parameter. □

Example 3.7

$$x = f(y, z)$$
$$y = f(z, x)$$
$$x \neq y$$

On the contrary to the previous example (Example 3.6), this constraint is solvable. To see this, give any atom a as a value to the unbound parameter z. Then evidently x and y determined by $x = f(y, a)$, $y = f(a, x)$ must not be the same, for otherwise $y = a$ and $a = x$ are derived from equational axioms, and hence $f(y, a) = a$ and $a = f(a, x)$, which are conflicts. So the given constraint is satisfiable (=solvable). Observe that in this case the constraint on the unbound parameters in $\{x = f(y, z), y = f(z, x)\}$ is $\{z = z\}$, which is the same as in the previous example. But there is no unbound parameters in $\{x = f(y, z), y = f(z, x), x = y\}$. In fact, as it follows from the simple equational theory that $y = z$, the unbound parameter z in the two equations now becomes bound in the whole constraint. So there is certainly a difference between the constraints on the unbound parameters in $\{x = f(y, z), y = f(z, x)\}$ and $\{x = f(y, z), y = f(z, x), x = y\}$. In fact, a change of state of the constraint on unbound parameters means that the constraint is solvable. On the other hand, no change of the state means that the constraint is unsolvable. □

Now we make a list of important properties used in Colmerauer's infinite tree unification theory [23].

(1) Every finite set of equations is decidable.

(2) Let p is a set of equations. Then $p \cup \{u \neq v\}$ is unsolvable iff $p \cup \{u = v\}$ has a consistent closure set of equations that has exactly the same constraint with p on unbound parameters. See Example 3.6 and 3.7 above.

(3) Let p be a consistent closed set of equations under the equational rules and q be a set of unequations. Then $p \cup q$ is solvable iff $p \cup \{l\}$ is solvable for all $l \in q$. This is called Independence Theorem [23].

(4) A countable set of equations are consistent iff so is every subset of the given set. This property is called *compactness*.

(5) A countable set of equations are solvable iff so is every subset of the given set. This property is called *solution compact*.

(6) The domain is constraint definable. Namely, every infinite tree is represented as a unique solution of a system of equations.

Of course, Colmerauer universe $J(H^A)$ have these properties. However, the properties are possessed by not only the infinite trees but also hypersets (non-well-founded sets) [59]. In fact, the properties in the list are derived from the simple fact that the functor H^A is set-based, conservative and form-based. Evidently, the power class building functor *pow* is set-based, conservative and form-based. Thus we have a new family of CLP($J(T)$) indexed by the set-based, form-based and conservative functors T. This result is a refinement of the CLP scheme.

We give an informal introduction of form-based functor, conservative functor, and component function. First, in the standard first-order unification, we have never $x = f$ from, for example,

$f(x, y) = f(y, z)$, where the prime functor f is not counted as a subterm. Derived equations such as $x = y$ are always ones between subterms appearing in the given equations. In general, elements of $I(T_X)$ are called T-terms. A conservative functor is introduced so that given a constraint on T-terms, every parameter is bound to a T-term in any bisimulation closure of the constraint. H^A and *pow*, for example, are conservative.

Given a first-order term, $f(g(x), y)$, for example, the arguments of $f(g(x), y)$ are $g(x)$ and y. The argument of $g(x)$ is x. The subterms of $f(g(x), y)$ are $f(g(x), y)$, $g(x)$, x, and y, which are obtained by applying the taking arguments in a transitive way. Note that all parameters appear as subterms. Moreover, from an equation $f(x) = f(a)$, we get $x = a$. In more general, from a given equation $p = q$, we get equations between arguments of p and q. A component function for T is, roughly speaking, a generalization of taking arguments of first-order terms to the T-terms. We will show that the existence of a component function implies conservativeness. Thus unification theorem for the power class functor is generalized for the set-based and conservative functors, and we can verify that useful functors T such as H^A are conservative by showing a component function for T.

Given a first-order term, $f(g(a), b)$, for example, we can represent this term with two equations $z = f(u, b)$, $u = g(a)$. Note that $f(u, b)$ and $g(b)$ are *flat* and also $f(g(a), b)$ is a unique solution of the two equations for u. Even if the given term is an infinite tree, we can represent it uniquely by possibly infinite number of flat equations. This being representable is a property of H^A. We abstract it as a form-based functor T. More generally, for any element $x \in T(M)$, there is a parametric set u as a *form* with parameters such that x is obtained from u by replacing the parameters with corresponding elements of M. Namely, a form-based functors are essentially term forming operations that fill argument places with elements of M. We will show that constraint definability holds in the final coalgebras for set-based and form-based functors.

The section is organized as follows. In section 2, basic definitions and other preliminaries are given. Among them are hyperset (Aczel [1]), bisimulation relation, term, and infinite tree. In section 3, the notion of constraint is defined. In section 4, conservative and form-based functors are introduced and some basic properties are given. Component function is introduced to give a sufficient condition for a set based functor to be conservative. It is proved that form-based functor implies constraint definability. An application to finite state automata is given. In section 5, the unification theorem is proved. In section 6, negative constraints are treated, and the independence theorem and solution compactness of the constraint for set-based and conservative functors is proved.

3.6.1 Backgrounds

We assume a universe V of hypersets with urelements for our metatheory. Elements of V are called *terms*. Δ denotes the class of urelements in V: $\Delta \subsetneq V$. We will need sufficiently many urelements. So we assume that Δ is a proper class. Urelements will be called *atoms* or *parameters* depending on the context. Elements of V that are not urelements are called *sets*. The emptyset $\emptyset$ is a set but not an urelement. If x is an urelement then there is no $y \in V$ such that $y \in x$. A *class functor* is an endo-functor on the superlarge category $\mathcal{C}$ of classes on V. Let T be a class functor. Given a class $C \subseteq V$, we use the notation T_C for the class functor defined by the following.

- $T_C(M) = C \cup T(M)$ for $M \subseteq V$.

- For $f: M \to M'$, $T_C(f)(x) = \begin{cases} T(f)(x) & \text{if } x \in T(M) \\ x & \text{if } x \in C \setminus T(M). \end{cases}$

The notation T_C is fundamental in this section.

We use the symbol $map(P, Q)$ for the set of *total* functions from a set P into a set Q and $map'(P, Q)$ for the set of *partial* functions from P into Q. A class $A \subseteq \Delta$ is called a *signature* if each $a \in A$ is assigned a non-negative integer.

We define functors pow, pow', $fot(A, -)$, $aut(A, -)$, H^A, and Σ^A, which are easily verified to be endo-functors on $\mathcal{C}$, i.e., the category of classes.

Definition 3.47 • K_C is a constant functor such that $K_C(M) = C$ and $K_C(f) = id_C$ for all $M, M' \subseteq V$ and $f: M \to M'$, where C is a given class.

- pow is the power class functor. $pow(f): pow(M) \to pow(M')$, $pow(f)(u) = \{f(x) \mid x \in u\}$ for $u \subseteq M$.
- pow' is the functor that assigns to each class $x \subseteq V$ the class of finite subsets of x. $pow'(f) = pow(f)$ for $f: M \to M'$, $M, M' \subseteq V$.
- $aut(A, X) = \{(q, p) \mid q \subseteq \{\lambda\}, p \in map'(A, X)\}$. $aut(A, f)((q, p)) = (q, p')$, where $p'(a) = f(p(a))$ for $a \in dom(p)$ and $A \subseteq \Delta$.
- $fot(A, X) = \{(\sigma, p) \mid \sigma \in F, p: \{1, \cdots, arity(\sigma)\} \to X\}$. $fot(A, f)((\sigma, p)) = (\sigma, p')$, where $p'(i) = f(p(i))$ for $1 \leq i \leq arity(\sigma)$, and A is a signature.
- $H^A = fot(A, -)$, where A is a signature.
- $\Sigma^A = aut(A, -)$, where $A \subseteq \Delta$.

□

Definition 3.48 ([5]) An endo-functor T on the superlarge category of classes is called *set-based* if for each class A and each $a \in T(A)$ there is a set $A_0 \subseteq A$ and $a_0 \in T(A_0)$ such that $a = T_{\iota_{A_0, A}}(a_0)$, where $\iota_{A_0, A}$ is the inclusion map $A_0 \hookrightarrow A$. □

It is obvious that pow, pow', H^A, Σ^A are set-based. $I(T)$ denotes the minimum fixpoint of T. $J(T)$ denotes the maximum fixpoint of T. If T is set-based then $I(T)$ and $J(T)$ exist [1]. It follows that V is the maximum fixpoint of a functor pow_Δ. On the other hand, the universe of pure sets is the minimum fixpoint of pow.

Given a signature $A \subseteq \Delta$, it can be proved that $J(H^A_X)$ forms a set if A and X are sets. Elements of $I(H^A_X)$ and, in particular, $I(H^A)$ are called *Herbrand terms* and *Herbrand trees*, respectively. It is clear that $I(H^A_X)$ is a proper subclass of $V_{A\omega}[X]$. There is an evident bijection ψ from the the standard first-order term notations onto the finitary Herbrand terms such that $\psi(f(x_1, \ldots, x_n)) = (f, \{(i, \psi(x_i)) \mid 1 \leq i \leq n\})$. In this translation we have identified each atom a of arity 0 with $(a, \emptyset)$.

An *n-ary relation* on a class M is a subclass of M^n. The *carrier* of an n-ary relation r is the class $\{x_i \in V \mid r(x_1, \cdots, x_i, \cdots, x_n), 1 \leq i \leq n\}$. $carri(r)$ denotes the carrier of r. A class t is called *transitive* if $t \subseteq pow(t)$. Given a class $u \in V$, $trans(u)$ denotes the minimum transitive class $t \in V$ such that $u \subseteq t$. $trans(u)$ is called the *transitive closure* of u. For a relation r on V, we define $fld(r) = trans(carri(r))$.

Example 3.8

$$fld(\{(x,a),(y,\{b,c\})\}) = \{x,a,y,\{b,c\},b,c\}$$

where x, y, a, b, $c \in \Delta$. □

$\mathcal{V}_X(s) \stackrel{\text{def}}{=} X \cap trans(\{s\})$, where $s \in V$, $X \subseteq \Delta$. Sets in $J(pow'_\Delta)$ are called *hereditarily finite*. It can be proved that if A is a set then $V_{A\omega}$ forms a set. For any set $x \in I(pow'_\Delta)$, it follows that $trans(x)$ is finite. So we call terms in $I(pow'_\Delta)$ *finitary*. In fact, finitary set terms are hereditarily finite and well-founded.

Example 3.9 Let N be the set of natural numbers. Then the set $\{N\}$ is finite and well-founded, but not finitary since $trans(\{N\})$ is infinite. □

Definition 3.49 Let $X \subseteq \Delta$ be a class A family $(b_x)_{x \in X}$ is called a *system of equations* (for X) if $b_x \in V \setminus X$ for each $x \in X$. □

Theorem 3.23 (Solution Lemma [1]) *Every system of equations for a class X has a unique solution in $J(pow_{\Delta \setminus X})$.*

Theorem 3.24 (Final Coalgebra Theorem [1, 5]) *Every set-based functor has a final coalgebra.*

Note that as a system of equations is a coalgebra for pow_Δ, the solution lemma can be seen as a special case of the final coalgebra theorem.

3.6.2 Bisimulation Constraints

Definition 3.50 A *constraint* is an ordered pair $\langle r, X\rangle$, where r is a binary relation and $X \subseteq \Delta$. A constraint $\langle r, X\rangle$ is called a *constraint for T* if, for all $x \in X \cap carri(p)$, either $r(x,y)$ or $r(y,x)$ for some $y \in I(T_X)$. □

A constraint $\langle r, X\rangle$ is called a *constraint on a class C* if r is a relation on C. By a constraint we mean one on V unless otherwise mentioned explicitly. We often call constraints of the form $\langle r, X\rangle$ *X-constraints*. A constraint $\langle r, X\rangle$ is an *extension* of a constraint $\langle s, Y\rangle$ if $s \subseteq r$, $Y \subseteq X$. An *assignment* is a partial function f from Δ into V such that $dom(f) \cap \mathcal{V}_{dom(f)}(ran(f)) = \emptyset$. Let f be an assignment. Then it follows from the solution lemma that there is a unique function $\hat{f}$ from V to $J(pow_{\Delta \setminus dom(f)})$ such that

(1) $\hat{f}(x) = f(x)$ if $x \in dom(f)$.

(2) $\hat{f}(a) = a$ if $a \in \Delta \setminus dom(f)$.

(3) $\hat{f}(s) = \{\hat{f}(u) \mid u \in s\}$ if s is a set.

A function f is called an *assignment for* a constraint $\langle r, X\rangle$ if $dom(f) = X$ and $ran(f) \subseteq J(pow_{\Delta\setminus X})$, i.e., $\mathcal{V}_X(ran(f)) = \emptyset$. Note that if $X = \emptyset$ then $\emptyset$ is the assignment for X by definition.

Definition 3.51 Let $c = \langle p, X\rangle$ be a constraint and let $D \subseteq V$. An assignment $f\colon X \to D$ is called a *solution of c in D* if $\hat{f}(u) = \hat{f}(v)$ for all $p(u, v)$. The constraint c is called *solvable in D* if there is a solution of c in D. □

Note that if T is a set-based with $J(T) = \emptyset$, then it follows from the definition of a solution that a constraint $\langle p, X\rangle$ on $J(T_X)$ is solvable in $J(T)$ iff $p = X = \emptyset$.

Definition 3.52 Let $X \subseteq \Delta$. A *bisimulation* is a constraint $\langle r, X\rangle$, where r is an equivalence relation such that if $r(u, v)$ then the following hold.

- If $u \in \Delta \setminus X$ and $v \notin X$ then $u = v$.
- If u and v are sets then

$$\forall x \in u \exists y \in v\ r(x, y) \quad \& \quad \forall y \in v \exists x \in u\ r(x, y).$$

□

A bisimulation is called a *bisimulation on a class C* if it is constraint on C.

Definition 3.53 Let $X \subseteq \Delta$. $x \in \Delta$ is *bound* in a constraint $\langle p, X\rangle$ if $x \in X$ and either $p(x, b)$ or $p(b, x)$ for some $b \notin X$. A constraint $\langle p, X\rangle$ is *bound* if each $x \in X$ is bound in $\langle p, X\rangle$. □

Clearly it follows from the solution lemma that a bound constraint $\langle r, X\rangle$ has at most one solution. We say a constraint $\langle s, Y\rangle$ is a *bisimulation of* a constraint $\langle r, X\rangle$ if $\langle s, Y\rangle$ is a bisimulation and an extension of $\langle r, X\rangle$. Also we say a constraint $\langle r, X\rangle$ *has* a bisimulation $\langle s, Y\rangle$ when $\langle s, Y\rangle$ is a bisimulation of $\langle r, X\rangle$. We say that a constraint $\langle s, Y\rangle$ is a *local bisimulation* of a constraint $\langle r, X\rangle$ if $X = Y$ and $\langle s, Y\rangle$ is a bisimulation of $\langle r, X\rangle$ and $s \subseteq \mathit{fld}(r) \times \mathit{fld}(r)$, i.e., only terms appearing in r appear in s. We make similar uses of words for a bisimulation for T. Constraints are called *finitary* if they are finitary as terms.

Proposition 3.25 *For every finitary constraint $c = \langle p, X\rangle$, the following hold.*

(1) The set of local bisimulations of c is finite.

(2) If there exists a bisimulation of c, then there exists also a local bisimulation of c.

Proof Suppose $\langle q, X\rangle$ is a local bisimulation of $\langle p, X\rangle$. Then $q \subseteq \mathit{fld}(p) \times \mathit{fld}(p)$. As $\mathit{fld}(p)$ is finite, q must be finite. Hence the set of such local bisimulations $\langle q, X\rangle$ is finite. So we get (1).

We prove (2). Let $\langle q, X\rangle$ be a bisimulation of $\langle p, X\rangle$. By definition, we get $p \subseteq q$. As $\mathit{fld}(p)$ is finite, $q \cap \mathit{fld}(p) \times \mathit{fld}(p)$ is finite. It is a routine to check that the constraint $\langle q \cap \mathit{fld}(p) \times \mathit{fld}(p), X\rangle$ satisfies all defining clauses of a bisimulation. As $p \subseteq q \cap \mathit{fld}(p) \times \mathit{fld}(p)$, it follows that $q \cap \mathit{fld}(p) \times \mathit{fld}(p)$ is a local bisimulation of $\langle p, X\rangle$. □

Proposition 3.26 *For every finitary constraint c, the existence of a bisimulation of c is decidable.*

Proof In general, for a given finite set B, the existence of a bisimulation p such that $\mathit{fld}(p) \subseteq B$ is decidable by an exhaustive search method.

It is clear that, for all constraints $\langle p, X\rangle$, $\langle q, X\rangle$ is a local bisimulation of $\langle p, X\rangle$ iff $\langle q, X\rangle$ is a bisimulation and $q \subseteq \mathit{fld}(p) \times \mathit{fld}(p)$. Since p is finitary, $\mathit{fld}(p)$ is finite. So, it follows from the above general remark that the existence of a local bisimulation of $\langle p, X\rangle$ is decidable. Hence, by Proposition 3.25, the existence of a bisimulation of $\langle p, X\rangle$ is decidable. □

3.6.3 Conservative and Form-based Functors

We introduce conservative functor and find a condition for a function to be conservative.

Definition 3.54 A class functor T is *conservative* if the following are equivalent for any $X \subseteq \Delta$ and constraint $\langle p, X\rangle$ on $I(T_X)$.

(1) $\langle p, X\rangle$ extends to some bisimulation.

(2) $\langle p, X\rangle$ extends to some bisimulation for T.

□

Example 3.10 *pow*, *pow'*, H^A, and Σ^A are conservative functors. See Proposition 3.27 below. □

Example 3.11 Let $A = f, g$ is a signature and f and g are distinct unary function symbols. Let $F = \mathit{fot}(A, -)$, i,e., $F(M) = \{f(g(x) \mid x \in M\} \cup \{f(x) \mid x \in M\}$. Consider the constraint $c_0 = \langle\{(f(g(x)), f(y))\}, \{x, y\}\rangle$. Clearly, c_0 extends to a bisimulation. Let c be a bisimulation of c_0. Evidently c must relate y to $g(x)$. As $g(x) \notin J(T_{\{x,y\}})$, c is not a constraint for T. Hence F is not conservative. □

Given a class-valued function γ on V, γ^* is the class-valued function on V defined so that for $u \in V$ $\gamma * (u)$ is the least class M such that $\gamma(u) \subseteq M$ and $\gamma(v) \subseteq M$ for all $v \in M$. Clearly, γ^* is well-defined and uniquely exists.

Definition 3.55 Let T be a set-based functor and $X \subseteq \Delta$. A set-valued function on $I(T_X)$ is called a *component function for T* if the following hold.

- If $u \in I(T_X)$ then $\gamma(u) \subseteq I(T_X) \cap \mathit{trans}(\{u\})$.
- If $u \in I(T_X)$ then $\mathcal{V}_X(u) \subseteq \gamma^*(u)$.
- For any bisimulation $\langle p, X\rangle$, if $p(u, v)$ with $u, v \in I(T_X) \setminus \Delta$ then

$$\forall x \in \gamma(u)\ \exists y \in \gamma(v)\ p(x, y)\ \&\ \forall y \in \gamma(v)\ \exists x \in \gamma(u)\ p(x, y).$$

□

Then the following proposition is obvious from the definition of a conservative functor.

Proposition 3.27 *If there is a component function for T, then T is conservative.*

Lemma 3.13 *Let $\langle p, X\rangle$ be a bisimulation. Then if $p((u,v),(u',v'))$ then $p(u,u')$ and $p(v,v')$.*

Proof Use the definition $(x,y) = \{\{x\},\{x,y\}\}$. It is an exercise in set theory. □

Proposition 3.28 *pow, pow', H^A, and Σ^A are conservative.*

Proof By Proposition 3.27, it suffices to show a component function γ_T on $I(T_X)$ to each of the four functors T and each $X \subseteq \Delta$. Define $\gamma_T(x) = \emptyset$ for each $x \in X$. For each $x \in I(T_X)$, define $\gamma_T(x)$ as follows.

(1) $\gamma_{pow}(x) = x$.

(2) $\gamma_{pow'}(x) = x$.

(3) $\gamma_{H^A}(x) = \{p(a) \mid a \in dom(p)\}$, where $x = (\sigma, p)$.

(4) $\gamma_{\Sigma^A}((\sigma,p)) = \{p(a) \mid a \in dom(p)\}$, where $x = (\sigma, p)$.

It is a routine to show that all γ_T above are component functions for T. In particular, Lemma 3.13 is used to prove (3) and (4). □

Corollary 3.29 *A constraint $c = \langle p, q, X\rangle$ on $I(H^A_X)$ is solvable in $J(H^A)$ if c is solvable in some non-empty domain. In particular, c is solvable in $J(H^A)$ iff so is in $J(pow'_A)$.*

This corollary means that the domain of Herbrand trees over A is embedded into the domain of hereditarily finite sets over A.

We introduce the notion of form-based functor so that the final coalgebra for the functor are constraint-definable.

Definition 3.56 A class functor T is called *form-based* if for any $M \subseteq V$ and $u \in T(M)$, $u = \hat{\varphi}(v)$ for some $X \subseteq \Delta$, $v \in T(X)$, $\varphi: X \to M$. □

Although our aim is to generalize the infinite tree unification to the one over final coalgebras, the conservative and form-based functors could be related to theory of structured objects developed in Aczel [3], Aczel [4], Fernando [31], and Lunnon [48]. However, comparisons of these theories are out of place here.

Proposition 3.30 (Constraint Definability) *Let T be a set-based and form-based functor such that there is an injective map $\psi: J(T) \to \Delta \setminus \mathcal{V}_\Delta(J(T))$. Then there is a coalgebra (X, α) for T such that the mediating map $(X,\alpha) \to (J(T), id_{J(T)})$ is surjective.*

Proof Let $X = \{\psi(t) \mid t \in J(T)\}$. $\varphi = \psi^{-1}: X \to J(T)$. φ is a bijection. We construct a coalgebra (X, α) for T such that $\hat{\varphi}(\alpha(x)) = \varphi(x)$ for all $x \in X$.

As T is form-based, for each $t \in J(T)$, there is $X_t \subseteq \Delta$, $v_t \in T(X_t)$, and $\varphi_t: X_t \to J(T)$ such that $\widehat{\varphi_t}(v_t) = t$. We can assume without loss of generality that $X_t \subseteq X$ and φ_t is the restriction of φ to X_t for all $t \in J(T)$. As T is set-based, T is monotone. So $T(X_t) \subseteq T(X)$. Hence (X, α) forms a coalgebra for T. It is clear that φ is the mediating map from the coalgebra (X, α) for T to the final coalgebra $(J(T), id_{J(T)})$ for T. Hence we have the theorem. □

We turn to finite state automata. An *A-automaton* is a coalgebra for the unary functor $aut(A, -)$. In the definition of aut, $\lambda \in \Delta$ is a distinguished urelement for indicating 'accept' states of the automata, and also the empty string of regular language. A set is called *rational* if it has a finite transitive closure. $pow(\Sigma^*)$ is complete w.r.t. $\subseteq$. Given an A-automaton $P: X \to aut(A, X)$, let T_P be a transformation on $map(X, pow(\Sigma^*))$ such that for given $L \in map(X, pow(\Sigma^*))$, $T_P(L)(x) = \lambda_x \cup \bigcup\{aL(p_x(a)) \mid a \in dom(p_x)\}$, where $p(x) = (\lambda_x, p_x)$.

Proposition 3.31 *For a given automaton P, T_P has a least fixpoint.*

Proof Clearly T_P is monotone. Define a family $(L_n)_{n \geq 0}$ of $L_n: X \to pow(\Sigma^*)$ such that $L_0(x) = \emptyset$ for all $x \in X$ and $L_{n+1} = T(L_n)$. Let $L: X \to pow(\Sigma^*)$ so that $L(x) = \bigcup\{L_n(x) \mid n \geq 0\}$.

$$\begin{aligned}
T_P(L)(x) &= \lambda_x \cup \bigcup\{aL(p_x(a)) \mid a \in dom(p_x)\} \\
&= \lambda_x \cup \bigcup\{\bigcup\{aL_n(p_x(a)) \mid n \geq 0\} \mid a \in dom(p_x)\} \\
&= \bigcup\{\lambda_x \cup \bigcup\{aL_n(p_x(a)) \mid a \in dom(p_x)\} \mid n \geq 0\} \\
&= \bigcup\{L_{n+1}(x) \mid n \geq 0\} \\
&\subseteq L(x)
\end{aligned}$$

Therefore L is a fixpoint of T_P. □

Let (X, δ) be a finite coalgebra for $aut(A, -)$ and π is a mediating arrow to the final coalgebra of $aut(A, -)$. Then $\pi(x)$ is a rational trees for all $x \in X$. These results are direct consequences of the final coalgebra theorem.

3.6.4 Unification Theorem

Lemma 3.14 *Every X-bisimulation is solvable in $J(pow_{\Delta \setminus X})$.*

Proof Suppose $\langle p, X\rangle$ be a bisimulation. Let $\langle q, X\rangle$ be the minimum bisimulation extension of $\langle p, X\rangle$ such that if $x \in X$ is unbound in $\langle p, X\rangle$ then $q(x, \emptyset)$. Evidently, the bound bisimulation $\langle q, X\rangle$ exists. Let $(b_x)_{x \in X}$ be a family of terms such that $q(x, b_x)$ and $b_x \notin X$ for each $x \in X$. By the solution lemma, there exists a unique assignment f for X such that $f(x) = \hat{f}(b_x)$ for all $x \in X$. Let $B = \{(\hat{f}(u), \hat{f}(v)) \mid q(u, v)\}$. As $\langle q, X\rangle$ is a bound bisimulation, it follows that $\langle B, \emptyset\rangle$ is a bisimulation. As it follows from Aczel [1] that any ($\emptyset$-)bisimulation on V is an identical relation on V, we get $B \subseteq =$. Hence we get $\hat{f}(u) = \hat{f}(v)$ for all $q(u, v)$. So f is a solution of $\langle q, X\rangle$. As $\langle q, X\rangle$ is an extension of $\langle p, X\rangle$, f is also a solution of $\langle p, X\rangle$. Hence $\langle p, X\rangle$ is solvable in $J(pow_{\Delta \setminus X})$. □

Then, for any given finite constraint c on $I(H_X^A)$, it is decidable whether c is solvable in R_A or not. In other words, the set of solvable constraint c on $I(H_X^A)$ in R_A forms a recursive set.

Definition 3.57 A triple $\langle p, q, X\rangle$ is called a *constraint* on $V' \subseteq V$ if both $\langle p, X\rangle$ and $\langle q, X\rangle$ are constraints on V' in the sense defined above. An assignment f for X is a *solution* of $\langle p, q, X\rangle$ in D if the following hold.

- f is a solution of the constraint $\langle p, X\rangle$ in D in the sense defined above.
- $\hat{f}(u) \neq \hat{f}(v)$ if $q(u, v)$.

□

Definition 3.58 Given a constraint $\langle p, X\rangle$, $\mathcal{X}(\langle p, X\rangle)$ is the set of ordered pairs $(x, y) \in p \cap X \times X$ such that neither x nor y is bound in $\langle p, X\rangle$. □

Example 3.12 $\mathcal{X}(\langle\{(x, y), (y, z), (x, \{x\})\}, \{x, y, z\}\rangle) = \{(y, z)\}$. □

The unification theorem is a generalization of a restricted version of Barwise's unification theorem [14] to the bisimulation constraints.

Theorem 3.32 (Unification Theorem) *Let T be a conservative and set-based functor and $X \subseteq \Delta$ such that $X \cap trans(J(T)) = \emptyset$. Then for every constraint $\langle p, X\rangle$ on $I(T_X)$ with $\mathcal{X}(p) \neq p$ the following are equivalent.*

(1) $\langle p, X\rangle$ has a solution in $J(T)$.

(2) $\langle p, X\rangle$ extends to some bisimulation.

Proof (1)$\Longrightarrow$(2): Suppose (1) is true. Let f be a solution of $\langle p, X\rangle$ in $J(T)$. Define $r = \{(u, v) \mid u, v \in \mathit{fld}(p), \hat{f}(u) = \hat{f}(v)\}$. It is a routine to check that $\langle r, X\rangle$ is a bisimulation. So, as T is conservative, $\langle p, X\rangle$ has a bisimulation for T.

(2)$\Longrightarrow$(1): Suppose (2) is true. As $\langle p, X\rangle$ extends to some bisimulation and T is conservative, $\langle p, X\rangle$ extends to some bisimulation $\langle q, Y\rangle$ for T. Choose any $a_0 \in \mathit{carri}(\mathcal{X}(p)) \setminus X$ and q' be the reflexive and symmetric closure of $\{(x, a_0) \mid x \in \mathit{carri}\,\mathcal{X}(p)\}$. Clearly, $c = \langle q \cup q', Y\rangle$ is a bound bisimulation for T. By Lemma 3.14, the bound bisimulation c has a unique solution in V. As c is bound, we can choose a coalgebra (Y, α) for T such that $(y, \alpha(y)) \in q \cup q'$ and $b_y \notin Y$ for all $y \in Y$. By the final coalgebra, the system of equations (Y, α) has a unique solution f' in $J(T)$. By the uniqueness, it follows that $f = f'$. Hence f restricted to X is a solution of $\langle p, X\rangle$ in $J(T)$. □

Remark When $J(T) = \emptyset$, a constraint $\langle p, X\rangle$ on $I(T_X) = X$ has a solution in $J(T)$ iff $X = p = \emptyset$. So, without the condition $\mathcal{X}(p) \neq p$, Theorem 3.32 does not hold in general for a functor T such that $J(T) = \emptyset$. □

Given a functor T, a T *unification problem* is to decide whether given X-constraints on $I(T_\Delta)$ are solvable in $J(T_{\Delta \setminus X})$.

Theorem 3.33 *If T is a set-based and conservative functor, then the finitary T-unification problem is decidable.*

Proof Suppose a finitary constraint $\langle p, X\rangle$ on $I(T_X)$ is given with $I(T) \cap X = \emptyset$. If $\langle q, X\rangle$ is a bisimulation of $\langle p, X\rangle$ on $I(T_X)$, then $\langle q \cap \mathit{fld}(p) \times \mathit{fld}(p), X\rangle$ is a bisimulation of p on $\mathit{fld}(p)$. So, if X and $\mathit{fld}(p)$ are finite, we can enumerate all X-bisimulation extensions of p on $\mathit{fld}(p)$. Moreover, as T is conservative, it follows that $\langle p, X\rangle$ is solvable in $J(T)$ iff p extends to a X-bisimulation on $\mathit{fld}(p)$. Hence, as $\mathit{fld}(p)$ is finite, the T-unification problem $\langle p, X\rangle$ is decidable. □

By this theorem (Theorem 3.33), in particular, as H^A is conservative and set-based, the H^A-unification problem is decidable. In fact this is the case of infinite tree unification in Colmerauer [23] without unequations. The full version of the infinite tree unification with unequations will be treated in a section below.

3.6.5 Negative Constraints

We prove Independence Theorem for the set-based, form-based, and conservative functors. Recall that we have identified $J(H^A)$ with the domain of infinite trees in Colmerauer [23] over A. A constraint over infinite trees is a binary relation on $I(H^A_X)$ that has an at most countable carrier.

Let R_A be the set of rational sets in $V_{A\omega}$.

Proposition 3.34 *Let T be a conservative and set-based functor, $X \subseteq \Delta$ be finite with $J(T) \cap X = \emptyset$, and $\langle p, X\rangle$ be a bisimulation for T. If $(\langle p_i, X_i\rangle)_{i\in\Lambda}$ is a family of non-empty constraints on $I(T_X)$ such that $X_i \subseteq X$ for all $i \in \Lambda$. Then the following are equivalent.*

(1) Every solution of $\langle p, X\rangle$ in $J(T)$ satisfies $\langle p_i, X\rangle$ in $J(T)$ for some $i \in \Lambda$.

(2) There is some $i \in \Lambda$ such that every solution of $\langle p, X\rangle$ in $J(T)$ satisfies $\langle p_i, X\rangle$.

(3) There exists a bisimulation $\langle q, X\rangle$ for T of $\langle p \cup p_i, X\rangle$ with $\mathcal{X}(q) = \mathcal{X}(p)$ for some $i \in \Lambda$.

Proof (2)$\Longrightarrow$(1): Obvious.

(3)$\Longrightarrow$(2): Suppose (3) is true. By the hypothesis, there exist some $i \in \Lambda$ and a bisimulation $\langle q, X\rangle$ on $I(T_X)$ of $\langle p \cup p_i, X\rangle$ such that $\mathcal{X}(\langle q, X\rangle) = \mathcal{X}(\langle p, X\rangle)$. Let f be any solution of $\langle p, X\rangle$. As $\mathcal{X}(\langle q, X\rangle) = \mathcal{X}(\langle p, X\rangle)$, it follows that f is also a solution of $\langle q, X\rangle$. Hence as $p_i \subseteq q$, f is a solution of $\langle p_i, X\rangle$. So we get (2).

(1) $\Longrightarrow$ (3): Suppose (1) is true. Note that it is clear that, in general, if $c = \langle s, Z\rangle$ is a bisimulation on $I(T_Z)$ then $\mathcal{X}(c)$ is an equivalence relation on Z: $\mathit{carri}(\mathcal{X}(c)) \subseteq Z$. We prove this case by induction on the number of equivalence classes of $\mathcal{X}(\langle p, X\rangle)$. Firstly suppose that $\mathcal{X}(\langle p, X\rangle) = \emptyset$. Suppose that f is any solution of $\langle p, X\rangle$ in $J(T)$. Then, from the hypothesis (1), there must be some $i \in \Lambda$ such that f is a solution of $\langle p_i, X\rangle$ in $J(T)$. Then $\langle p \cup p_i, X\rangle$ has a solution in $J(T)$. Let $q = \{(u, v) \mid u, v \in \mathit{fld}(p) \cup \mathit{fld}(p_i), \hat{f}(u) = \hat{f}(v)\}$. Then clearly,

$\langle q, X\rangle$ is a bisimulation of $\langle p \cup p_i, X\rangle$. As $p \cup p_i \subseteq q$, q is a bisimulation of $\langle p \cup p_i, X\rangle$ for T. As $\mathcal{X}(\langle p, X\rangle) = \emptyset$, also $\mathcal{X}(\langle q, X\rangle) = \emptyset$. Hence $\mathcal{X}(\langle q, X\rangle) = \mathcal{X}(\langle p, X\rangle)$.

Secondly let $k \neq 0$ be the number of equivalence classes of $\mathcal{X}(\langle p, X\rangle)$ and suppose that (1) implies (3) whenever the number of equivalence classes of $\mathcal{X}(\langle p, X\rangle)$ is less than k. Let $x \in carri(\mathcal{X}(p))$ and let $a \in \Delta \setminus (X \cup \mathcal{V}_\Delta(p))$ be a new atom in Δ. For all y such that $(x, y) \in \mathcal{X}(p)$, replace y appearing in p and p_j $(j \in \Lambda)$ by a to obtain p' and p'_j $(j \in \Lambda)$, respectively. Then the number of equivalence classes of $\mathcal{X}(\langle p', X\rangle)$ is $k-1$, and $\langle p', X\rangle$ and $(\langle p'_j, X\rangle)_{j\in\Lambda}$ satisfy (1). Then, by induction hypothesis, there is a bisimulation $\langle q', X\rangle$ of $\langle p' \cup p'_i, X\rangle$ for T for some $i \in \Lambda$ such that $\mathcal{X}(\langle q', X\rangle) = \mathcal{X}(\langle p', X\rangle)$. Let $q = \{l \in V \times V \mid l^a_y \in q', (y, x) \in \mathcal{X}(\langle p, X\rangle)\}$, where l^a_y means the term obtained from l by replacing all occurrences y in l with the atom a. It is an easy verification to show that $\langle q, X\rangle$ is a bisimulation for T of $\langle p \cup p_i, X\rangle$ and $\mathcal{X}(\langle q, X\rangle) = \mathcal{X}(\langle p, X\rangle)$. So we get (3). □

Corollary 3.35 *Let $X \subseteq \Delta$ be finite and $(\langle p_i, X\rangle)_{i\in\Lambda}$ be a family of non-empty constraints on $I(T_X)$. Then the following are equivalent:*

(1) For every assignment f for X in T, there is some $i \in \Lambda$ such that f satisfies $\langle p_i, X\rangle$.

(2) For some $i \in \Lambda$, every assignment for X in T satisfies $\langle p_i, X\rangle$.

(3) There is some $i \in \Lambda$ such that $p \subseteq \{(x, x) \mid x \in I(T, X)\}$.

Proof Let $p = \{(x, x) \mid x \in X\}$. Then p and the family of constraints satisfies the premise condition of Proposition 3.34. So condition (1), (2), (3) in the proposition are equivalent. Also it follows that Condition (3) of this corollary is equivalent to the condition (3) in Proposition prop:free-parameter. Hence, as p is satisfied with all assignments for $\langle p, X\rangle$ in $J(T)$, it follows that the condition (1) and (2) of this corollary are equivalent. □

Theorem 3.36 (Independence Theorem) *Let T be set-based and conservative, $X \subseteq \Delta$ be finite, and let $\langle p, X\rangle$ be a bisimulation on $I(T_X)$. Let $\langle q, X\rangle$ a constraint on $I(T_X)$. Then the following are equivalent.*

(1) The constraint $\langle p, q, X\rangle$ is solvable in $J(T)$.

(2) $\langle p \cup \{l\}, X\rangle$ is solvable T for each $l \in q$.

Proof (1)$\Longrightarrow$ (2): Obvious.

(2)$\Longrightarrow$ (1): Suppose (2) is true but (1) is not true. Equivalently, suppose that, for any solution f of $\langle p, X\rangle$ in $J(T)$), $\hat{f}(u) = \hat{f}(v)$ for some $(u, v) \in q$. Then by Proposition 3.34 there exist some $(u, v) \in q$ and a bisimulation $\langle p', X\rangle$ on $J(T)$ of $\langle p \cup \{(u, v)\}, X\rangle$ such that $\mathcal{X}(\langle p', X\rangle) = \mathcal{X}(\langle p, X\rangle)$. It follows that this contradicts (2). Therefore we get (1). □

Remark An analogy from a simple set theory might be helpful for understanding the independence theorem: Given sets $p, p_1, \cdots, p_n \subseteq S$, the following are equivalent, where $\overline{x} \overset{\text{def}}{=} S \setminus x$, i.e., the complement of x.

- $p \cap \overline{p_1} \cap \cdots \cap \overline{p_n} = \phi$.

- $p \subseteq p_1 \cup \cdots \cup p_n$.

In addition to this general property, the independence theorem rests on a special property that the 'total space' S can not be covered by any finite family of 'subspaces' with properly lower 'dimensions' than that of S. Of course, this is not the case in general. For example, take the negative constraint,

$$\langle \emptyset, \{(x,a),(x,b)\}, \{x\}\rangle$$

in the domain $S = \{a,b\}$. Both of $\langle \emptyset, \{(x,a)\}, \{x\}\rangle$ and $\langle \emptyset, \{(x,b)\}, \{x\}\rangle$ are satisfiable, but the total constraint is unsolvable in S. □

We prove that every set-based and conservative functor is solution compact.

Solution Compactness

Theorem 3.37 (Compactness Theorem) *Let T be a set-based and conservative functor. Then for any countable constraint $\langle p, X\rangle$ for T, the following are equivalent.*

(1) $\langle p, X\rangle$ has a bisimulation for T.

(2) $\langle q, X\rangle$ has a bisimulation for T for every finite subset q of p.

Proof (1) $\Longleftrightarrow$ (2): Obvious.

(2) $\Longleftrightarrow$ (1): Suppose (2) is true. As p is countable, we can choose a countable chain $d_0 \subseteq d_1 \subseteq \cdots$ of finite subsets of p so that $p = \bigcup\{d_i \mid i \in N\}$. Let $D = \{d_i \mid i \in N\}$. Let C be the set of ordered pairs (d, b) of $d \in D$ and a minimal local bisimulation b of d for T. Clearly C is an infinite set. Let $\leq$ be a binary relation on C defined by $(d,b) \leq (d',b') \iff d \subseteq d'$ and $b \subseteq b'$. Clearly C is partially ordered by $\leq$. As every finite subset of p has a bisimulation for T, for any $d \in D$, $(d,b) \in C$ for some bisimulation b for T.

If $(d,b) \in C$, $d' \in D$ and $d' \subseteq d$, then by Proposition 3.25 there is a bisimulation b' for T such that $b' \subseteq b$ and $(d',b') \in C$. Hence C is connected. For each element x of C there exists at most a finite number of direct successors of x. We define a family $(x_n)_{n\in N}$ of elements of C such that x_{n+1} is a direct successor of x_n for all $n \in N$. Define $x_0 = (\emptyset, \emptyset)$. Note that x_0 has a infinite number of successors in C. We assume that x_n has been defined and has a infinite number of successors in C. Choose x_{n+1} among the direct successors of x_n so that x_{n+1} has infinitely many successors in T. With $x_n = (d_n, b_n)$ thus defined for $n \in N$, as the union of a monotone sequence of bisimulations for T forms also a bisimulation for T, it follows that $p = \bigcup\{d_n \mid n \in N\}$ and $\bigcup\{b_n \mid n \in N\}$ is a bisimulation of p for T. □

3.6.6 Record Merge in a Final Coalgebra for Π_A

Let $F, A \subseteq \Delta$ be disjoint two sets. Elements F are called *features*. A special atom $\perp \in \Delta \setminus (F \cup A)$ means an undefined values of algebras. Define a class functor

$$\Pi = map(F, -).$$

Clearly Π_B is pure, set-based and subterm-closed for any $B \subseteq \Delta$. Elements of $J(\Pi_A)$ are called *records* over (F, A).

We define a merge operation on records as a coalgebra $(pow(J(\Pi_A)), \mu)$ for $\Pi_{\{\perp\}\cup A}$ by

- $\mu(u) = \perp$ if there are $x, y \in u$ such that $x \neq y$ and either $x \in A$ or $y \in A$.
- $\mu(u) = a$ if $a \in A \cap u$.
- $\mu(u) = f_u$ if u is a set, where $dom(f_u) = \bigcup\{dom(g) \mid g \in u\}$, $f_u(\nu) = \{f(\nu) \mid f \in u, \nu \in dom(f)\}$.

μ is well-defined. By the final coalgebra theorem, there is a function $\pi: pow(J(\Pi_A)) \to J(\Pi_{A\cup\{\perp\}})$ such that $\pi(u) = \Pi_{\{\perp\}\cup A}(\pi)(\mu(u))$, for $u \in pow(J(\Pi_{\{\perp\}\cup A}))$. The operation μ is a *record merge operations.* Define $u^* = \Pi_{\{\perp\}\cup A}(\pi)(\mu(u))$.

Example 3.13

$$\begin{array}{rcll} \{\{(\nu, a)\}, \{(\nu, b)\}\}^* & = & \perp, & \text{where } a \neq b \in A. \\ \{\{(\nu_1, a)\}, \{(\nu_2, b)\}\}^* & = & \{(\nu_1, a), (\nu_2, b)\}, & \text{where } \nu_1 \neq \nu_2 \in F. \end{array}$$

□

We use $\sqsubseteq_A$ for the maximum $\emptyset$-subsumption relation on $J(\Pi_A)$. The notion of a solution of X-subsumption constraint on $I(\Pi_{A\cup X})$ in $J(\Pi_A)$ is defined in a similar way to the bisimulation constraint.

Proposition 3.38 *The following are equivalent.*

(1) For all $w \in v$, $w \sqsubseteq u$.

(2) $u^* \sqsubseteq v$.

The following definition is a slight modification of Barwise [14].

Definition 3.59 Let $R = J(\Pi_{A\cup X})$ for disjoint $A, X \subseteq \Delta \setminus F$. A constraint $\langle p, X\rangle$ is called a *(X-)subsumption* on R if p is reflexive and symmetric binary relation on R satisfying the following.

(1) If $x \in A$, $y \notin X$, $p(x, y)$ then $x = y$.

(2) If $x \in A$, $y \notin X$, $p(y, x)$ then $x = y$.

(3) If $x, y \notin \Delta$, $p(x, y)$ then $dom(x) \subseteq dom(y)$ and $p(x(\nu), y(\nu))$ for all $\nu \in dom(x)$.

□

We define an binary operation on records for merging *parametric* records. Let $b = (b_x)_{x\in X}$ be a system of equations such that $b_x \in I(\Pi_{A\cup X}) \setminus X$. Then:

Definition 3.60 $\mu_b(u) \stackrel{\text{def}}{=} \mu(u')$, where $u' = \{f \in u \mid f \in u \setminus X\} \cup \{b_x \mid x \in u \cap X\}$. □

Definition 3.61 Given a X-bisimulation p and X-subsumption on $I(\Pi_{A \cup X})$. A *record compatibility relation* r is a binary relation on $I(\Pi_{A \cup X})$ defined w.r.t. p, q satisfying the following.

(1) $p \subseteq r$.

(2) If for some z, $q(x, z)$ and $q(y, z)$, then $r(x, y)$.

(3) If $r(x, y)$, then for all $\nu \in dom(x) \cap dom(y)$, $r(x(\nu), y(\nu))$.

□

Given p, q, r as the above, for each unbound $x \in X$ in p, define $R_x = \{y \mid r(x, y), y \in X\}$. By the final coalgebra theorem, $\pi(x) \in J(\Pi_{\{\perp\} \cup A})$. Moreover, as each equivalence class of r has no *conflict*, by the final algebra theorem, $\perp \notin trans(\pi(x))$. This method contains decision procedure for subsumption problem, which recently Dörre [28] first solved by using a well-known method for transforming non-deterministic finite automata into deterministic ones. Our work gives an account to the solution from the final coalgebra theorem [1, 5].

We can have another type of merge operations. This merge will be used for a specification of a multiple-inheritance of class-hierarchy system in Section 5.6.5. Let A and F be the same as above. Let $\Pi' = pow(A) \times map'(F, -)$ and $\mu'(u) = (\bigcup \alpha \mid (\alpha, r) \in u,$ for some $r, \{r(i) \mid (\beta, r) \in u,$ for some $\beta\})$. Then $(pow(J(\Pi')), \mu')$ is a coalgebra for Π'. Define u^+ similarly to the above u^*.

Example 3.14

$$\begin{aligned}
\{(\emptyset, \{(\nu, a)\}), (\emptyset, \{(\nu, b)\})\}^+ &= (\emptyset, \{(\nu, \{a, b\})\}), \quad \text{where } a, b \in A. \\
\{(\{a\}, \emptyset), (\{b\}, \{(\nu, c)\})\}^+ &= (\{a, b\}, \{(\nu, c)\}), \quad \text{where } a, b, c \in A, \nu \in F.
\end{aligned}$$

□

3.7 Compactness of the Complex Number Field

A few examples from domains of numbers may be useful for the reader to get intuitions on infinite sets of constraints. Let C be the complex number field. C is compact with respect to countable families of algebraic equations. As this fact has been turned out to be a fundamental new fact, we give a proof for this in the subsection below. Now let R be the real number field. R is not compact with respect to algebraic constraints. To be more precise, let S be the set of equations

$$x_n = x_{n+1}^2 + 2$$

for non negative integers n. The constraint S has no solution in R, for otherwise we can infer $x_0 \geq 2^m$ for any $m > 0$, which is impossible. Also R is not compact with respect to the theory with inequality ($<$). To see this, take the set of inequalities $n \leq x$ for positive integers n, which has no solution in R. However it is easy to see from the elementary calculus that R is solution compact with respect to algebraic constraints with inequality ($<$).

Finally let K be any field. For instance, let K be C and let S be a countably infinite set of linear finitary equations over K. As the well known Gaussian sweeping-out method can be applied also for such infinite systems of linear equations with countable many variables, it is proved that S is solvable if and only if every subset of S is solvable. Now we have explained the properties of C and R, which are summarized as follows:

(1) C is compact with respect to *countable* algebraic equational constraints.

(2) R is not compact with respect to algebraic equation constraints.

(3) Both C and R are compact with respect to linear equation constraints.

(4) R is not compact with respect to inequality constraints.

Compactness of the complex number field might be a folklore theorem in algebraic geometry. However, the result seems interesting in that the theorem involves a countable number of variables due to the situation of perpetual process, which is an usual in and a quite opposite direction of interests of standard algebraic geometry.

We prove that every countable family of algebraic equations is solvable in the field of complex numbers if and only if so is every its finite subfamily. This result assures the solvability for incremental algebraic constraints in perpetual process, e.g., Gröbner-base constraint solver such as the one built-in in the system CAL [80, 81]. We use the adjective *countable* for the adjective phrase "at most countably infinite". For instance, by a countable set, we mean a set which is at most countably infinite.

Theorem 3.39 *Every countable family S of algebraic equations over the field C of complex numbers is solvable in C if and only if every finite subfamily of S is solvable in C.*

Proof Suppose S is a countable family of algebraic equations such that every finite subfamily of S is solvable in C. Let X be the set of variables appearing in S. As it follows from Hilbert's zero point theorem that if every ideal generated by finite subfamily of S has not the unit 1, also the ideal I_S of $C[X]$ generated by S has not 1. Hence by Zorn's Lemma there exists a maximal ideal M of $C[X]$ such that $I_S \subseteq M$ and M has not 1. Let $K = C[X]/M$. Then by theorem 3.40, K is an algebraic extension of C. As C is algebraically closed we get $K = C$. So M must be an ideal which is generated by a family $\{x - a_x\}_{x \in X}$, where a_x is an element of C. Then we have a solution $f\colon x \mapsto a_x$ of S. □

Remark We show that the complex number field C is not compact with respect to the class of uncountable families of algebraic equations. Let X and Y be disjoint sets of variable which have the same cardinality of the power set of C. Let $h : X \times X \to Y$ be an injection. Let Z be the set $\{(x-y)h(x,y) - 1 \mid x \neq y,\ x \in X,\ y \in X\}$. Every finite subset of Z is solvable in C but Z is never solvable in C, for otherwise we have an injection $\psi\colon X \to C$, which means that the cardinality of C is larger than or equal to that of X. This is a contradiction.

□

Remark There is a proof by Oscar Zariski of a fundamental theorem in the field theory that every finitely generated ring over a field F is an algebraic extension of F whenever the ring forms a field. We show that a modified Zariski's proof can be applied to the case for the complex field C and rings which has a countably infinite number of generators. □

Theorem 3.40 *Let F be a field which has an uncountable cardinality and let B $(B \supset F)$ be a commutative ring generated by a countable number of elements over F. If B is a field, then B is an algebraic extension of F.*

Proof Let χ be a countable set of generators of B as a ring over F, i.e., $B = F[\chi]$. Let χ' and χ'' be subsets of χ such that χ' is a transcendental basis of B and $B = (F[\chi'])[\chi'']$. If χ'' is empty, as $B = F[\chi']$ is a field, χ' must be empty. So the theorem holds. Suppose that neither χ' nor χ'' are empty. Let $A = F[\chi']$ and let K be the quotient field of A. Write $\chi'' = \{x_1, x_2, \cdots\}$. There exists a family $\{f_i\}_{i>0}$ of polynomials with one variable over K such that x_i is a root of f_i for each $i > 0$. Let S be the multiplicatively closed subset of A generated by the set of denominators of coefficients of f_i for $i > 0$. As F is uncountable and χ'' is countable, S can not have all irreducible polynomials in A.

Let $A_g = S^{-1}A(\subseteq K \subseteq B)$. Every coefficient of f_i belongs to the quotient ring A_g. Hence, as $x \in \chi''(\subseteq B)$ is integral with respect to the sub-integral domain A_g, $B(= A_g[\chi''])$ is integral with respect to A_g. So in particular, K is integral with respect to A_g. We can prove as a routine that $A = F[\chi']$ is integrally closed. Hence so is $A_g = S^{-1}A$. On the other hand, as the quotient filed of A_g is K and K is integral with respect to A_g, we get $A_g = K$. So A_g must be a field. But from the remark about S above it is impossible for A_g to be a field. Therefore χ' must be empty. □

Remark The following is a standard theorem: Let F be a field and let B $(B \supseteq F)$ be a commutative ring generated by a *finite* number of elements over F. If B is a field, then B is an algebraic extension of F with a *finite* degree. □

3.8 Concluding Remarks

Using hypersets and compact constraints, declarative and operational semantics have become essentially the same. In this respect, the present semantics seems to be closely related to constructive type theory [50] in that the meaning of a given goal is a proof tree that is decorated by satisfiable constraints at each node. The goal can be viewed as a noncanonical constraint. Also, the solution compactness requirement of the CLP scheme is close to the intuitive notion of constructive approximation. A clearer relationship between these theories remains to be discovered.

The original purpose of the present work was to describe semantics of meta-predicates of logic programming languages using basic ideas in situation theory and situation semantics (STASS). Meta-predicates such as *var* or *cut* of Prolog are essentially defined operationally. So the clear structure of the proposed semantics in hypersets is expected to provide a good new setting for defining the semantics of these meta-predicates. For example, using the idea from STASS that the meaning of a sentence is a relation between situations, meaning of commands will be formalized as relations between computation trees. Moreover, constraint logic programming certainly has aspects of *infon logic programming*, seeing constraints as infon or soa in the sense of the STASS literature. For instance, it can be seen that the infon $x = y$ is supported by a physical computation state s, i.e.,

$$s \models \ll =, x, y \gg .$$

The situation or state s has parameter cells for x and y with pointers from x to y. However, details are outside the scope of this thesis.

Finally, we have seen that the set-based, form-based, and conservative functors on the category of classes characterize constraint theories on the coalgebras for the functors as built-in constraints of the constraint logic programming scheme. In particular, we have seen that these functors are a generalization of the Colmerauer's infinite tree unification with unequations.

Chapter 4

Constraint Logic Programming over Record Algebras

A new class of algebras called record algebras is introduced as a mathematical model for feature structures. A record algebra is commutative and idempotent partial monoid R provided with an operator domain G. G is a monoid whose elements act on R from both left and right sides. R is called a G-record algebra and G a feature monoid. The record algebra is an extension of the feature algebra in computational linguistics. A constraint theory $(R, \bowtie)$ is given for complete and standard record algebras R with a record compatibility relation $\bowtie$. The standard unification theory $(H, =)$ on Herbrand universe H with the identity relation $=$ is embedded into $(R, \bowtie)$. Also based on the theory $(R, \bowtie)$ a logic programming over $(R, \bowtie)$ is defined with both declarative and operational semantics, where the maximum semantics is used for the declarative semantics. It is shown that both semantics of the program are sound and complete including the negation-as-failure rule. As an application of this logic programming language class, the definite clause grammar over H is generalized to that over G-record algebras.

4.1 Introduction

In this chapter, we propose a class of algebras called record algebras as a mathematical model of feature structures. In fact, this structure was first introduced into the standard logic programming language Prolog as an extension of the first-order term and it turned out to be very useful for natural language processing [64, 55]. We develop a unification theory using parametric records, taking them as a partial description of pure records, i.e., parameter-free ones. We write $(R, \bowtie)$, informally, for the proposed record unification theory, where R is the domain of pure records and $\bowtie$ a binary relation on R. We develop the theory $(R, \bowtie)$ so that it is a straightforward extension of the standard unification theory $(H, =)$ over Herbrand universe H, i.e., the domain of pure first-order terms. In fact, the former will be proved to be a conservative extension of the latter. Note that we treat only one relation $\bowtie$ as the first-order unification theory does only the identity relation $=$.

Records are simple recursive structures of the form $\{(a_1, r_1), \cdots, (a_n, r_n)\}$, where a_i are distinct *atomic features* and r_i are atoms or possibly another records. r_i are also called *features*. A formal definition will be given in section 4.2 to the record. We treat three familiar kinds of builtin operations on records: 'merge'$(+)$, 'left'$(\cdot)$, and 'right' $(/\!\!/)$.

Example 4.1 a, b, c, d are features, r, s, t are atoms.

$+$ (merge)	:	$\{(a,r),(b,s)\}+\{(b,s),(c,t)\}$	$=$	$\{(a,r),(b,s),(c,t)\}$.	
$\cdot$ (left)	:	$a\cdot\{(b,r),(c,s)\}$	$=$	$\{(a,\{(b,r),(c,s)\})\}$.	(See Figure 4.1.)
$/\!\!/$ (right)	:	$\{(a,\{(b,r),(c,s)\}),(d,t)\}/\!\!/a$	$=$	$\{(b,r),(c,s)\}$.	(See Figure 4.2.)

□

Note that operations are partial in general. For instance $\{(a,p)\}+\{(a,q)\}$ is undefined when p and q are distinct 'atoms'. Using this algebra, each record $\{(a_1,r_1),\cdots,(a_n,r_n)\}$ $(n\geq 1)$ is expressed as $a_1r_1+\cdots+a_nr_n$, where ax stands for $a\cdot x$.

With these simple examples in mind, a *G-record algebra* will be formally defined to be 6-tuple $(R,G,+,\cdot,/\!\!/,\epsilon)$, where R is a commutative and idempotent partial monoid under $+$ with an operator domain G which acts on R from both left$(\cdot)$ and right $(/\!\!/)$ sides as a monoid. We call G a feature monoid and R a G-record algebra. An element of R is a *record*. An element of G is a *feature*. ϵ is the *unit* of R with respect to $+$. As a convention for the case $n=0$ above, we identify the empty record $\emptyset$ with ϵ.

We have no type other than the record. Data such as numbers and strings which come at terminals of record structures are *atomic* records. A standard example of records are trees which have tags at only leaf nodes. Let R be the set of such trees. A feature is a path in the tree starting from the root node. The set of features forms a monoid under the concatenation. $x/\!\!/\alpha$ denotes the subtree of the tree x which can be accessed along the feature α in x. αx denotes the tree which is obtained by putting the tree x at the end of the feature α. $+$ is a tree merge. ϵ is a singleton tree which is not assigned a tag. Unlike the standard unification theory, $(H,=)$, record algebras, R, involve partiality. In fact, R is a partial semi-lattice structure w.r.t. (with regard to) $\leq$, where $a\leq b$ is defined by $a+b=b$.

Atomic constraints on records are of the form $p\bowtie q$, where p and q are record terms. The symbol $\bowtie$ means a binary relation on R representing compatibility of records, i.e., $u\bowtie v$ iff $u+v$ is defined in R, where $u,v\in R$. A *solution* of $p\bowtie q$ is an assignment f such that there is a common 'instance' t in R of p and q. In other words, $p\bowtie q$ is solvable iff there is a record $t\in R$ which is of both parametric types p and q. Also another informal reading of $p\bowtie q$ is that there is a record $t\in R$ such that p and q are a partial description of t. The concept of solutions in $(R,\bowtie)$ is a natural extension of that in the standard theory $(H,=)$. The notion of solutions in $(R,\bowtie)$ is fundamental and will be defined formally. The relation $\bowtie$ is not an equivalence relation. In fact, there is no full transitive rule in the theory $(R,\bowtie)$ but only a restricted one: the transitive rule

$$x\bowtie p\wedge x\bowtie q\Longrightarrow p\bowtie q$$

is applied in the unification process only when x is a parameter.

A *unification theory* of the record algebra can be seen as a closure operation on constraints. In fact, a constraint is defined to be *unifiable* iff it has a consistent closure. The closure is, roughly, a generalization of the unifier. For example, in $(R,\bowtie)$ the closure of the constraint $\{(a,y),(b,y)\}\bowtie\{(b,1),(c,y)\}$ contains $y\bowtie 1$, which means $y=1$ with 1 being atomic. We will show that for every constraint C on the record algebra R, the constraint C is unifiable iff C is solvable in R. In fact, our unification theory over records is satisfaction complete in the sense of the logic programming (CLP) scheme [39]. Thus, the unification theory $(R,\bowtie)$ characterizes the set of solvable constraints in a decidable way. Also we show that the unification theory will be compact in the sense that a constraint is solvable iff so is its every finite subconstraint.

Now we turn to how to build the record algebra into logic programming. We view the logic programming over records as a form of inductive or coinductive definition for domains of records [1, 56]. So the semantics of the program is defined to be the maximum or minimum fixpoint of the program viewed as a monotone transformation on the power set of the domain R of records. In particular, we are interested in non-well-founded structures [1] such as streams for a variety of possible applications, e.g., type inference involving recursive type definitions among others. So unlike traditional semantics of logic programming, we treat in this paper only the maximum semantics of programs [46].

Example 4.2 The program below consists of three Horn clauses for a recursive data type definition of list structures, where x and l are parameters, a, b, nil, $atom$, $list$ are atoms, $type$, car, cdr, and $form$ are features.

(1) $\{(type, atom), (form, a)\}$.

(2) $\{(type, atom), (form, b)\}$.

(3) $\{(type, list), (form, \{(car, x), (cdr, l)\})\}$:-
$\{(type, atom), (form, x)\}, \{(type, list), (form, l)\}$.

The minimum semantics of the program is the set

$$\{\{(type, atom), (form, a)\}, \{(type, atom), (form, b)\}\}.$$

On the other hand, the maximum semantics of the program is the largest set M of records such that:

- $M = N \cup \{\{(type, atom), (form, a)\}, \{(type, atom), (form, b)\}\}$.
- N is the largest set such that for any $r \in N$, there are some $v \in M$ and $u \in \{a, b\}$ such that $r = \{(type, list), (form, \{(car, u), (cdr, v)\})\}$.

This kind of straightforward maximum semantics is given in [58] based on the hyperset theory [1]. This program example will be treated formally in example 4.11 and 4.12. □

We will show the soundness and completeness results and the display-theorem (theorem 4.17), which asserts that every solution is displayed by a computation for the given goal. Also we will show the soundness and completeness of the negation-as-failure rule. However our constraint language does not exactly fit to the CLP scheme. In fact, as the atomic constraint $\epsilon \bowtie x$ holds for any record x, the unit ϵ is not *constraint definable*, which means $(R, \bowtie)$ is not solution compact [39]. This aspect of $(R, \bowtie)$ may be easily modified by separating the role of the symbol $\bowtie$ into two, i.e., identity ($=$) and subsumption ($\sqsubseteq$) so that the modified language is solution compact [58]. This modification is, however, out of place.

There is a straightforward translation from Herbrand universe H into the record algebra R which maps, for instance, $f(a, b)$ to $\{(f_1, a), (f_2, b)\}$, where f_1 and f_2 are argument places of f. The first-order term $f(a, g(x))$, for instance, is translated to the record $\{(f_1, a), (f_2, \{(g_1, x)\})\}$. Partial descriptions $\{(f_1, a)\}$ and $\{(f_2, b)\}$ together mean the first-order term $f(a, b)$, whereas there is no such term for $\{(f_1, a)\}$ and $\{(f_1, b)\}$ with a, b being distinct atoms. This kind of partiality is an essential aspect of records structure constraints which are not seen in the standard unification theory.

As applications of the unification theories $(R, \bowtie)$, we will show first that the standard domain of $(H, =)$ is embedded into a record algebra of $(R, \bowtie)$. Then we extend the DCG (definite clause grammar) over H to that over the record algebra R. Also we will see that DAGs used in unification grammar formalism are viewed constraints on records.

We will give the details of the unification theory $(R, \bowtie)$ over record algebras R which is a conservative extension of the standard unification theory $(H, =)$ over Herbrand universe H. The standard term unification $f(x, x) = f(a, y)$, for instance, in $(H, =)$ is translated to $\{(f_1, x), (f_2, x)\} \bowtie \{(f_1, a), (f_2, y)\}$ in $(R, \bowtie)$ preserving the solvability. The closure of the latter contains three atomic constraints: $x \bowtie a$, $x \bowtie y$, and $y \bowtie a$, which means $x = y = a$. More formally speaking, there is a translation $\tau : H \to R$ between the first-order term unification (including infinite trees) $(H, =)$ and the proposed record unification theory $(R, \bowtie)$ such that $s = t$ is unifiable in $(H, =)$ iff $\tau(s) \bowtie \tau(t)$ is unifiable in $(R, \bowtie)$ and also that $s = t$ is solvable in $(H, =)$ iff $\tau(s) \bowtie \tau(t)$ is solvable in $(R, \bowtie)$. A non trivial thing is that the binary relation symbol $\bowtie$ between records is not interpreted as the record identity but as a kind of compatibility of two records[1]. In fact, an extended notion of solutions for record constraints is needed to make the record unification complete as desired. Moreover, there is a converse mapping from the record domain R to the standard term domain H in the sense that the record unification can be reduced to the standard term unification. For example, take a compatibility constraint (4.1).

$$\{(a, y), (b, y)\} \bowtie \{(b, 1), (c, y)\}. \tag{4.1}$$
$$\{(a, y), (b, y), (c, u)\} = \{(a, v), (b, 1), (c, y)\}. \tag{4.2}$$
$$f(y, y, u) = f(v, 1, y). \tag{4.3}$$

The compatibility constraint is reduced to the equational constraint (4.2) introducing new parameters u and v for 'hidden' features c and a. Clearly, the two constraints (4.1) and (4.2) are equivalent in the sense that the equation is solvable iff so is the given constraint. Now the equation (2) is equivalent to the term equation (4.3), where f is a function symbol. The present work, however, concerns $\bowtie$-constraints themselves as a study of partiality of information, without reducing them to standard constraints.

There are many related works on feature structures, which are still on going, so that we take only some of representative works among them related to this work and give brief notes on them from the view point of this work. First of all Pereira and Shieber [71] applies Scott's domain theory to give a denotational semantics to unification grammars viewing them as a computer program. Although it is not clear that the record algebra is an instance of the scheme, both works share the basic idea in that grammars are computer programs. In fact, the present work treats DCG as a form of coinductive definitions for a desired domain of records as legal feature structures.

As explained above by introducing new parameters for 'hidden' features, the proposed record constraints can be translated into first-order term equations or more generally those in the order sorted algebra (OSA) [34]. However it is the behavior of the relation $\bowtie$, i.e., a logic of 'compatibility' that the present work is interested in. In fact, we give a record constraint

[1]Although the proposed theory of $\bowtie$ on records is not exactly an equational theory, we still use the word 'unification' for such a theory based on a strong similarity to each other.

for only $\bowtie$ relation, and thereby we need no new parameters in unification procedures. Also the proposed record algebra has non-well-founded structures in general and the semantics of programs is the maximum semantics. It is interesting but not clear how the initial algebra semantics of OSA, for instance, can be applied to the record algebras.

PATR-II [83] is a standard computational framework for DAG-based unification grammars. Our unification over the record algebra is equivalent to that of PATR-II over feature structures, i.e., graph merging, though we includes infinite record structures. The proposed embedding of the standard domain $(H, =)$ into the record unification $(R, \bowtie)$ gives a mathematical model to a common sense view that Herbrand terms are a *degenerated form* of records and a DCG is a compiled form of unification grammar (Shieber). Also we will show a natural translation of DAGs into constraints in $(R, \bowtie)$ so that the unification theory over DAGs can be interpreted as a constraint theory over the record algebra in a natural way. In other words records are considered to be denotations of directed graphs (DGs), which is an analogy to that hypersets are denotations of directed graphs [1].

Seeing records as partial functions, the record algebra is related to situation semantics [18] and situation theory [12, 14]. That is, records can be used for representation of *state of affairs* which contains a partially specified list of arguments [55]. Pollard [77] proposed the notion of *anadic* relations for situation semantics. The record algebra serves as a model for anadic relations.

In his thesis, Aït-Kaci [7] criticizes first-order terms from type-theoretical point of view and proposes to see them as records. He uses semi-lattice for the framework. Aït-Kaci [7], however, gives no declarative semantics of the program over the semi-lattice while we give a declarative semantics for constraint logic programming over record algebras.

Kasper and Rounds [43] proposes automata models of unification theory over feature structures. Also more recently Smolka [86] formalizes unification theories in feature logic with subsorts including negation and disjunction. He showed a linear time translation from constraint language over feature structures into a quantifier-free sublanguage of first-order predicate language.

Courcelle [25] treats infinite trees, and Maher [49] gives an axiomatization of infinite trees. However, they treat only trees which have fixed arities. Also the same with case of Colmerauer [21], whereas the record has not a fixed arity.

This chapter is organized as follows. In Section 4.1, we introduce a class of record algebras and give a unification theory over them. The main goal is the equivalence theorem between the solvability and unifiability. In Section 4.2 we give a class of unification grammars over record algebras being guided by the idea that grammars are computer programs [71]. Soundness and completeness results are obtained. DAGs and the notion of arity will be given a new interpretation respectively from the point of the record algebra. Section 4 conclude the chapter.

4.2 The Record Algebra and Unification

We introduce record algebras and describe a unification theory over them. As things in record algebras are partial, we make a general convention for equations and evaluation in partial algebras used in the rest of the paper. By $e\downarrow$, we mean that the expression e is defined, i.e., has a value, where e is an expression in the constraint language. Details of 'logic of partial terms' will be given at appropriate places in the below. An equation $l \simeq r$ means that if either l or r

is defined then both of them have the same value:

$$l \simeq r \iff [l\downarrow \vee r\downarrow \Longrightarrow l\downarrow \wedge r\downarrow \wedge l = r],$$

where l and r are expressions. The use of terminologies follows standard algebra text books such as [38, 94, 19].

4.2.1 The Record Algebra

Definition 4.1 A *feature monoid* G is a monoid such that the following hold:

(1) Each $\alpha \in G$ has only a finite number of prefixes, where $\beta \in G$ *is a prefix of* $\alpha \in G$ if $\alpha = \beta\gamma$ for some $\gamma \in G$.

(2) $(G, \leq)$ is a partial order structure, where $\leq$ is a binary relation over G defined by $\alpha \leq \beta$ iff α is a prefix of β.

(3) $(G, \leq)$ forms a tree, i.e., the set of $\{\beta \in G \mid \alpha \leq \beta \leq \gamma\}$ is totally ordered by $\leq$ for any $\alpha, \beta \in G$.

□

Elements of G are called a *feature*. Every free monoid is a feature monoid. We write $\alpha \not\sim \beta$ iff α and β are *incomparable* features, i.e., $\alpha \not\sim \beta \iff \alpha \not\leq \beta \wedge \beta \not\leq \alpha$ in G. α and β are *incompatible* iff there is no upper bound of α and β in G. As the unit $\varepsilon \in G$ is the minimum element of $(G, \leq)$, it follows from assumption (3) that α and β are incomparable in G iff they are incompatible in G.

The feature monoid is a generalization of monoids of strings over given letters with the string concatenation. Let L be a set of *atomic features*. A sequence of atomic features $a_1, \ldots, a_n$ is written $\langle a_1, \ldots, a_n\rangle$, where n is a non negative integer and is called the length of the sequence. As usual convention, the sequence $\langle a_1, \cdots, a_n\rangle$ denotes the *empty string* $\langle\rangle$ when $n = 0$. Also ε denotes the empty string. The length of the empty string is 0. We write $\alpha\beta$ for the *concatenation* of two sequences α and β. The concatenation is defined by the following equations:

$$\begin{aligned} \varepsilon\alpha &= \alpha. \\ \alpha\varepsilon &= \alpha. \\ \langle a_1, \ldots, a_n\rangle\langle b_1, \ldots, b_m\rangle &= \langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle. \end{aligned}$$

The symbol X^* denotes the free monoid generated over a set X. For example, with L being a set of letters, L^* is the set of words of finite length over letters in L. For the convenience of notation the string $\langle a\rangle$ of the length one is written simply a. Clearly from the definition, L^* is a feature monoid.

Remark Condition (3) above is narrow. It is desirable to find more general conditions for (3) in which a computational unification theory such as one developed in the present work is still effective. However this is an open problem. □

Definition 4.2 Let G and M be sets and let $F \subsetneq \{\cdot, +, /\!\!/\}$. Then $\mathcal{E}(G, M, F)$ is the least set E of *expressions* which satisfies the following.

(1) $M \subseteq E$.

(2) If $\cdot \in F, \alpha \in G, x \in E$ then $\cdot(\alpha, x) \in E$.

(3) If $/\!\!/ \in F, \alpha \in G$ and $x \in E$ then $/\!\!/(x, \alpha) \in E$.

(4) If $+ \in F$, $x, y \in E$ then $+(x, y) \in E$.

□

We use usual notations:

$$\begin{aligned} \alpha x &\overset{\text{def}}{=} \cdot(\alpha, x). \\ x /\!\!/ \alpha &\overset{\text{def}}{=} /\!\!/(x, \alpha). \\ x + y &\overset{\text{def}}{=} +(x, y). \end{aligned}$$

Given a record term p the set $\mathcal{V}(p)$ denotes the set of parameters appearing in p.

Definition 4.3 A *merge system* is a partial semi-group $(M, +)$, where M is an associative, commutative and idempotent under a partial binary operation $+ \colon M \times M \to M$. The axioms follows, where $a, b, c \in \mathcal{E}(\emptyset, M, \{+\})$.

$$\begin{array}{lrcl} \textbf{Partiality} & a \in M & \Longrightarrow & a\downarrow. \\ & a + b\downarrow & \Longrightarrow & a\downarrow \wedge b\downarrow. \\ \textbf{Associative} & a + (b + c) & \simeq & (a + b) + c. \\ \textbf{Commutative} & a + b & \simeq & b + a. \\ \textbf{Idempotent} & a + a & \simeq & a. \end{array}$$

□

Note that a merge system may have not a unit.

Example 4.3 $(pow(P), \cup)$ is a merge system, where P is a set and $pow(P)$ denotes the set of subsets of P. □

$(M, +)$ is a *trivial merge system* if M is a set and $+$ is a binary operation on M such that $x + x = x$ but $x + y$ is undefined whenever $x \neq y$, where $x, y \in M$.

Definition 4.4 A *G-record algebra* is a 6-tuple $\mathcal{R} = (R, G, +, \cdot, /\!\!/, \epsilon)$, where

(1) $(R, +)$ is a merge system with the *unit* ϵ.

(2) G is a feature monoid which acts on R as an operator domain from both left and right:

$$\begin{aligned} \cdot &: G \times R \to R. \\ /\!\!/ &: R \times G \to R. \end{aligned}$$

(3) Partiality axioms for $\mathcal{R}$ follows, where $\alpha \in G$, and $a, b \in \mathcal{E}(G, R, \{\cdot, +, /\!\!/\})$:

$$\begin{array}{rcl} & \Longrightarrow & \alpha\downarrow. \\ & \Longrightarrow & \epsilon\downarrow. \\ a + b\downarrow & \Longrightarrow & a\downarrow \wedge b\downarrow. \\ \alpha a\downarrow & \Longrightarrow & a\downarrow. \\ a/\!\!/\alpha\downarrow & \Longrightarrow & a\downarrow. \end{array}$$

(4) The following equations hold, where $\alpha \in G$, $a, b, c \in \mathcal{E}(G, R, \{\cdot, +, /\!\!/\})$:

$$\begin{array}{lrcll} \textbf{Associative} & a + (b + c) & \simeq & (a + b) + c. & \\ \textbf{Commutative} & a + b & \simeq & b + a. & \\ \textbf{Idempotent} & a + a & \simeq & a. & \\ \textbf{Unit} & a + \epsilon & \simeq & \epsilon + a & \simeq a. \\ \textbf{Left Distributive} & \alpha(a + b) & \simeq & \alpha a + \alpha b. & \\ \textbf{Right Distributive} & (a + b)/\!\!/\alpha & \simeq & a/\!\!/\alpha + b/\!\!/\alpha. & \\ \textbf{Cancellation} & (\alpha a)/\!\!/\alpha & \simeq & a. & \end{array}$$

□

Elements of R are called *records*. Note that actions by a feature are partial in general. We abuse R for $\mathcal{R}$. In the record algebra, $x = y$ follows from $\alpha x = \alpha y$ by the cancellation axiom. For any $x \in R$, $\varepsilon x = x$ and in particular $\varepsilon\epsilon = \epsilon$ hold.

Definition 4.5 Records $x \in R$ are *atomic* if $x/\!\!/\alpha$ is undefined for any $\alpha \in G \setminus \{\varepsilon\}$. □

The unit ϵ is not always atomic. Such an example will be found in example 4.4 and 4.5. We recall the definition of the feature algebra and show that the record algebra is an extension of the feature algebra. Roughly speaking, a record algebra is a feature algebra which has an internal merge operation in addition to external ones. Moreover, the operators operate on records from both sides not only as 'field selectors' but also 'record constructors' respectively, whereas operators in feature algebras work only as selectors.

Definition 4.6 Given a set F of *features* and C of *constants*, a *feature algebra* is an ordered pair $(D, (\cdot)^{\mathcal{A}})$ which satisfies the following, where D is a set:

(1) $f^{\mathcal{A}}: D \rightarrow D$ is a partial function if $f \in F$.

(2) $c^{\mathcal{A}} \in D$ if $c \in C$.

(3) $a^{\mathcal{A}} \neq b^{\mathcal{A}}$ if $a, b \in C$, $a \neq b$.

(4) $a^{\mathcal{A}} \notin dom(f) \in D$ if $f \in F$ and $a \in C$.

□

As distinct constants are interpreted to be distinct, we can assume $C \subseteq D$. Assuming the hyperset theory [1], it is straightforward to show that every feature algebra is a record algebra.

Proposition 4.1 *Given a set F of features, C of constants, and a feature algebra $\mathcal{A} = (D, (\cdot)^{\mathcal{A}})$. Then there is a F^*-record algebra $\mathcal{R}_{\mathcal{A}} = (R, F^*, +, \cdot, /\!\!/, \epsilon)$ and an injection $\psi: D \rightarrow R$ such that the following hold.*

(1) $\psi(C) \subseteq R$.

(2) $\psi(f^{\mathcal{A}}(d)) \simeq \psi(d)/\!\!/f \qquad (d \in D)$.

Proof For each $d \in D$, let $S_d = \{(f, f^{\mathcal{A}}(d)) \mid f \in F, d \in dom(f^{\mathcal{A}})\}$ and $D' = \{d \in D \mid S_d \neq \emptyset\}$. Solve the system $\{d = S_d \mid d \in D'\}$ of equations. By the solution lemma [1], there is a unique solution ψ to the system. Let $R_0 = \{\psi(d) \mid d \in D'\} \cup D \setminus D'$ and ϵ be some new atom not in R. Let $R = R_0 \cup \{\epsilon\}$. Define a merge system $(R, +)$ so that $(R_0, +)$ is a trivial merge system and $\epsilon + x = x + \epsilon = x$ for all $x \in R$. Then define $\psi(x)/\!\!/f \overset{\text{def}}{=} \psi(f^{\mathcal{A}}(x))$ and $f \cdot \psi(x) \overset{\text{def}}{=} \psi(x_f)$, where x_f is some element in $f^{-1}(x) \subseteq D$, i.e., $x = f(x_f)$. It is clear that $(\alpha x)/\!\!/\alpha = x$ and $x/\!\!/\alpha\downarrow \Longrightarrow \alpha(x/\!\!/\alpha)\downarrow$. Also define so that $\psi(c)/\!\!/f$ is undefined for any $c \in C$ and $f \in F$. □

Definition 4.7 Let $(M, +)$ and $(M', +)$ be two merge systems. A total function $\varphi: M \rightarrow M'$ is a *merge homomorphism* if for all $a, b \in M$,

$$\varphi(a + b) \simeq \varphi(a) + \varphi(b).$$

□

Definition 4.8 Let R and R' be G-record algebras over M and M', respectively. A *homomorphism* from R into R' is a (total) function $h: R \rightarrow R'$ satisfying the following:

$$\begin{array}{rcll} h(\epsilon) & = & \epsilon. & \\ h(u) & \in & M' & (u \in M). \\ h(\alpha u) & \simeq & \alpha h(u). & \\ h(u/\!\!/\alpha) & \sim & h(u)/\!\!/\alpha. & \\ h(u + v) & \simeq & h(u) + h(v). & \end{array}$$

□

If the function h is a bijection and the inverse h^{-1} of h is a homomorphism from R' into R then R and R' are called *isomorphic* to each other.

We define a binary relation $<$ on R by

$$a \leq b \iff a + b = b.$$

Then it is proved that the binary relation $<$ is a partial order relation on R as follows: the reflective law $a \leq a$ follows the idempotent law $a + a = a$. The transitive law that $a \leq b \wedge b \leq c \Longrightarrow a \leq c$ is proved as follows: add c to both sides of the equation $b = a + b$ and then apply the equation $c = b + c$, then we obtain the equation $c = a + c$, which concludes the transitive law. The anti-symmetric law, i.e., $a \leq b \wedge b \leq a \Longrightarrow a = b$, follows directly from the commutative law of the record algebra. So we have the proposition:

Proposition 4.2 *Let R be a G-record algebra and $\leq$ the binary relation on R defined by $a \leq b \iff a + b = b$. Then $(R, \leq)$ is a partial order structure.*

Let S be a subset of a G-record algebra R. We write $\bigsqcup S$ for the supremum of S with respect to $\leq$. Also we call it the *sum* of S. A non empty subset S of R is *consistent* if there exists the sum for any two elements of S.

Definition 4.9 A *standard* G-record algebra R over M is a G-record algebra over M such that the following hold.

(1) $\alpha u + \beta v$ is defined for any $u, v \in R$ whenever $\alpha \not\sim \beta$.

(2) For each $u \in R$, there exist a set $H \subseteq G$ and a family $\{x_\alpha \in \{\epsilon\} \cup M \cup X\}_{\alpha \in H}$ such that $u = \bigsqcup\{\alpha x_\alpha \mid \alpha \in H\}$.

□

Remark The above H is not always an anti-chain, where an *anti-chain* is a subset H of G such that every distinct two elements in H are incomparable in G. For example, define a (G, M)-PTT t by

$$t \overset{\text{def}}{=} \{a^n \epsilon \mid n \geq 0\}$$

where $a \in G$, $a \neq \varepsilon$. The PTT t satisfies the constraint $x \bowtie ax$ with $x = t$ and has an exactly one branch at every node. $\{a^n \mid n \geq 0\}$ is not an anti-chain. However it is clear that if $t = \bigsqcup\{\alpha x_\alpha \mid \alpha \in H\}$ for some $H \subseteq G$ and family $\{x_\alpha\}_{\alpha \in H}$ then $H \subseteq \{a^n \mid n \geq 0\}$ and, hence, H can not be an anti-chain. □

A subset B of R is a *G-basis* if each $x \in R$ is the sum of some subset S_x of $G[B]$, i.e., $x = \bigsqcup S_x$, where $G[B] \overset{\text{def}}{=} \{gy \mid y \in B \cup \{\epsilon\}, g \in G\}$. Also we say *$R$ is generated over B* when B is a G-basis of R.

Definition 4.10 Let M be a merge system. R is a *G-record algebra over* (M, φ) if the following hold.

(1) $\varphi: M \to R$ is a merge homomorphism.

(2) $\varphi(M)$ is a G-basis of R.

(3) Every element of $\varphi(M)$ is atomic.

□

It is clear that condition (3) is equivalent both to that if $\alpha \neq \varepsilon$ then $\alpha x \notin \varphi(M)$ for any $x \in R$ and to that $x /\!\!/ \alpha$ is not defined for any $x \in M$ and $\alpha \in G$. The injection φ may be implicit when the context is clear.

Definition 4.11 R is *complete* if every consistent subset of R has a sum and the sum operation is commutative with both the left and right action of G, i.e.:

$$\begin{array}{rcl} \bigsqcup S{\downarrow} & \iff & S \text{ is consistent.} \\ \alpha x{\downarrow} & \iff & x{\downarrow}. \\ \alpha(\bigsqcup S) & \simeq & \bigsqcup\{\alpha s \mid s \in S\}. \\ (\bigsqcup S)/\!\!/\alpha & \simeq & \bigsqcup\{s/\!\!/\alpha \mid s \in S, s/\!\!/\alpha{\downarrow}\}. \end{array}$$

□

We define $a \sqcup b \overset{\text{def}}{=} \bigsqcup\{a, b\}$. In the rest of the chapter, we assume every G-record algebra is complete.

Proposition 4.3 *If R is a G-record algebra then the following hold in R, where a, $b \in R$.*

(1) $a + b \simeq a \sqcup b$.

(2) $a \sqcup b{\downarrow} \Longrightarrow a{\downarrow} \wedge b{\downarrow}$.

Proof (1) Suppose first $a + b{\downarrow}$. We show $a \sqcup b{\downarrow}$ and $a + b = a \sqcup b$. As $a + b{\downarrow}$ it follows that $a \leq a + b$ and $b \leq a + b$. Suppose $x \in R$ and $a \leq x$ and $b \leq x$. Then by definition of $\leq$, it follows that $x = a + x$ and $x = b + x$, whence

$$x = x + x = (a + x) + (b + x) = a + b + x$$

i.e., $a + b \leq x$. Therefore by definition of $a \sqcup b$, it follows $a + b = a \sqcup b$.

In the second case suppose $a \sqcup b{\downarrow}$. By definition of $\sqcup$, it follows that $a \leq a \sqcup b$ and $b \leq a \sqcup b$. So by definition of $\leq$, it follows that $a + a' = a \sqcup b$ and $b + b' = a \sqcup b$ for some $a', b' \in R$. As $a \sqcup b{\downarrow}$ and $a \sqcup b = a \sqcup b + a \sqcup b$, it follows that $(a + a') + (b + b'){\downarrow}$. By the commutativity law, it follows that $(a + b) + (a' + b'){\downarrow}$. Hence, by the partiality axiom for $+$, it follows that $a + b{\downarrow}$. Therefore, from the first case above, we get $a + b = a \sqcup b$.

(2) Suppose $a \bigsqcup b$. Then it follows from (1) that $a + b{\downarrow}$. By the partiality axiom of $+$ we get $a{\downarrow}$ and $b{\downarrow}$. □

Proposition 4.4 $\alpha(a \sqcup b) \simeq \alpha a \sqcup \alpha b$.

Proof Applying the left action α on both sides of $a \sqcup b \simeq a + b$, $\alpha(a \sqcup b) \simeq \alpha a \sqcup \alpha b$ follows the distributive law between α and $+$. □

Proposition 4.5 *Let R be a complete G-record algebra. Then if $S, S' \subseteq R$ then the following holds:* $\bigsqcup S + \bigsqcup S' \simeq \bigsqcup S \cup S'$.

Proof Clearly $\bigsqcup S + \bigsqcup S'{\downarrow} \iff \bigsqcup S \cup S'{\downarrow}$ from the partiality hypothesis on $\bigsqcup$ above. So we can assume that both of them have values. As $S \subseteq S \cup S'$ we get $\bigsqcup S \leq \bigsqcup S \cup S'$. Similarly, we get $\bigsqcup S' \leq \bigsqcup S \cup S'$. Hence, we get $\bigsqcup S + \bigsqcup S' \leq \bigsqcup S \cup S'$. For the converse, as $u \leq \bigsqcup S + \bigsqcup S'$ for any element u in $S \cup S'$, we get $\bigsqcup S \cup S' \leq \bigsqcup S + \bigsqcup S'$. □

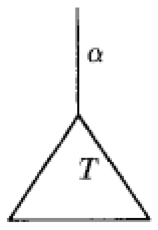

Figure 4.1: The left action by α on T: αT

Proposition 4.6 *Let S be a consistent subset of a (complete) G-record algebra R. If $S = \bigcup\{S_\lambda \mid \lambda \in \Lambda\}$ then $\bigsqcup S = \bigsqcup\{\bigsqcup S_\lambda \mid \lambda \in \Lambda\}$.*

Proof As $S_\lambda \subseteq S$, we get $\bigsqcup S_\lambda \leq \bigsqcup S$. Hence $\bigsqcup\{\bigsqcup S_\lambda \mid \lambda \in \Lambda\} \leq \bigsqcup S$. For the converse, as every $x \in S$ is an element of S_{λ_0} for some λ_0, we get $x \leq \bigsqcup S_{\lambda_0}$. Hence, we get $x \leq \bigsqcup\{\bigsqcup S_\lambda \mid \lambda \in \Lambda\}$. Therefore $\bigsqcup S \leq \bigsqcup\{\bigsqcup S_\lambda \mid \lambda \in \Lambda\}$. □

4.2.2 Partially Tagged Trees

In this section, we introduce a domain of *partially tagged trees* (PTT) as a canonical record algebra. A PTT is a kind of unordered possibly non-well-founded trees which is tagged only at some of leaf nodes. The set of PTTs will be characterized to be a complete free record algebra with some additional conditions.

Let M be a set of *tags*. Substructure of tags are left unanalysed. Let G be a feature monoid with the unit ε.

Definition 4.12 A *tree T over G* is a non empty subset of G which is closed under prefixes, i.e., if $\alpha\beta \in T$ then $\alpha \in T$. □

The singleton $\{\varepsilon\}$ is the *unit tree*. By definition, the unit tree is the minimum tree over G. As we have assumed that every feature monoid is a tree, the set $\{y \in T \mid y \leq x\}$ is totally ordered with $\leq$ for any $x \in T$. Clearly, if T_1 and T_2 are trees then $T_1 \cup T_2$ is a tree. Also $T_1 \cap T_2$ is a tree. Moreover, the class of trees over G is closed under both of the set-theoretical union and intersection of an arbitrary family of trees.

Let α and T be a feature and tree, respectively. By αT, we mean the smallest tree which has all features $\alpha\alpha'$ for any $\alpha' \in T$. (See Figure 4.1.) For instance, let $G = L^*$ and $T = \{\varepsilon, \langle b\rangle, \langle c\rangle, \langle c, d\rangle\}$ with $L = \{a, b, c, d\}$. Then $aT = \{\varepsilon, \langle a\rangle, \langle a, b\rangle, \langle a, c\rangle, \langle a, c, d\rangle\}$.

Let S be a set of features. By $S/\!\!/\alpha$, we mean the maximum tree, S' such that $\alpha S' \subseteq S$. By definition of an tree $S/\!\!/\alpha$ is a tree whenever it exists. For a tree S $S/\!\!/\alpha$ is a *subtree* of S. (See Figure 4.2).

β is called a *direct successor* of α in T if $\alpha \leq \beta$, $\alpha \neq \beta$ and there is no $\gamma \in T$ such that $\alpha < \gamma < \beta$. A feature as a node of a tree may have infinite number of direct successors. A feature α of a tree T is a *leaf* if α has no successor in T. We write *leaf*(T) for the set of leaves. A *tag function* is a *partial* function from *leaf*(T) into M. In particular, the empty function $\emptyset$ is a tag function.

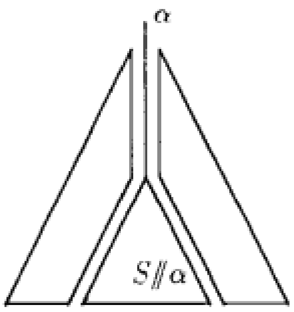

Figure 4.2: $S/\!\!/\alpha$.

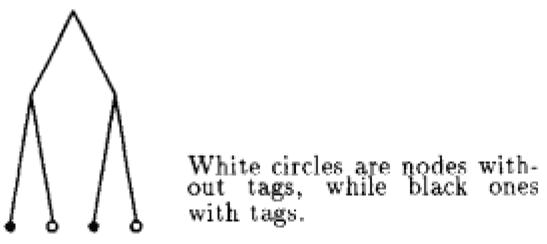
White circles are nodes without tags, while black ones with tags.

Figure 4.3: A partially tagged tree.

Definition 4.13 A *partially tagged tree* (PTT) is an ordered pair (T, f) of a tree T and a tag function of T. (See Figure 4.3.) □

The *unit* PTT, denoted by ϵ, is the ordered pair of the unit tree and the empty function $\emptyset$:

$$\epsilon = (\{\varepsilon\}, \emptyset).$$

Given a PTT $t = (T, f)$, $v \in M$ is the *tag at* α in t if $f(\alpha) = v$. We call PTTs a (G, M)-PTT when the feature monoid G and the set M of tags should be explicit. Now, for a feature α, we define left actions αt $(= \alpha \cdot t)$ and right actions $t/\!\!/\alpha$ also on PTTs. Let $t = (T, f)$ be a PTT. For a feature α, we define $\alpha t \overset{\text{def}}{=} (\alpha T, f^\alpha)$, where f^α is a tag function of αT such that $dom(f^\alpha) = \{\alpha\gamma \mid \gamma \in dom(f)\}$ and $f^\alpha(\alpha\gamma) = f(\gamma)$ for $\gamma \in dom(f)$. For features α in T, the right (partial) action is defined by the following equations:

$$t/\!\!/\alpha \overset{\text{def}}{=} (T/\!\!/\alpha, f_\alpha)$$

where f_α is a tag function of $T/\!\!/\alpha$ defined by $f_\alpha(\beta) = f(\alpha\beta)$.

Let $t_1 = (T_1, f_1)$ and $t_2 = (T_2, f_2)$ be two PTTs. Then $t = (T_1 \cup T_2, f_1 \cup f_2)$ is the *merge* of t_1 and t_2, written $t_1 + t_2$, iff $f_1 \cup f_2$ is a tag function of the tree $T_1 \cup T_2$. By this definition, $t_1 + t_2$ is undefined if there is some $\alpha \in T_1 \cap T_2$ such that $f_1(\alpha)$ and $f_2(\alpha)$ are defined but are not the same. Also $t_1 + t_2$ is undefined if there is some β such that $f_1(\beta)$ is defined but β is not a leaf of T_2.

A set of PTTs is *consistent* if any two PTTs in the set have the merge. Let $t_1 = (T_1, f_1)$ and $t_2 = (T_2, f_2)$ be PTTs. By $t_1 \leq t_2$, we means that $T_1 \subseteq T_2$ and $f_1 \subseteq f_2$. Let R be the set of (G, M)-PTTs. As the set union is complete, it is easily checked that the partial order structure $(R, \leq)$ is complete. Also it is proved without difficulty that the left and right actions are commutative with the operation of taking the supremum. M can be embedded into R by identifying each element a of M with the singleton PTT whose unique tag is a. We denote this embedding by $\varphi\colon M \to R$. We show that the set of (G, M)-PTTs is characterized as the most universal complete G-record algebra over M.

Proposition 4.7 *Let M and G be a merge system and a feature monoid respectively. Then there exists a unique (R, φ) such that the following hold.*

(1) R is a complete and standard G-record algebra over (M, φ).

(2) For any record algebra R' and a homomorphism $\varphi' : M \to R'$, there exists a complete G-homomorphism f from R into R' such that $f \circ \varphi = \varphi'$, where $\circ$ is the function composition operator.

Proof Let M be a merge system. Let R be the set of (G, M)-PTTs, and φ an injection from M into R such that $\varphi(x)$ is the *singleton* PTT whose tag is x, i.e., $\varphi(x) = \{(\{\varepsilon\}, \{(\varepsilon, x)\})\}$. It is clear that R is a complete and standard G-record algebra over (M, φ). As R is standard, the function f' from $G[\varphi(M)]$ into $G[\varphi'(M)]$ which assigns $\alpha\varphi'(x)$ to $\alpha\varphi(x)$ maps consistent subsets of R to those of R'. Hence, f' determines a complete G-record homomorphism f from R into R'. It is easy to see that $f \circ \varphi = \varphi'$.

The uniqueness is a routine. □ (R, φ) in the above proposition is called a *free* complete G-record record algebra over M.

Corollary 4.8 *Let R be the set of (G, M)-PTTs and let φ be the embedding injection $\varphi\colon M \to R$. Then the 6-tuple $(R, G, +, \cdot, /\!\!/, \epsilon)$ is a complete and standard G-record algebra over M with the injection $\varphi\colon M \to R$.*

Example 4.4 We show a non trivial record algebra on which features operate totally from both sides. Let G be a feature monoid and A a non-empty set which has sufficient number of elements. A^G is the set of functions from G into A. For each $f \in A^G$ $f /\!\!/ \alpha$ is the function f' in A^G defined by $f'(\beta) \overset{\text{def}}{=} f(\alpha\beta)$ for each $\beta \in G$. We define $t /\!\!/ \alpha \overset{\text{def}}{=} \{f /\!\!/ \alpha \mid f \in t\}$ for each $t \subseteq A^G$. Also for each $t \subseteq A^G$, we define $\alpha \cdot t$ $(= \alpha t)$ to be the largest set $t' \subseteq A^G$ such that $t' /\!\!/ \alpha \subseteq t$.

Let $\mathcal{R} = (pow(A^G), G, \cap, \cdot, /\!\!/, A^G)$. Then it is clear that $\mathcal{R}$ is a complete G-record algebra. Note that $t /\!\!/ \alpha$, αt, and $t \cup t'$ are defined for all $t, t' \subseteq A^G$, $\alpha \in G$, i.e., $\mathcal{R}$ is a totally defined algebra. □

Example 4.5 Let $G, A, \mathcal{R}$ be the same as in example 4.4. For some indexing set Π, we construct a family $\{\mathcal{R}_M\}_{M \in \Pi}$ of complete G-record *sub*algebras $\mathcal{R}_M$ of $\mathcal{R}$ *over* M. In fact, Π is the set of sets $M \subseteq pow(A^G)$ such that

(1) $\emptyset \notin M$.

(2) If $x \in \bigcup M$, $\alpha \in G$, and $\alpha \neq \varepsilon$ then $x /\!\!/ \alpha \notin \bigcup M$.

(3) If $t, t' \in M$ and $t \neq t'$ then $t \cap t' = \emptyset$.

We show that $\Pi \neq \emptyset$. Let S be the set of $f \in A^G$ such that the image of f is $\{a, b\} \subseteq A$ for some distinct $a, b \in A$ and $f^{-1}(a) = \{\varepsilon\}$. Let $M_S = \{\{f\} \mid f \in S\}$. As A has at least two elements, it follows that $S \neq \emptyset$ and hence, $M_S \in \Pi$. Therefore $\Pi \neq \emptyset$. For each $M \in \Pi$, let $\mathcal{R}_M = (R_M, G, \cap, \cdot, /\!\!/, \epsilon_M)$ be the least complete G-record subalgebra of $\mathcal{R}$ such that $M \subseteq R_M$. It is clear that $\mathcal{R}_M$ exists and is a totally defined record algebra. As ϵ_M is the unit of $\mathcal{R}_M$, it follows that $\epsilon_M \cap t = t$ for all $t \in R_M$ and hence $\epsilon_M = \bigcup R_M$. It is clear from the construction of $\mathcal{R}_M$ that in $\mathcal{R}_M$ the following hold.

(1) If $t, t' \in M$ and $t \neq t'$ then $t \cap t' = \emptyset$.

(2) If $\alpha t = \emptyset$ then $t = \emptyset$.

(3) If $t /\!\!/ \alpha = \emptyset$ then $t = \emptyset$.

(4) If $t \in M$, $u \in R_M$, and $\alpha \in G \setminus \{\varepsilon\}$ then $t \cap \alpha u = \emptyset$.

(5) If $\{\alpha_j\}_{j \in J}$ is an anti-chain in G, and $\{t_j \mid j \in J\}$ is a family of elements of R_M then $\bigcap\{\alpha_j t_j \mid j \in J\} \neq \emptyset$.

For $M \in \Pi$, it follows from these properties that by removing $\emptyset$ from R_M, we get a desired partial G-record algebra $\mathcal{R}'_M$ over M, where $\mathcal{R}'_M = (R_M \setminus \{\emptyset\}, G, \cap, \cdot, /\!\!/, \epsilon_M)$.

□

4.2.3 The Unification Theory

In this subsection, we fix M, G, R, X as follows unless mentioned otherwise explicitly: R is a complete and standard G-record algebra over a merge system $(M, +)$ and X is a set of parameters. Elements of $\mathcal{E}(G, M \cup X, \{\cdot, +, /\!\!/\})$ are called a *record term*. It is also called a *parametric record*[2]. Elements of $\mathcal{E}(G, M \cup X, \{\cdot\})$ are called a *basic record term*. It is convenient to have the left action built into the record terms. So we identify elements of $\mathcal{E}(G, M, F)$ up to the following equations:

$$\begin{aligned} (\alpha\beta)x &= \alpha(\beta x). \\ \varepsilon x &= x. \end{aligned}$$

In fact, we abuse the symbol $\mathcal{E}(G, M, F)$ for the quotient set of $\mathcal{E}(G, M, F)$ modulo the least congruence relation generated by the above three equations. This convention will be used without mentioning.

Note that we have no syntactical counter part for ϵ. Hence ϵ does not appear in any record term. In particular, ϵ is not a record term. An *atomic constraint* is an ordered pair of record terms, written

$$p \bowtie q$$

where p, q are record terms.

[2]Record terms are called a *partially specified term* (PST) in [55].

Remark As $u /\!/ \alpha \bowtie v$ is semantically reduced to $u \bowtie \alpha w$ plus $w \bowtie v$ with w being a new parameter, we assume $p, q \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$ without loss of generality. □

A *constraint* is a possibly infinite set of atomic constraints. We give a set of *constraint rules* in table 4.1. Each rule there means a condition on constraints.

Definition 4.14 A constraint C is *closed* if C satisfies all clauses in the constraint rule table 4.1. □

Definition 4.15 Recalling that R is a complete and standard G-record algebra over M, a pair of u and v in $\mathcal{E}(G, M \cup X, \{\cdot, +, /\!/\})$ is a *conflict* if one of the following hold.

(1) $u \in M$ and $v \in M$ but $u + v$ is undefined in $(M, +)$.

(2) $u \in M$ and $v = \alpha w$ for some record term w and $\alpha \in G \setminus \{\varepsilon\}$.

□

Clearly, if u and v are a conflict pair then there is no assignment such that $f(u + v)$ is defined. The set of constraint rules is designed so that every constraint C is *solvable* iff it is unifiable, i.e., there is a closed and consistent extension C' of C.

In Table 4.1, C is a constraint, $x \in X$, and $\alpha, \beta \in G$ are incomparable features appearing in C, and u, v, w are record terms appearing in C. Note that x appearing in the restricted transitive rule is a *parameter* in X.

Table 4.1: Constraint Rules

Merge (0)	$u \bowtie v \in C \wedge u, v \in M$	$\Longrightarrow$	$u + v{\downarrow}.$
Base	$\alpha u \bowtie \beta v \subset C, \alpha \not\sim \beta$	$\Longrightarrow$	$u \bowtie u \in C, v \bowtie v \in C.$
Reflexive	$u \in X \cup M$	$\Longrightarrow$	$u \bowtie u \in C.$
Symmetric	$u \bowtie v \in C$	$\Longrightarrow$	$v \bowtie u \in C.$
Restricted Transitive	$x \bowtie u \in C, x \bowtie v \in C$	$\Longrightarrow$	$u \bowtie v \in C.$
Merge (1)	$(u + v) \bowtie w \in C$	$\Longrightarrow$	$u \bowtie w \in C.$
Merge (2)	$(u + v) \bowtie w \in C$	$\Longrightarrow$	$u \bowtie v \in C.$
Cancellation	$\alpha u \bowtie \alpha v \in C$	$\Longrightarrow$	$u \bowtie v \in C.$

Definition 4.16 A *closed constraint* over R is a set of atomic constraints satisfying the rules on table 4.1. □

From the reflexive, symmetric, and restricted transitive rules on the table, every closed constraint contains an equivalence relation between parameters. However, as there is not a full transitive law, the closed constraint gives no equivalence relation on $\mathcal{E}(G, M \cup X, \{\cdot, +, /\!/\})$ in general. The minimum closed extension is called the *closure* of S. Clearly the closure of a finite constraint S is computed effectively. A *unification problem* is to find the closure. The input of the unification algorithm is a finite set C of atomic constraints:

$$C = \{p_1 \bowtie q_1, \ldots, q_n \bowtie q_n\}.$$

The output is the consistent closure $\overline{C}$ or 'conflict' when $\overline{C}$ has a conflict.

In the following example, $G = \{a, b\}^*$ $(a \neq b)$ and $M = \{1, 2\}$ is a trivial merge system. That is, 1+2 is undefined and $1 + 1 = 1$, $2 + 2 = 2$. Let R be a G-record algebra over M. Let $X = \{x, y\}$ be a set of parameters. For saving the space, we omit obvious atomic constraints obtained by, for instance, the reflexive rule or the base rule. Also we use the symmetric rule implicitly.

Example 4.6 Let $C_1 \stackrel{\text{def}}{=} \{ax + bx \bowtie by + a1\}$. By merge rule (1), we get $ax \bowtie a1$ and $bx \bowtie by$. Applying the cancellation rule to the two, we get $x \bowtie 1$ and $x \bowtie y$. Applying the restricted transitive rule, we get $y \bowtie 1$. Thus the closure of C_1 is

$$\{ax + bx \bowtie by + a1, ax \bowtie a1, bx \bowtie by, x \bowtie 1, x \bowtie y, y \bowtie 1\}.$$

There is no conflict in the closure. □

Example 4.7 The example gives a cyclic graph.

$$C_2 \stackrel{\text{def}}{=} \{x \bowtie ay + by, y \bowtie ax + bx, x \bowtie y\}.$$

Applying the restricted transitive law to parameter x, we get $y \bowtie ay + by$. It follows from this and he restricted transitive law applied to $y \bowtie ax + bx$ w.r.t. y, we get $ax + bx \bowtie ay + by$. By repeating merge law (1), we get $ax \bowtie ay$ and $bx \bowtie by$. Applying the cancellation law to each of them, we get $x \bowtie y$. Now no rule is applicable. The closure of C_2 is:

$$\{x \bowtie ay + by, y \bowtie ax + bx, x \bowtie y, ax \bowtie ay, bx \bowtie by, x \bowtie ax + bx, y \bowtie ay + by\}.$$

The output of this unification means a singleton graph which has two self-loops with features a and b. □

4.2.4 Satisfiability of Record Constraints

M, X, G, R are the same in the previous subsection. Recall the definition of $\mathcal{E}$ (definition 4.2).

Definition 4.17 We define a function $\pi: \mathcal{E}(G, X \cup M, \{\cdot, +\}) \to pow(\mathcal{E}(G, X \cup M, \{\cdot\}))$ inductively by the following equations:

$$\begin{array}{rcll} \pi(x) & = & \{x\} & \text{if } x \in M \cup X. \\ \pi(x + y) & = & \pi(x) \cup \pi(y). & \\ \pi(\alpha x) & = & \{\alpha y \mid y \in \pi(x)\}. & \end{array}$$

□

Example 4.8 If $a, b, c \in G$ and $x, y, z \in X$ then $\pi(a(bx + c(y + z))) = \{abx, acy, acz\}$. □

We abuse the π as $\pi(S) \stackrel{\text{def}}{=} \bigcup\{\pi(x) \mid x \in S\}$. An *assignment* is a partial function from X into R.

Proposition 4.9 *If f is an assignment then there is the largest partial function $h: \mathcal{E}(G, M \cup X, \{\cdot, +, /\!\!/\}) \to R$ which satisfies the following.*

$$\begin{array}{rcrcl} x \in X, h(x)\downarrow & \Longrightarrow & x & \in & dom(f). \\ x \in dom(f) & \Longrightarrow & h(x) & = & f(x). \\ x \in M & \Longrightarrow & h(x) & = & x. \\ & \Longrightarrow & h(x+y) & \simeq & h(x) + h(y). \\ & \Longrightarrow & h(\alpha x) & \simeq & \alpha h(x). \\ & \Longrightarrow & h(x /\!\!/ \alpha) & \simeq & h(x) /\!\!/ \alpha. \end{array}$$

Proof A proof is done by structural induction on record terms. □ We use the assignment f also for the extension h.

Definition 4.18 Let p be a record term in $\mathcal{E}(G, M \cup X, \{\cdot, +, /\!\!/\})$, f an assignment and t a record in R. Then t *is an instance of p with f*, written

$$t :_f p$$

if for any $\alpha x \in \pi(p)$ the following hold, where $x \in X \cup M$, $\alpha \in G$:

(1) $t /\!\!/ \alpha\downarrow$.

(2) $x \le t /\!\!/ \alpha$ if $x \in M$.

(3) $f(x) = t /\!\!/ \alpha$ if $x \in X$.

□

Clearly $t :_f p$ implies $f(p)\downarrow$.

Example 4.9 $ac + ad + bc + bd :_f ax + bx$ is a valid assertion, where $a, b \in G$, $c, d \in M$, $c + d\downarrow$, and $f(x) = c + d$. □

Definition 4.19 f is a *solution* of $p \bowtie q$ (in R) iff $t :_f p$ and $t :_f q$ for some record $t \in R$. Given a constraint C, f is a *solution* of C (in R) if f is a solution of each atomic constraint in C. □

f *solves* a constraint if f is solution of the constraint. Recall that R is standard.

Lemma 4.1 *Let $x \in X$, $\alpha, \beta \in G$, $u, v, w \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$, and f an assignment. Then the following hold.*

(1) If $u, v \in M$ and f solves $u \bowtie v$ then $u + v\downarrow$.

(2) If $u \in X$ and f solves $x \bowtie \alpha u$ then $f(u) = f(x) /\!\!/ \alpha$.

(3) If $u \in M$ and f solves $x \bowtie \alpha u$ then $u \le f(x) /\!\!/ \alpha$.

(4) If f solves $u \bowtie v$ then f solves $v \bowtie u$.

(5) If f solves $x \bowtie u$ and $x \bowtie v$ then f solves $u \bowtie v$.

(6) If $\alpha \not\prec \beta$ and f solves $\alpha u \bowtie \beta v$ then f solves both $u \bowtie u$ and $v \bowtie v$.

(7) If f solves $(u + v) \bowtie w$ then f solves $u \bowtie w$.

(8) If f solves $(u + v) \bowtie w$ then f solves $u \bowtie v$.

(9) If f solves $\alpha u \bowtie \alpha v$ then f solves $u \bowtie v$.

(10) If $u, v \notin M \cup X$ and f solves $u \bowtie v$ then there is an extension f' of f such that f' solves both $x \bowtie u$ and $x \bowtie v$ for some $x \in dom(f') \setminus dom(f)$.

Proof

(1) As f solves $u \bowtie v$, there is some $t \in R$ such that $u :_f t$ and $v :_f t$. Hence, as $u, v \in M$, we get $u \leq t$ and $v \leq t$. Hence $u + v\downarrow$.

(2) Suppose that $u \in X$ and f solves $x \bowtie \alpha u$. Then there is some $t \in R$ such that $f(x) = t /\!\!/ \varepsilon = t$ and $u = t /\!\!/ \alpha$. Hence $f(u) = f(x) /\!\!/ \alpha$.

(3) Suppose that $u \in M$ and f solves $x \bowtie \alpha u$. Then there is some $t \in R$ such that $f(x) = t /\!\!/ \varepsilon = t$ and $u \leq t /\!\!/ \alpha$. Hence $u \leq f(x) /\!\!/ \alpha$.

(4) It is obvious by definition that f solves $u \bowtie v$ iff f solves $v \bowtie u$. The case follows from this equivalence.

(5) Suppose that f solves $x \bowtie u$ and $x \bowtie v$. Then $t :_f x$, $t :_f u$, $t' :_f x$, $t' :_f v$ for some $t, t' \in R$. Hence we get $t = f(x) = t'$, whence $t :_f u$ and $t :_f v$. Hence, by definition, f solves $u \bowtie v$.

(6) Suppose $\alpha \not\prec \beta$ and f solves $\alpha u \bowtie \beta v$. Then $t :_f \alpha u$ and $t :_f \beta v$ for some $t \in R$. So $t /\!\!/ \alpha :_f u$ and $t /\!\!/ \alpha :_f v$. Hence f solves $u \bowtie u$ and $v \bowtie v$.

(7) Suppose f solves $(u + v) \bowtie w$. Then $t :_f u + v$ and $t :_f w$ for some $t \in R$. As $t :_f u + v$ implies $t :_f u$ f solves $u \bowtie w$.

(8) Suppose f solves $(u + v) \bowtie w$. Then $t :_f u + v$ for some t. As $t :_f u + v$ implies $t :_f u$ and $t :_f v$ f solves $u \bowtie v$.

(9) Suppose f solves $\alpha u \bowtie \alpha v$. Then $t :_f \alpha u$ and $t :_f \alpha v$ for some $t \in R$, whence $t /\!\!/ \alpha :_f u$ and $t /\!\!/ \alpha :_f v$. Hence f solves $u \bowtie v$.

(10) Suppose $u, v \notin M \cup X$ and f solves $u \bowtie v$. Let x be a new parameter not in $dom(f) \cup \mathcal{V}(u) \cup \mathcal{V}(v)$. Let f' be the extension of f such that $dom(f') = \{x\} \cup dom(f)$ and $f'(x) = f(u) + f(v)$. It is clear that f' is well-defined, f' solves both $x \bowtie u$ and $x \bowtie v$. Hence f' satisfies condition (10) in the proposition.

$\square$

Definition 4.20 $\models$ is the *largest* ternary relation which satisfies the following clauses, where $x \in X$, $u, v, w \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$, $\alpha, \beta \in G$:

$$\begin{array}{lrcl}
(1) & u, v \in M \wedge R, f \models u \bowtie v & \Longrightarrow & u + v\downarrow. \\
(2) & u \in X \wedge R, f \models x \bowtie \alpha u & \Longrightarrow & f(u) = f(x) /\!\!/ \alpha. \\
(3) & u \in M \wedge R, f \models x \bowtie \alpha u & \Longrightarrow & u \leq f(x) /\!\!/ \alpha. \\
(4) & R, f \models u \bowtie v & \Longrightarrow & R, f \models v \bowtie u. \\
(5) & R, f \models x \bowtie u \wedge R, f \models x \bowtie v & \Longrightarrow & R, f \models u \bowtie v. \\
(6) & \alpha \not\sim \beta \wedge R, f \models \alpha u \bowtie \beta v & \Longrightarrow & R, f \models u \bowtie u \wedge R, f \models v \bowtie v. \\
(7) & R, f \models (u + v) \bowtie w & \Longrightarrow & R, f \models u \bowtie w. \\
(8) & R, f \models (u + v) \bowtie w & \Longrightarrow & R, f \models u \bowtie v. \\
(9) & R, f \models \alpha u \bowtie \alpha v & \Longrightarrow & R, f \models u \bowtie v. \\
(10) & R, f \models u \bowtie v & \Longrightarrow & R, f \models x \bowtie u \wedge \ R, f \models x \bowtie v \quad (\exists x \in dom(f)).
\end{array}$$

□

Proposition 4.10 *The relation $\models$ exists.*

Proof Clearly $\emptyset$ satisfies all clauses in definition 4.20. Take the union of all such relations. Then it is clear that the union also satisfies the clauses. □

Remark As the class of complete and standard G-record algebras R can be a proper class even if G is fixed the relation $\models$ is a proper class relation in general. □

Proposition 4.11 *Let $p, q \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$ and f an assignment such that $\mathcal{V}(p) \cup \mathcal{V}(q) \subseteq dom(f)$. Then the following are equivalent.*

(1) f is a solution of $p \bowtie q$ in R.

(2) $R, f' \models p \bowtie q$ for some extension f' of f.

Proof

(1)$\Longrightarrow$(2): Suppose f is a solution of $p \bowtie q$ in R. Then there is the consistent closure C of $p \bowtie q$. Let $C' = \{u \bowtie v \in C \mid u, v \notin X \cup M\}$ and x_c be a new and unique parameter for each $c \in C'$. Let f' be the extension of f such that $f(x_{u \bowtie v}) = f(u) + f(v)$ for $u \bowtie v \in C'$. Let C'' be the consistent closure of $C \cup \bigcup\{\{x_c \bowtie u, x_c \bowtie v\} \mid c = (u \bowtie v), c \in C'\}$ and finally let $D = \{(R, f', c) \mid c \in C''\}$. From lemma 4.1, D as a ternary relation satisfies all defining clauses for $\models$ in definition 4.20. As $\models$ is the largest such relation, we get $D \subseteq \models$.

(2)$\Longrightarrow$(1): Suppose $R, g \models p \bowtie q$. If $p, q \in M \cup X$ then it is clear by definition of a solution and $\models$ that f solves $p \bowtie q$. Otherwise, by condition (10) in definition 4.20, there is a parameter $x \in dom(g)$ such that $R, g \models x \bowtie p$ and $R, g \models x \bowtie q$. Then it follows from conditions in definition 4.20 that for any $u \in \pi(p) \cup \pi(q)$, we have $R, g \models x \bowtie u$. As x is a parameter and u is a basic record term, it follows from the definition of $\models$ that $g(x) :_g u$. Hence g solves $p \bowtie q$ in R. □

Example 4.10 Let $x \in X$ be a parameter. We show that the constraint

$$ad1 + bd2 \bowtie ax + bx$$

has no solution, where 1 and 2 are used as distinct atoms. To see this, suppose that a solution f exists. Then by definition of a solution, there exists $t \in R$ such that $t :_f \{ax + br\}$ and $t :_f \{ad1 + ad2\}$. Then by definition of an instance we have $t/\!\!/a = f(x)$ and $t/\!\!/b = f(y)$. Also $ad1 \leq t$ follows from $t :_f \{ad1 + ad2\}$. Hence $d1 \leq t/\!\!/a = f(x)$. So we get $1 \leq f(x)/\!\!/d$. Similarly we get $2 \leq f(x)/\!\!/d$. This is a contradiction, because the sum $1 + 2$ of distinct tags 1 and 2 was assumed to be undefined.

However note that the constraint $ad1 + bd2 \bowtie ac3 + bc3$, which is obtained by applying substitution $x \mapsto c3$, is true in the record algebra R. So this example explains why a restricted notion of a solution is necessary for an equivalence between the satisfiability and unifiability in our constraint language. □

Definition 4.21 A record $r \in R$ is an *initial segment* of a basic record term u if one of the following hold.

- $r = \alpha\epsilon$ and $u = \alpha v$, where $v \in \mathcal{E}(G, M \cup X, \{\cdot\})$, $\alpha \in G$.
- $r = u = \alpha c$ where $\alpha \in G$, $c \in M$.

□

Theorem 4.12 (Record Solution Theorem) *Every consistent closed constraint has a solution.*

Proof Given a consistent closed constraint S, let S' be a minimal constraint such that the following hold.

- $S \subseteq S'$.
- S' is closed under the constraint rules on table 4.1.
- If $x \bowtie \alpha y \in S'$ and $y \bowtie \beta z \in S'$ then $x \bowtie \alpha\beta z \in S'$, where $\alpha, \beta \in G$, $x, y, z \in M \cup X$.

Choose a unique new parameter, say $x_{p\bowtie q}$, for each $p \bowtie q \in S'$ such that $p, q \notin M \cup X$. Let C be the reflexive closure of $S' \cup \bigcup\{\{x_{p\bowtie q} \bowtie p, x_{p\bowtie q} \bowtie q\} \mid p \bowtie q \in S', p, q \notin M \cup X\}$. It is clear that C has the same set of solutions as S. C may be infinite even when S is finite. Anyway it suffices to show that C is satisfiable. Let $B = \{u \bowtie v \mid p \bowtie q \in C, u, v \in \pi(u) \cup \pi(v)\}$. Clearly $B \subseteq C$. We first construct a solution of B. For $x \in M \cup X$, let D_x be the set of initial segments of some basic record term u such that $x \bowtie u \in C$. As C is consistent and closed, D_x has no conflict. We make a convention that $D_x = \{\epsilon\}$ if D_x is empty.

We consider atomic constraints in C of the form $\alpha x \bowtie y$, where $x, y \in M \cup X$. If $x, y \in M$ then α must be ε and $x + y\downarrow$ because C is consistent. If $x \in X$ and $y \in M$ then also α must be ε. However this is a special case of the following by changing the role of x and y.

So finally we suppose $y \bowtie \alpha x$ with $y \in X$, $x \in M \cup X$. Suppose first $x \in X$. Now we show that $D_x = D_y/\!\!/\alpha$, where $D/\!\!/\alpha$ denotes the set $\{w/\!\!/\alpha \mid w \in D, w/\!\!/\alpha\downarrow\}$. Suppose $u \in D_x$. By definition of D_x there is some u' such that $x \bowtie u' \in C$ and u is an initial segment of u'. As $y \bowtie \alpha x \in C$, $x \bowtie u' \in C$ and C is closed under 'unfolding', $y \bowtie \alpha u'$ must be in C. Hence, again by definition of the D_x, we get $u \in D_y/\!\!/\alpha$. For the converse, suppose $u \in D_y/\!\!/\alpha$. Then

$y \bowtie \alpha u' \in C$ for some u' such that αu is an initial segment of $\alpha u'$. So u is an initial segment of u'. As $y \bowtie \alpha x \in C$ after a sequence of several steps of applying constraint rules, we have $x \bowtie u' \in C$. Thus $u \in D_x$. Hence, we get $D_x = D_y /\!\!/ \alpha$. Let f be the assignment defined by putting $f(z) = \bigsqcup D_z$ for any parameters z of S. As R is a complete G-record algebra over M we get $f(x) = f(y) /\!\!/ \alpha$.

In the second case, suppose $x \in M$. As $\alpha x \in D_y$, we get $\alpha x \leq \bigsqcup D_y = f(y)$ and hence, $x \leq f(y) /\!\!/ \alpha$. Therefore f is a solution of B.

We show that f is a solution of C. Using the fact that f is a solution of B, it is a routine to check that the set $\{(R, f, c) \mid c \in C\}$ satisfies all defining clauses of $\models$ in definition 4.20. As $\models$ is the largest such ternary relation, we get $\{(R, f, c) \mid c \in C\} \subseteq \models$. Hence, by proposition 4.11, f is a solution of C in R. □

As an obvious corollary of this theorem, we get the equivalence between the unifiability and the satisfiability.

Theorem 4.13 (Unification Theorem) *Let $p, q \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$. Then the following are equivalent.*

(1) $p \bowtie q$ is unifiable, i.e., has a consistent closure.

(2) $p \bowtie q$ is satisfiable in R.

Let S be a constraint. It is easy to see that the closure of S is the union of the closures of all finite sets of S. The *compactness theorem* follows directly from this:

Theorem 4.14 (Compactness Theorem) *Let C be a set of constraints. Then C has a solution iff every finite subset of C has a solution.*

4.3 Unification Grammar over Records

In this section, we use R, X, $(M, +)$ for a complete and standard G-record algebra over M, a set of parameters, and a merge system, respectively.

4.3.1 Semantics of the Program

A *program clause* over $\mathcal{E}(G, M \cup X, \{\cdot, +\})$ is an ordered pair (p, B) of a record term p and a finite set B of record terms. A *program* P is a finite set of program clauses. A *goal* is a non-empty finite set of record terms. A program clause $(p, \{p_1, ..., p_n\})$ $(n \geq 0)$ is written

$$p \leftarrow p_1, ..., p_n.$$

In the case $n = 0$, we write

$$p.$$

for the program clause $(p, \emptyset)$ as usual. A program P is fixed throughout this section.

Example 4.11 The program P_0 below consists of three Horn clauses for a recursive data type definition for list structures. L is a set of atomic features. Let $\mathcal{R} = (R, G, +, \cdot, /\!\!/, \epsilon)$ be a record algebra over M, where $G = L^*$ and M is a trivial merge system. In the program, we assume that $x, l \in X$, $a, b, nil, atom, list \in M$, and $type, car, cdr, form \in L$.

(1) $type\ atom + form\ a.$

(2) $type\ atom + form\ b.$

(3) $type\ list + form\ (car\ x + cdr\ l) \leftarrow$
$type\ atom + form\ x,\ type\ list + form\ l.$

□

Definition 4.22 [**Model**] A subset M of R is a *model* of the given program P if for each $t \in M$ there exists some program clause $p \leftarrow p_1, \ldots, p_n$ and assignment f such that the following hold:

(1) $t :_f p$.

(2) For each $1 \leq i \leq n$ there exists $t_i \in M$ such that $t_i :_f p_i$.

□

Clearly, $\emptyset$ is a model of any program. Also M and M' are models then $M \cup M'$ is a model. Hence, there exists the maximum model of P.

Definition 4.23 The *semantics* $\mathcal{M}_P$ *of the program* P is the maximum model of the program P. □

Definition 4.24 We define a transformation $\Phi_P : pow(R) \to pow(R)$. Given a $Q \subseteq R$, $\Phi_P(Q)$ is the set of records $t \in R$ such that $t :_f p$ for some program clause $(p, B) \in P$ and assignment f such that for any $q \in B$ there is $t' \in Q$ such that $t' :_f q$. □

Example 4.12 Let $S \subseteq R$. Then $\Phi_{P_0}(S) = \{type\ atom + form\ a. type\ atom + form\ b\} \cup S'$, where S' is the set of records $t \in R$ such that the following hold for some $s \in S$ and an assignment f.

(1) $t :_f type\ list + form\ car\ x + form\ cdr\ l.$

(2) $s :_f type\ atom + form\ x.$

(3) $s :_f type\ list + form\ l.$

$\mathcal{M}_{P_0}$ is the largest fixpoint of Φ_{P_0}. □

As $(pow(R), \subseteq)$ is a complete partial ordered structure and Φ_P is a monotone function w.r.t. the order, it follows from the standard theorem that there is a maximum fixpoint of Φ_P. Also it is a routine to show that the maximum fixpoint is the maximum semantics $\mathcal{M}_P$. For more details, the reader is referred to Aczel [1], in which an existence theorem of the minimum and maximum fixpoints of set-based class functors is given in more general setting.

Definition 4.25 An assignment f is an *answer solution* of a goal G in $\mathcal{M}_P$ if for each $q \in G$ there exists $t_q \in \mathcal{M}_P$ such that $t_q :_f q$. □

We make a convention. Let $(D, \leq)$ be a partial order structure and let f and f' be partial functions from some set into D. f' is an *extension* of f if $dom(f) \subseteq dom(f')$ and $f(x) \leq f'(x)$ for any $x \in dom(f)$.

Definition 4.26 A *support* is a consistent and closed constraint. □

Definition 4.27 A *computation state* (state for short) is a pair (Q, E) of a goal Q and support E. □

Definition 4.28 A *resolution step* is an ordered pair (s, s') written as

$$s \rightarrow s'$$

of two states $s = (Q, E)$ and $s' = (Q', E')$, where $Q = \{p_1, ..., p_n\}$, satisfying the following:

(1) There exist copies $q_i \leftarrow q_1^i, ..., q_{k_i}^i$ $(1 \leq i \leq n)$ of program clauses such that

$$Q' = \{q_1^1, ..., q_{k_1}^1, ..., q_1^n, ..., q_{k_n}^n\}.$$

(2) E' is the consistent closure of $\{p_1 \bowtie q_1, ..., p_n \bowtie q_n\} \cup E$. □

The constraint $\{p_1 \bowtie q_1, ..., p_n \bowtie q_n\}$ above is called the *constraint associated with* the resolution step. A *computation* is a finite or countable sequence of states such that if s is the successor of s' then $s \rightarrow s'$. A *success computation* is a computation Γ such that Γ is a countably infinite one or the goal component of the last state of Γ is empty. A *failure computation* is a computation which is not a success one.

4.3.2 Soundness and Completeness

Given a computation Γ, the supports appearing in Γ form a monotone increasing sequence. So the union of these supports is a support, which we call *the support of* Γ. The support of a finite computation Γ is the support at the last state of the computation Γ. A *computation for a goal* Q is a computation which starts from the state $(Q, \emptyset)$.

Theorem 4.15 (Soundness Theorem) *Let Γ be a success computation for Q and E the support of Γ. Then E is solvable and every solution of E is an answer solution of Q.*

Proof Suppose Γ is a success computation for Q and let E be the support of Γ. As E is consistent and closed by the unification theorem 4.13, there exists a solution f of E. For every element p of Q, it follows from the definitions of $\mathcal{M}_P$ and the computation that $f(p) \in \mathcal{M}_P$. Hence, as $p \in Q$ is arbitrary, f is an answer solution of Q. □

Lemma 4.2 (Resolution Step Lemma) *If an assignment f is a solution of E and an answer solution of Q $(\neq \emptyset)$ in $\mathcal{M}_P$, then there exist an extension f' of f and a resolution step $(Q, E) \rightarrow (Q', E')$ such that f' is a solution of E' and an answer solution of Q'.*

Proof Let $p \in Q$. As f is an answer solution of p in $\mathcal{M}_P$, by definition of $\mathcal{M}_P$, there exists an extension f'_p of f and a fresh copy (h_p, D_p) of a program clause such that f'_p is an solution of $p \bowtie h_p$ and an answer solution of D_p. Let $Q' = \bigcup\{D_p \mid p \in Q\}$ and let E' be the closure of $C \cup E$, where $C = \{p \bowtie h_p \mid p \in Q\}$. As (h_p, D_p) is a fresh copy, the family $\{f'_p\}_{p \in Q}$ is compatible as functions. Hence, there is an extension of f which satisfies both C and E. Therefore, by the unification theorem 4.13, E' is consistent and closed. By definition 4.38 of the resolution step, we get finally $(Q, E) \rightarrow (Q', E')$. □

The following completeness theorem is obtained by repeating the resolution step lemma 4.2.

Theorem 4.16 (Completeness Theorem) *If f is an answer solution of a goal Q then f extends to a solution of the support of some success computation for Q.*

In the rest of this section, we assume that $dom(f)$ of assignments f is large enough to include all necessary parameters for evaluating expressions in the context. By $sol(D)$, we mean the set of answer solutions of D.

Lemma 4.3 (Lifting Lemma) *Let $(Q, E) \rightarrow (Q', E')$ be a resolution step. Let F be a support such that $sol(E) \subseteq sol(F)$. Then there exist a support F' and a resolution step $(Q, F) \rightarrow (Q', F')$ such that $sol(E') \subseteq sol(F')$.*

Proof Let C be the constraint associated with the resolution step $(Q, E) \rightarrow (Q', E')$. By definition of an resolution step, E' is the consistent closure of $C \cup E$. Let F' be the closure of $C \cup F$. Clearly, $sol(E') \subseteq sol(F')$ follows from $sol(E) \subseteq sol(F)$. Hence, as $sol(E')$ is not empty, $sol(F')$ is not empty. Therefore, F' is a support. As (Q, F) and (Q', F') satisfy all defining clauses in definition 4.38 of a resolution step, we get $(Q, F) \rightarrow (Q', F')$. □

Let Γ be a success computation from the state (Q, E). By repeating application of the lifting lemma, we have a success computation Γ' starting from the state $(Q, \emptyset)$. Let (H, D) and (H, D') be corresponding states on the two computations Γ and Γ'. By induction on the number of resolution steps from the initial state, it is proved that D is the closure of $E \cup D'$. We call Γ' the *lifting* of Γ.

Definition 4.29 A parameter x is *free* in a support E if $x \bowtie u \in E$ implies $u = x$. □

For example, in the constraint $\{x \bowtie x, z \bowtie ay, u \bowtie v\}$ x, y are free but z, u, v are not free. The following theorem is a counter part of the theorem in Lloyd [46] having the same title.

Theorem 4.17 (Display Theorem) *Let E be a support such that every solution of E is an answer solution of a goal Q. Let $\mathcal{F}(E)$ be the set of free parameters appearing in E. Then there exists a success computation $\Gamma_\emptyset$ from $(Q, \emptyset)$ such that the restriction of each solution of E to $\mathcal{F}(E)$ can extends to a solution of the support of the computation.*

Proof We assume for the sake of simplicity that there are sufficiently many constants. For each $x \in \mathcal{F}(E)$, let c_x be a new constant such that $c_x \neq c_y (x \neq y)$. Let C be the reflexive closure of $\{x \bowtie c_x \mid x \in M\}$. From the assumption $E' = C \cup E$ is a support. Hence, by the completeness theorem, there exists a success computation $\Gamma_{E'}$ from (Q, E'). By the lifting lemma, there exists a success computation Γ_E from (Q, E). Applying the lifting lemma again to $\Gamma_{E'}$, we obtain a success computation $\Gamma_\emptyset$ from $(Q, \emptyset)$. We use Σ_Δ for the support of a computation Δ. By the remark above, we can construct Γ_E and $\Gamma_\emptyset$ so that Σ_{Γ_E} is the closure of $E \cup \Sigma_{\Gamma_\emptyset}$ and $\Sigma_{\Gamma_{E'}}$ is the closure of $C \cup \Sigma_{\Gamma_E}$.

$$\begin{array}{rlll} \Gamma_\emptyset : & (Q, \emptyset) & \rightarrow & \cdots \\ \Gamma_E : & (Q, E) & \rightarrow & \cdots \\ \Gamma_{E'} : & (Q, E') & \rightarrow & \cdots \end{array}$$

As $E \cup \Sigma_{\Gamma_\emptyset}$ has the closure Σ_{Γ_E} for the proof of the theorem, it suffices to show that each parameter in $\mathcal{F}(E)$ is free in $\Sigma_{\Gamma_\emptyset}$. Hence, it is sufficient to show that the closure of $C \cup \Sigma_{\Gamma_\emptyset}$ is a support. In fact, the closure of $C \cup \Sigma_{\Gamma_\emptyset}$ must be a support because the closure of $C \cup E \cup \Sigma_{\Gamma_\emptyset}$ is the support $E_{\Gamma_{E'}}$. □

Theorem 4.18 (Soundness of NAF) *If there is no success computation from $(Q, \emptyset)$ then Q has no answer solution.*

Proof This is the contraposition of the soundness theorem 4.15 □

Theorem 4.19 (Completeness of NAF) *If Q has no answer solution then there is no success computation from $(Q, \emptyset)$.*

Proof This is the contraposition of the completeness theorem 4.16. □

Due to the maximum semantics the proof of soundness and completeness of negation-as-failure rule has become almost obvious. Infinite computations is always meaningful in $\mathcal{M}_P$, while in the least Herbrand model infinite computations are meaningless.

4.3.3 DAGs as Constraints on Records

In this subsection, we show a relationship between DAGs (directed acyclic graphs) used in unification grammars [84] and record structures. This is done by giving a simple translation from DAGs into constraints on records. In stead of DAGs, we treat directed graphs (DGs) as a more general class than the DAG class.

Let X, L, M be a set of *nodes*, *atomic features*, *tags*, respectively. Let R be a free complete and standard G-record algebra over M, where $G \overset{\text{def}}{=} L^*$. We assume without loss of generality that R and X are disjoint to each other.

Definition 4.30 A directed graph (DG) D is a 5-tuple $D = (N, A, f, g, s)$, where $N \subseteq X$, $A \subseteq N \times N$, f is a partial function from N into M, g is a function from A to L, and s is the *root node.* □

Given a DG, $D = (N, A, f, g, s)$, let C_D be the least set C such that the following hold:

(1) If $(x, y) \in A$ and $g((x, y)) = a$ then $x \bowtie ay \in C$.

(2) If $f(x) = c$ then $x \bowtie c \in C$.

Viewing C_D as a binary relation on $\mathcal{E}(G, M \cup X, \{\cdot, +\})$, we take C_D as a constraint on G-record algebra R over M. Thus, the DG D is a constraint over the record algebra R. Let $D_i = (N_i, A_i, f_i, g_i, s_i)$ $(i = 1, 2)$ be two DGs. The graph merge of D_1 and D_2 is the union of the constraint $\{s_1 \bowtie s_2\} \cup C_{D_1} \cup C_{D_2}$. Thus, through this translation, it is straightforward to give the proposed record algebra semantics to the unification grammar. Our semantics covers some basic part of DAG-based unification grammar theory in Shieber [83].

$$\begin{tikzcd}
 & x \arrow[dl, "a"'] \arrow[ddr, "c"] & & & \\
y \arrow[drr, "b"'] & & \Rightarrow & & x \bowtie ay \\
 & & d & & x \bowtie cd \\
 & & & & y \bowtie bd
\end{tikzcd}$$

Figure 4.4: A DAG as a constraint on records

The notion of *structure sharing* in DAG-based theory corresponds to that of sharing parameters in constraint language $(R, \bowtie)$. From the view point of this translation, the notion of structure sharing belongs only to the constraint language. The structure sharing is not a property of objects but just occurrences of the same parameters.

As records can be infinite or non-well-founded, they can represent more complex structures than (finite) DAGs can do. In particular, the record domain seems to be suitable for representing and processing circular situations proposed by Barwise and Etchemendy [16] in addition to the ordinary linguistic information processing.

4.3.4 Arity in Record Algebra

In this subsection we show an embedding of a (complete) Herbrand domain into a record algebra. This embedding allows us to use nested structures consisting of record terms and standard terms in a uniform way. Let G be a feature monoid. A *sort* (of G) is a prefix-closed subset S of G, i.e., if $\alpha\beta \in S$ then $\alpha \in S$. The *arity* of the sort S is defined to be the set of minimal elements of $S \setminus \{\varepsilon\}$. G is *sorted* if there is a family $\{S_j\}_{j \in J}$ of sorts of G such that $G = \bigcup\{S_j \mid j \in J\}$ and $S_j \cap S_{j'} = \{\varepsilon\}$ for $j \neq j'$.

Let F be a set of function symbols. We assume each $f \in F$ is assigned a set $arg(f)$ of *argument places*. Moreover, we assume $arg(f) \cap arg(f') = \emptyset$ for $f \neq f'$. G_F is a free monoid over $\bigcup\{arg(f) \mid f \in F\}$. For $f \in F$, let $S_f = \{\varepsilon\} \cup \{a\alpha \mid a \in arg(f), \alpha \in G_F\}$. Clearly, G_F is sorted with $\{S_f\}_{f \in F}$. Let F_0 be the set of symbols in F which has no argument place.

Definition 4.31 Given a set F of function symbol, $\hat{R}_F$ denotes the free complete G_F-record algebra over F_0, where $(F_0, +)$ is a trivial merge system. □

Note that it follows from proposition 4.7 and corollary 4.8 that $\hat{R}_F$ is standard.

Definition 4.32 Given a record $x \in R$, $[x]$ denotes the set of records $y \in R$ such that $y \leq x$. □

Definition 4.33 A G-record algebra R is *sorted* if there is a family $\{R_i\}_{i \in I}$ of G-record algebras such that

(1) $R = \bigcup\{R_i \mid i \in I\}$.

(2) If $x \in R_i \cap R_j$ and $x \neq \epsilon$ then x is atomic, where $i \neq j$.

(3) For $x, y \in R$ if $x + y\downarrow$ then there is some $i \in I$ such that $x, y \in R_i$.

(4) $\forall x \in R \forall \alpha \in G \exists i \in I\ [x]/\!\!/\alpha\downarrow \Longrightarrow [x]/\!\!/\alpha \subseteq R_i$.

□

In harmony with the introduction of sorts, we add the following clause to the definition 4.15 of the conflict for the record unification theory.

Definition 4.34 (In addition to definition 4.15.) Any basic constraint of the form $\alpha u \bowtie \beta v$ is a *conflict*, where α and β belong to distinct sorts of the feature monoid G, $\alpha \neq \varepsilon$, $\beta \neq \varepsilon$ and u, v are record terms. □

For example, $f_1 x \bowtie g_2 y$ is a conflict, whereas $f_1 x \bowtie f_2 y$ is not a conflict, where f_1 and f_2 are distinct argument places of f and g_2 is an argument place of g, provided that $f \neq g$.

Let F be a set of function symbols and $F_+ = F \setminus F_0$ as above. Let $\mathcal{Z}$ be the set of ordered pairs $(\{S_f\}_{f \in F_+}, S_F)$ such that $S_f \subseteq \hat{R}_F$ and $S_F \subseteq \hat{R}_F$. We define an order relation $\leq$ on $\mathcal{Z}$ by $(\{S_f\}_{f \in F_+}, S_F) \leq (\{S'_f\}_{f \in F_+}, S'_F)$ iff $S_f \subseteq S'_f$ for each $f \in F_+$ and $S_F \subseteq S'_F$.

Definition 4.35 $\{R_f\}_{f \in F_+}$ and R_F are a family of sets and a set such that the ordered pair $(\{R_f\}_{f \in F_+}, R_F)$ is the largest element in $(\mathcal{Z}, \leq)$ which satisfies the following:

(1) If $x \in R_f$ then $x = \{\epsilon\} \cup F_0$ or $x = a_1 x_1 + \cdots + a_n x_n$, where $\{a_1, \cdots, a_n\} \subseteq arg(f)$ and $\{x_1, \cdots, x_n\} \subseteq R_F$.

(2) $R_F = \bigcup\{R_f \mid f \in F_+\}$.

□

Proposition 4.20 *Given a set of function symbols there is a complete, standard and sorted G_F-record algebra $(R_F, G_F, +, \cdot, /\!\!/, \epsilon)$ over F_0 w.r.t. $\{R_f \mid f \in F_+\}$ satisfying definition 4.35.*

Proof It is clear by definition. □

Let L be a set of atomic features, F a set of function symbols, $F_0 = \{f \in F \mid arg(f) = \emptyset\}$, $L_F = \bigcup\{arg(f) \mid f \in F\}$, and X a set of parameters. To introduce DCG over the record algebras in the below, we define terms over X, L, F expressing records and an embedding translation τ between standard terms and record terms.

Definition 4.36 Let L and F be as the above and let P be a set. Then $\mathcal{T}(L, F, P)$ is the *least* set T such that the following hold:

(1) $P \subseteq T$.

(2) If $a_1, \ldots, a_n$ are features and $p_1, \ldots, p_n \in T$ then the set $\{(a_1, p_1), \ldots, (a_n, p_n)\}$ is in T.

(3) If $f \in F$ is a function symbol of arity $n \leq$ and $p_1, \ldots, p_n \in T$ then the form $f(p_1, \ldots, p_n)$ is in T.

□

An element of $\mathcal{T}(L, F, X)$ is called a *term*. Let $G = (L \cup L_F)^*$.

Definition 4.37 A *translation* τ is a partial function from $\mathcal{T}(L, F, X)$ into $\mathcal{E}(G, F_0 \cup X, \{\cdot, +\})$ such that the following hold:

(1) If $u \in F_0 \cup X$ then $\tau(u) = u$.

(2) $\tau(\{(a_1, p_1), \ldots, (a_n, p_n)\}) = a_1\tau(p_1) + \cdots + a_n\tau(p_n)$, where $a_i \in L$.

(3) $\tau(f(p_1, \ldots, p_n)) = b_1\tau(p_1) + \cdots + b_n\tau(p_n)$, where $arg(f) = \{b_1, \cdots, b_n\}$.

□

H_F denotes the Herbrand universe over F. By $(H_F, =)$, we mean the standard unification theory over H_F. We take the theory $(H_F, =)$ as a familiar congruence closure operation on sets of standard term equations. By $(R_F, \bowtie)$, we mean the theory of sorted record constraints taken as the closure operation given by table 4.1. Now we are at the place to state and prove that $(R_F, \bowtie)$ is a 'conservative extension' of $(H_F, =)$.

Theorem 4.21 *Let s and t be first-order terms then the following are equivalent.*

(1) $s = t$ is solvable in $(H_F, =)$.

(2) $\tau(s) \bowtie \tau(t)$ is solvable in $(R_F, \bowtie)$.

Proof Let C be a constraint in $(H_F, =)$. As the unifiability and satisfiability is equivalent in $(H, =)$ ([46]) and $(R, \bowtie)$ (theorem 4.13), respectively, it is straightforward to show that for any standard terms s, t the following are equivalent.

(1) C is the consistent closure of $s = t$ in $(H, =)$

(2) $\tau(C)$ is the consistent closure of $\tau(s) \bowtie \tau(t)$ in $(R, \bowtie)$.

□

4.3.5 Definite Clause Grammar over Records

As an application of record algebras, we extend the definite clause grammar (DCG) over the Herbrand universe to that over a record algebra. Let R be a G-record algebra over a merge system M, where G is a feature monoid. A *DCG* is a finite set of *rules* of the following form:

$$p_0 \leftarrow q_1 \bowtie r_1, \ldots, q_m \bowtie r_m \mid p_1, \ldots, p_n$$

where p_i, q_j, $r_j \in \mathcal{E}(G, M \cup X, \{\cdot, +\})$, $n \geq 0$, $m \geq 0$, and X is a set of parameters. In the same way as for programs over the record algebra R, the *semantics* of a DCG, say D, over R is defined to be the largest subset $\mathcal{M}_D$ of R such that the following hold: Any $t \in \mathcal{M}_D$ there exists some DCG rule $p \leftarrow C \mid Q$ in D and assignment f such that the following hold:

(1) t is an instance of p with f.

(2) f satisfies the constraint C.

(3) f has some extension f' such that every element of Q has an instance in $\mathcal{M}_D$ with f'.

Also an operational semantics of a DCG is defined in the same way for the program semantics except a slight modification of definition 4.38 of the constraint associated with resolution steps as follows:

Definition 4.38 An ordered pair (s, s') of two states $s = (Q, E)$ and $s' = (Q', E')$ is a *resolution step*, written $s \rightarrow s'$, if the following hold, where $Q = \{p_1, ..., p_n\}$:

(1) $Q' = \{q_1^1, ..., q_{k_1}^1, ..., q_1^n, ..., q_{k_n}^n\}$ for some fresh copies $q_i \leftarrow C_i \mid q_1^i, ..., q_{k_i}^i$ $(1 \leq i \leq n)$ of rules in D.

(2) E' is the consistent closure of $\{p_1 \bowtie q_1, ..., p_n \bowtie q_n\} \cup E \cup C_1 \cup \cdots \cup C_n$.

□

The set $\{p_1 \bowtie q_1, ..., p_n \bowtie q_n\} \cup C_1 \cup \cdots \cup C_n$ is called the constraint *associated with* the resolution step. The same results about soundness and completeness are obtained in almost the same way as in the case of program semantics.

For illustration, we show a simplified interpreter for DCG and a sample DCG. Let L be a set of atomic features, F a set of function symbols, $F_0 = \{f \in F \mid arg(f) = \emptyset\}$, and X a set of parameters. Define $F' = F \cup \{\#\}$ for some new function symbol $\# \notin F$ so that $arg(\#) = L$. So by proposition 4.20, we have the sorted complete and standard $G_{F'}$-record algebra $R_{F'}$ over F_0' defined for F'.

Thus we can precisely say that the example below is a DCG over the record algebra $R_{F'}$ and is written in $\mathcal{E}(G_{F'}, F_0' \cup X, \{\cdot, +\})$. Infix notations are used freely as in the standard Prolog. Unit clauses (2) and (3) below are for lexical items. The equality $=$ in the body of (1) means the builtin constraint $\bowtie$. a/b is used for the ordered pair (a, b). Each clause r there means a grammar rule $\Theta(r)$ defined by the following equations, where τ is the translation in definition 4.3.4.

$$\begin{array}{lll}
\Theta((p, C, B)) & \stackrel{\text{def}}{=} & (\tau(p), \tau'(C), \tau''(B)). \\
\tau'(C) & \stackrel{\text{def}}{=} & \{\tau(q) \bowtie \tau(r) \mid q = r \in C\}. \\
\tau''(B) & \stackrel{\text{def}}{=} & \{\tau(q) \mid q \in B\}.
\end{array}$$

```
(1) {cat/s, head/H}<-  H={subject/H1} |
        {cat/np, head/H1},
        {cat/vp, head/H}.
(2) lex(jack, {cat/np, head/jack}).
(3) lex(runs, {cat/vp, head/{subject/X,
                                pred/run(X)}}).
```

The clauses from (4) to (8) describe a simplified interpreter for the DCG grammars.

```
(4) parse([X|Y]-Y, F)<- lex(X, F).
(5) parse(X-Y, (A, B))<-
        parse(X-Z, A),
        parse(Z-Y, B).
(6) parse(X-Y, F)<-
        (F<-B),
        parse(X-Y, B).
```

The execution of the grammar looks like this:

```
?-parse([jack, runs]-[], F).

F={cat/s,{head/{subject/jack, pred/run(jack)}}}.
```

4.4 Concluding Remarks

Several relevant issues on feature structure such as complement and disjunction feature constructors are out of place. Also set values as feature values [79] and unification under inheritance hierarchies [87] are not considered. However, in the programming language CIL [55] (also described in Chapter 4), from which the record algebra came out, full first order terms possibly with parameters are allowed to be features like *brother*(n) and *brother*(m) in

$$\{(brother(n), John), (brother(m), Jack)\},$$

where n and m are parameters. Also, this aspect is not treated in the record algebra.

We have put a condition onto the structure of feature monoids so that they are essentially the same as free monoids. It is an open problem to extend the notion of feature monoids as wide as possible so that the intuitive notion of feature structures is still preserved.

Chapter 5

CIL: A Constraint Logic Programming System

5.1 The Outline of CIL

CIL is an extension of Prolog for natural language processing. CIL can be seen as an extension of DCG (Definite clause grammar) formalism [73, 72]. It was one of the earliest working systems in the middle of 80s which treated feature structures. Main aspects of CIL are as follows. First of all, a recursive data structure called *partially specified terms* (PST) are built-in. A PST is a set of attribute-value pairs. The general form of a PST is:

$$\{a_1/b_1, \cdots, a_n/b_n\}$$

where a_i is 'attributes', b_i is the 'value' of a_i. b_i may be an 'atom' or a PST. Unlike the standard terms and syntax trees, a PST has no order relation among its daughters, i.e. it is an unordered tree. Computational linguistics has been shifted its concerns from ordered trees to unordered ones.

Components of a PST can be accessed by names. The expression $\pi!s$ denote the component of PST π whose 'slot name' is s. For example:

$$\{a/b, c/d\}!a = b.$$

CIL has *term constraints, arithmetic constraints* and *Boolean constraints* to support coroutine programming for linguistic analysis. Also *one-way unification* is introduced as device for the user to define his own constraints. By the one-way unification one can prevent from instantiating parameters at the caller side on the parameter passing. Constraints of CIL are passive in general, and they do not produce values. So, the standard two-way unification must not be used for the parameter passing.

A term constraint is a set of term equations and unequations. They are passive constraints. For example, the term constraint $f(x,a) = f(b,y)$ declares that the values which may be bound to the variables x and y must be the constants b and a, respectively.

The Boolean constraints are reduced to the following three predicates:

$$and(x, y, z)$$

$$not(x, y)$$

$$or(x, y, z).$$

The declarative reading of $and(x, y, z)$, for example, is that the logical product of x and y is z. Operationally, when some of x, y, and z are known, and the others are determined uniquely by the truth table, then they are bound automatically according to the table. In other words, boolean constraints are partially active. For example, suppose that the call $and(x, y, z)$ and the unification $z = true$ are performed, then the following two unifications are performed automatically: $x = true$ and $y = true$. This idea of relational Boolean predicates is similar to the relational arithmetic predicate $add(x, y, z)$, which is used for subtraction as well as addition. Relational approaches to control problems make programming less dependent on the flow of information. The idea here is, to make a correspondence between event states and the truth values as follows:

the event has occurred	$\Longleftrightarrow$	*true*
the event is impossible	$\Longleftrightarrow$	*false*
the event is possible	$\Longleftrightarrow$	< unbound >

So the control problem is reduced to some calculus of such as Boolean algebras. In fact, the truth table is the well-known Kleene's 3-valued logic. This idea of Boolean constraints are easy to implement, and term constraints and arithmetic constraints are built on these Boolean constraints. For example, consider the term constraint (5.1). Provided that u, v, w are the truth value of the three constraints (5.1), (5.2), and (5.3), it follows from (5.4) that 5.1 is reduced to (5.5).

$$f(x, a) = f(b, y) \tag{5.1}$$

$$x = b \tag{5.2}$$

$$y = a \tag{5.3}$$

$$f(x, a) = f(b, y) \iff x = b \land y = a \tag{5.4}$$

$$and(v, w, u) \tag{5.5}$$

The Boolean constraint solver is implemented based on the *freeze* predicate[21]. It is a lazy execution primitive so that the call $freeze(x, g)$ suspends the goal g until the variable x has a value.

CIL build-ins *complex indeterminates* of situation semantics. Its form is:

$$x : C$$

and reads that "x such that C". See Section 5.6.4 for historical remarks. The complex indeterminates, $x : C$, are interpreted by the following rule.

$$unify(x : C, y) :\!-\ \ unify(x, y),\ solve(C).$$

This rule is close to that given in DLOG [33] for an indefinite description.

The domain of CIL consists of nested structures of possibly non-well-founded first-order terms and records, which were described in Chapter 3 and 4.

For example, one can write a CIL term like this:

$$f(\{a/x, b/g(\{c/x\})\}).$$

The CIL project started around 1983. Several versions of CIL has been implemented. The most earliest experimental version was implemented in Edinburgh-Prolog on DEC-2060 machine. Then it was transplanted on PSI-II machine as a programming environment mainly for natural language processing [8, 63].

5.2 CIL System

5.2.1 Syntax of Terms

We define a class of *terms* and *program clauses* of CIL by extending the standard first-order term. We first assume that two disjoint sets *PARAMETER* and *CONSTANT*. For the sake of simplicity, we assume *CONSTANT* includes atomic symbols, integer constants and function symbols. CIL syntax is an extension of Edinburgh Prolog Syntax [72]. The following delimiter symbols are used in the language as usual:

```
{   }  ,  (   )   /  :-    ;
```

Definition 5.1 A *term* is defined *inductively* as follows:

(1) A parameter is a term.

(2) If f is a constant and $x_1, \ldots, x_n$ are terms with $n \geq 0$ then the form $f(x_1, \ldots, x_n)$ is a term.

(3) If $a_1, \ldots, a_n$ are first-order terms and $x_1, \ldots, x_n$ are terms then the *set* $\{a_1/x_1, \ldots, a_n/x_n\}$ of *ordered pairs* a_i/x_i is a term.

□

A constant is a term by (2) with $n = 0$. A term of the form $f(x_1, \ldots, x_n)$ in (2) is called a *totally specified term (TST)*, while the term $\{a_1/x_1, \ldots, a_n/x_n\}$ is called a *partially specified term (PST)*. The empty set $\{\} = \emptyset$ is called the *empty PST*. Several function symbols are reserved as follows:

$\boldsymbol{x, y}$	: conjunction $x \wedge y$.
$\boldsymbol{x; y}$	: disjunction $x \vee y$.
$\boldsymbol{not(x)}$	: negation $\neg x$.
$\boldsymbol{x : y}$	: an object x with a constraint y.
$\boldsymbol{x@y}$	: an object x with a lazy constraint y.
$\boldsymbol{x\#y}$	: a tagged term. This is equivalent to $x : (x = y)$.
$\boldsymbol{x!y}$	: a labeled term. This is equivalent to $z : (x = \{y/z\})$, which means the y-component of x.
$\boldsymbol{x?}$	: a frozen term. The execution of the subgoal which contains $x?$ is suspended while x is unbound.

In the above, binary functions are used in infix forms, and the unary function ? in postfix form.

Example 5.1 The following are terms in CIL.

Parameters	`X  Man  X101  Salary  _325.`
Constants	`378  'Man'  x1013  abc.`
TSTs	`[1, 2, 3, 5]  3+5  f(1, abc, X).`
	`soa(give, {agent/A, object/B, recipient/'Jack'}, 1).`
PSTs	`{}  {agent/father(X), object/O, recipient/ X}.`
	`{f(X)/Y, f(Y)/X} .`
Conditioned Terms	`Z@(Z>0)  X:(man(X), wife_of(X,Y), pretty(Y)).`
Tagged Terms	`X#4  Sit#soa(R, {agent/A, soa/Sit}, P).`
Labeled Term	`Man!name!first.`
Frozen Terms	`X?  (Man!name)?.`
Conjunction	`(X>0, X<10).`
Disjunction	`(X>0; X<0).`
Negation	`(not X<Y).`
Query	`?- print(X?), X=ok.`

□

5.2.2 Program Clause

A CIL *program* is a finite set of program clauses. A *(program) clause* is a pair (h, b) of a term h and a set b of terms. The clause $(h, \{b_1, \ldots, b_n\})$ is written

$$h : b_1, \ldots, b_n.$$

The term h is called the *head* of the clause while the set b is called the *body* of the clause. A *unit clause* is a program clause whose body is empty. The unit clause $(h, \emptyset)$ is written simply

$$h.$$

A *query* is of the form

$$?\text{-}g.$$

where g is a set of terms. The set g is called the *goal* of the query. A *subgoal* is an element of the goal.

These reserved forms are macros; they are translated into normal forms on being read by the system. The *rules of macro expansions* are as follows:

1. $x@c \Longrightarrow x : \mathit{freeze}(x, c)$.

2. $x\#y \Longrightarrow x : (x = y)$.

3. $x!y \Longrightarrow z : (x = \{y/z\})$.

4. $x : c \longrightarrow x$. As a side effect, c is moved so that c becomes a subgoal of the clause.

5. $x? \Longrightarrow x$, and $g \Longrightarrow \mathit{freeze}(x, g_x^{x?})$, where g is the subgoal in the clause which contains the occurrence $x?$. The symbol g_a^b means the new term obtained by substituting a for all occurrences of b in the term g.

These rules are applied in *outer-most-first* way.

Example 5.2 The clause (2) is the translation of (1).

(1) $p(x\#\{a/y\}) : -b((y!c)?)$.
(2) $p(x) : -x = \{a/y\}, y = \{c/v\}, \mathit{freeze}(v, b(v))$.

□

Note that all of ?, @, :, #, and ! are eliminated on the way of the translation. So, in the following sections, we assume that none of these signs appears in CIL terms.

5.3 Operational Semantics of CIL

The operational semantics of CIL follows that of standard Prolog [46]; the computation selects clauses *from-top-to-down*, and goals *from-left-to-right*.

A *computation state* (state, for short) is a triple (g, E, F), where g is a goal, E a set of equations in the *solved form*, and F a set of pairs (v, g') of a parameter v and a goal g'. E is also called an *environment* or *parameter bindings*. A parameter x is *bound* in the state if E has an equation of the form $x = t$ or $t = x$ for some non parameter term t. F means a set of *frozen goals*. A parameter is called *frozen* in the state if there is a pair (x, g) in F for some goal g. For the given goal g, the *initial state* is the state $(g, \emptyset, \emptyset)$. A *computation tree* is an ordered tree such that each node is labeled with states.

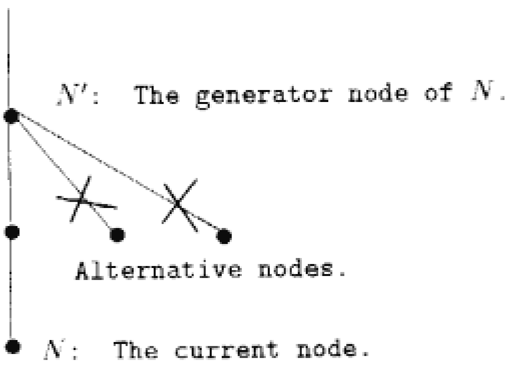

Figure 5.1: Cutting the alternatives.

The computation for a given goal is a construction process of a computation tree. The initial tree is a singleton tree; the single node is labeled with the initial state for the goal. The computation constructs the tree in *top-down, depth-first and left-to-right* basis. Basic operations are *expansion* and *cut*. To define them, let first T be the current computation tree and N the current node.

Expansion	Create a daughter node N' of N with a label s', provided that $s \longrightarrow s'$.
Cut	Do $cut(T, N', N)$, provided that N' is the generating node in N'.

Now we must define $s \longrightarrow s'$, generating node, and $cut(T, N', N)$. First, in general, each subgoal g has a unique ancestor node N_g such that g was introduced at N_g first in the computation tree. The node N_g is called the *generating ancestor* of the subgoal g. For the second, to define the cut operation, let T be a computation tree and let N and N' be a node of T such that N' is ancestor node of N in T. Then we define $cut(T, N', N)$ to be the computation tree obtained by 'pruning off' all the descendant nodes of N' which is neither an ancestor nor descendant of N (see Figure 5.1). Finally, we define the transition relation $\longrightarrow$ in the following, where $\cup$ means the disjoint union of sets; $u \cup v$ presupposes $u \cap v = \emptyset$.

Definition 5.2 For states s, s', we define $s \longrightarrow s'$ iff one of the following holds:

Resolution	$s = (\{g_1, g_2, \ldots, g_n\}, E, F)$ and $s' = (\{b_1, \ldots, b_m, g_2, \ldots, g_n\}, E', F)$ where there is a fresh copy $a{:-}b_1, \ldots, b_m$ of some program clause and E' is the solved form of $E \cup \{g_1 = a\}$.
Unification	$s = (\{r = r'\} \cup g, E, F)$ and $s' = (g, E', F)$, where E' is the solved form of $E \cup \{r = r'\}$
Freeze	$s = (\{freeze(u, g')\} \cup g, E, F)$ and $s' = (g, E, \{(u, g')\} \cup F)$.
Melt	$s = (g, E, \{(u, g')\} \cup F)$ and $s' = (g' \cup g, E, F)$, where u is bound in E.
Cut	$s = (\{cut\} \cup g, E, F)$ and $s' = (g, E, F)$. As a side effect of this step, all the *alternative branches* in the computation are pruned off, as explained above.
Disjunction	$s = (\{(g; g')\} \cup g'', E, F)$ and either $s' = (g \cup g'', E, F)$ or $s' = (g' \cup g'', E, F)$.
Negation	$s = (\{not\ g\} \cup g', E, F)$ and either $s' = (g \cup \{cut, fail\}, E, F)$ or $s' = (g', E, F)$.
Fail	For $s = (\{fail\} \cup g, E, F)$, s' is undefined.

□

The basic step $(g, E, F) \longrightarrow (g', E', F')$ logically reads that $g' \wedge E' \wedge F'$ *implies* $g \wedge E \wedge F$; any solution of $g' \wedge E' \wedge F'$ extends to some solution of $g \wedge E \wedge F$.

5.4 Basic Constraints

CIL built-in constraints are classified into the three groups: unification, boolean, arithmetic and (first-order) term constraint. These constraints have a common property that the direction of data flow, i.e., input and output, depends on the context. In addition to them, a meta constraint $freeze(x, g)$ is explained.

The term constraint is written $u = v$, where u and v are first-order terms. Unlike the unification, it never instantiates any parameters of the equation. For example, the execution $x = a$ constrains x so that only a is the possible value of x. Rewriting rules are useful to describe behaviors of constraints. The rule $\alpha \Longrightarrow \beta$ reads that if the constraint α is in the system then replace it with β.

5.4.1 Freeze

The call *freeze*(x, g) suspends the execution of g while x is unbound. If there are two constraints *freeze*(x, g) and *freeze*(x, h) in the current state, where x is unbound. Then replace them with *freeze*$(x, g \cup h)$. If x is bound then replace *freeze*(x, g) by *solve*(g). In summary, we have the following rewriting rules:

$$\begin{array}{lcll} \mathit{freeze}(x, g), \mathit{freeze}(x, h) & \Longrightarrow & \mathit{freeze}(x, g \cup h), & \text{if } x \text{ is unbound.} \\ \mathit{freeze}(x, g) & \Longrightarrow & \mathit{solve}(g), & \text{otherwise.} \end{array}$$

5.4.2 Constraint Solver

The form *constr*(c, g) calls the constraint solver. There are three types of calls: *constr*$(c, true)$, *constr*$(c, false)$, and *constr*(c, x), where c is a constraint and x is a unbound parameter. They are called *active*, *passive* and *intermediate mode*, respectively. In the passive mode, the constraint solver never instantiates parameters appearing in the constraint c. The constraints are checked without unification when some parameters are bound by other processes. In the active mode, the constraint solver executes equality constraints as the ordinal unification. In the intermediate mode, the parameter may be bound to the value by the solver only when it is the unique possible value for the parameter.

5.4.3 Boolean Constraint

Boolean constraints have the following three forms: *and*(x, y, z), *or*(x, y, z), and *not*(x, y). They mean $z = x \wedge y$, $z = x \vee y$ and $y = \neg x$, respectively. These constraint bind values to the parameters when only one possible solution remains in the state. For instance, suppose that the subgoal *and*(x, y, z) was called at some time with all the three parameters unbound and that now z is bound to the Boolean value *true*. Then x and y are bound to *true* because it is the only possible solution by the truth table for $z = x \wedge y$. These constraint predicates are implemented based on *freeze*. In general, it is a policy of the CIL design that constraint solving is driven by the binding events. Here are the rules for Boolean constraints:

$$\begin{array}{l} \mathit{constr}(\neg(c), m) \Longrightarrow \mathit{constr}(c, m'), \mathit{not}(m, m'). \\ \mathit{constr}(a \wedge b, m) \Longrightarrow \mathit{cosntr}(a, r), \mathit{constr}(b, s), \mathit{and}(r, s, m). \\ \mathit{constr}(a \vee b, m) \Longrightarrow \mathit{cosntr}(a, r), \mathit{constr}(b, s), \mathit{or}(r, s, m). \end{array}$$

The design of CIL Boolean constraints provides the user with tools for program controls in relational boolean calculations. In general, Boolean constraints are important: many non-trivial classes of constraints can be reduced to Boolean constraints. For instance, Johnson's attribute-value logic [41] and Smolka's feature logic [86] are based on translations from their logics into some sublanguages of quantifier-free first-order logics, i.e., ones close to the propositional calculus.

The current CIL boolean constraint solver is not complete. However, in theory, it is not difficult to implement a complete solver for Boolean constraints. In fact, the following mathematical fundamentals are enough for a complete Boolean constraints.

A Boolean algebra $B = (D, \wedge, \vee, \neg, 1, 0)$ and a *Boolean expression* are defined as usual. Also a Boolean ring $R = (E, \cdot, +, 1, 0)$ is defined as usual. Let $B = (D, \wedge, \vee, \neg, 1, 0)$ a given Boolean algebra. Then, we define the Boolean ring $R = (D, \cdot, +, 1, 0)$ by the following equations.

$$\begin{aligned} xy &= x \cdot y = x \wedge y. \\ x + y &= (x \wedge \neg y) \vee ((\neg x) \wedge y). \end{aligned}$$

Clearly, the following hold:

$$\begin{aligned} x \vee y &= (x + y) + xy. \\ \neg x &= 1 + x. \\ x \wedge y &= xy. \end{aligned}$$

So, there is an evident translation τ from Boolean expressions e into Boolean ring expressions $\tau(e)$ such that e is satisfiable in B iff $\tau(e)$ is satisfiable in R. In Boolean ring, we have the following:

$$\begin{aligned} ax = b &\implies ab = b. \\ ax = b &\implies (1 + a)(x + b) + b = x. \\ x = (1 + a)y + b &\implies ax = b. \end{aligned}$$

Thereby, the following conditions are equivalent.

- $ax = b$ is satisfiable.
- $ab = b$ is satisfiable.
- $x = (1 + a)y + b$ for some y.

In fact, these properties of Boolean ring are mathematical background for the solvers based on parameter elimination. See Dincbas et al [27]. Prolog- III [47] build-ins a complete boolean constraint solver based on a different algorithm from the one explained above. Also see a recent work on Gröbner-base based Boolean constraint solver [81].

5.4.4 Arithmetic Constraint

The form $add(x, y, z)$ is an built-in arithmetic constraint that $x + y = z$. For example, the constraint solver will bind x to 2 when, for example, y is 1 and z is 3.

5.4.5 Term Constraint

The form $constr(u = v, m)$ is a built-in term constraint, where u and v are first-order terms and m is a parameter, *true*, or *false*. The constraint is solved by repeating the following rewriting rules, where m and x be an unbound parameter.

- $constr(u = v, true) \Longrightarrow solve(u = v)$.
- $constr(a = a, m) \longrightarrow solve(m = true)$.

- $constr(f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n), m) \Longrightarrow \begin{cases} and(m_1, m_2, m_3) \\ \vdots \\ and(m_{n-1}, m_n, m) \\ constr(t_1 = u_1, m_1) \\ \vdots \\ constr(t_n = u_n, m_n) \end{cases}$

  where m_i is a new parameter, for $1 \leq i \leq n$.
- $constr(f(\ldots) = g(\ldots), m) \Longrightarrow solve(m = false)$, where $f \neq g$.
- $constr(x = u, m) \Longrightarrow constr(v = u, m)$, where x is bound to the term v.

5.4.6 One-way Unification and Sequential Control

The predicate *assign* is a built-in predicate for one-way unification, i.e., *pattern matching*. The following example shows the difference between the standard unification and *assign*, the pattern matching. The result of the unification $f(x,1) = f(2,y)$ is $x = 2$ and $y = 1$. On the other hand, the result of the pattern matching $assign(f(x,1), f(2,y))$ is $y = 1$ but leaves x unbound.

Let x and v be a parameter and non-parameter term, respectively, and let u be any term. Then the rewriting rules for $assign(u,v)$ are as follows.

(1) $assign(x,u) \Longrightarrow assign(v,u)$, if x is bound to v.
(2) $assign(u,x) \Longrightarrow assign(u,v)$, if x is bound to v.
(3) $assign(u,x) \Longrightarrow solve(u = x)$, if x is unbound.
(4) $assign(a,a) \Longrightarrow solve(true)$, where a is a constant.
(5) $assign(f(t_1,\cdots,t_n), f(u_1,\cdots,u_n)) \Longrightarrow assign(t_1,u_1), \cdots, assign(t_n,u_n)$.
(6) $assign(a,b) \Longrightarrow solve(fail)$, if a and b are different constants.
(7) $assign(f(\cdots), g(\cdots)) \Longrightarrow solve(fail)$, where f and g are different function symbols.

Note that there is no applicable rule in the list for $assign(x,u)$ if x is unbound and u is bound to a non-parameter term. In this case we say that the constraint *is suspended*. This suspending mechanism is frequently used in the actual implementation of CIL. We say that a constraint g *is partially solved* when g has been rewritten into g' by the normalization rules and there is no applicable rules for g'.

We need one more primitive, *sequand*, which means 'sequential execution'. $seqand(g, g')$ constrains that g' is executed only after g is partially solved. Here is an example for using the one way unification and sequential 'and'. The standard membership predicate is written in Prolog:

$member(x, [x|\_]).$
$member(x, [\_|y])\text{:-}member(x, y).$

Let *mem* be a constraint version of *member*. In the current CIL, the definition of *mem* can be given as the following unit clause:

$$defcon(mem(x,y), seqand(assign(y,[h|t]), (x = h; mem(x,t)))) \tag{5.6}$$

where *defcon* is a reserved predicate symbol for the declaration of the user constraint rules.

The example *mem* explains how the user defined constraint is solved. The point is that the constraint reduction of *mem* goes *without backtracking*. It is this point that is different from the standard execution of *member*. The call of the constraint $mem(x, y)$ goes as follows. First, parameters h and t are assigned values from y by the one-way unification, i.e., the pattern matching with the input parameter y, The disjunctive constraint $x = h \vee mem(x,t)$ is partially solved. If the term constraint $x = h$ is not partially solved yet, the constraint $mem(x, t)$ is called recursively. If y is unbound, due to the use of *seqand* in the body of the definition the call $mem(x, y)$ suspends not only $assign(y, [h|t])$ but also $(x = h; mem(x, t))$ while y is unbound.

Example 5.3 Let us trace the following call (5.7).

$$constr(mem(x, [a]), true). \tag{5.7}$$

By definition (5.6) of *mem*, this is equivalent to the goal (5.8).

$$constr(\boldsymbol{seqand}(\boldsymbol{assign}([a], [h|t]), (x = a; mem(x, []))), true). \tag{5.8}$$

By the rewriting rule of ***assign***, the constraint $assign([a], [h|t])$ is reduced to $h = a$ and $t = []$. Hence, by the *seqand* rule and the conjunction rule, the constraint (5.8) is reduced to the three constraints (5.9), (5.10), (5.11) for some new parameters v and v'.

$$constr(x = a, v). \tag{5.9}$$

$$constr(mem(x, []), v'). \tag{5.10}$$

$$or(v, v', true). \tag{5.11}$$

The constraint (5.10) is reduced to $v' = false$. By the *or*-Boolean constraint rule with this binding, the constraint (5.11) is reduced to $v = true$. Hence from (5.9), now the problem has been reduced to the goal (5.12).

$$constr(x = a, true). \tag{5.12}$$

Finally, we obtain $x = a$ from the last *active* constraint (5.12). This result is what we expect, because the constraint $mem(x, [a])$ declares that there is no other possibility for x than $x = a$. □

5.5 Built-in Predicates

CIL built-in functions are listed below with example uses. Only relevant predicates to either PSTs or constraint are listed because other ones follow those of the standard Prolog. By $[\![x]\!]$, we means the term t such that $x = t$ is in the environment. For example, if $x = 1$, then $[\![x]\!] = 1$. For convenience, if x is unbound then we define $[\![x]\!] = x$. Suppose that u is a parameter and the equation $u = p$ is in the environment for some PST p. Then, by $u(x)$ we means $p(x)$ for convenience. For example, if $u = \{a/1, b/2\}$ then $u(a) = 1$ and $u(b) = 2$.

5.5.1 Extended Unification

unify(t,u) This call unifies t with u. This call is also written $t = u$. This predicate is an implementation of the unification described in Section3.5.4 and 4.2.3.

The call $x = \{a/1\}$, $y = \{b/2\}$, $x = y$ yields $x = y = \{a/1, b/2\}$. Also $\{a/b, c/\{d/x\}\}!c!d = h$ yields $x = h$. The execution of $x = \{a/ok\}$, $y = \{a/@print\}$, $x = y$ displays *ok*, where *print* is a built-in output predicate. The call $\{a/x, b/c\} = \{a/1, b/y\}$ yields $x = 1$, $y = c$. The call $x\#\{a/1, b/x!a\} = y$ yields $x = y, x = \{a/1, b/1\}$. The call $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$, $x = y$ yields, $x = y = \{a/b, c/x\}$. The execution of $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$ yields $x = \{a/b, c/y\}$, $y = \{a/b, c/x\}$.

assign(u,v,z) This call matches u with v by the one-way unification. It returns $z = true$ if the matching is successful, else $z = false$ if matching fails, otherwise nothing. The last case means that the process has been suspended to wait till u is more instantiated. The parameters in u are treated as read-only parameters. The call $assign(f(x), f(y), z)$ yields $z = true$, $x = y$. The call $assign(f(x), f(a), z)$ yields no bindings, i.e., x and y remain unbound. The call $assign(f(a), f(x), z)$ yields $x = a$, $z = true$.

5.5.2 Utilities

same(u,v) This call checks whether u and v are intentionally the same. For example, the first of the following two calls fails, but the second succeeds.

$$same(\{a/\{b/\_, c/\_\}\}, \{a/\{c/1, b/b\}\}).$$
$$\{a/A\#\{b/\_, c/\_\}\} = \{a/B\#\{c/1, b/b\}\}, same(A, B).$$

dif(u,v) This call declares that u and v are different from each other. It is equivalent to $not(dif(u, v))$.

fullCopy(u,v) This call makes a fresh copy of u and unifies v with it. Even the frozen conditions accumulated on the parameters in u are copied. The call $x = \{a/@print, b/x\}$, $fullCopy(x, z)$, $z!b!a = ok$ will display ok on the output screen.

typeOf(u,type(v,w)) This call is equivalent to the call $fullCopy((v, w), (u, z))$, $solve(z)$.

createType(u, v, type(w,z)) This call is equivalent to the call $fullCopy((w, z), (u, v))$. The call $createType(y, (y = 1; y = 2), t)$, $typeOf(1, t)$, $typeOf(2, t)$ succeeds.

instance(u,v) This call is equivalent to the call $fullCopy(v, w)$, $unify(w, u)$.

5.5.3 Record Utilities

The following utilities handle records.

getRole(u, k, v) This call is equivalent to the call.

$$(k = k_1; \cdots; k = k_r), u = \{k/v\}$$

where $dom([\![u]\!]) = \{k_1, \ldots, k_r\}$ in the state. Put differently, this call finds the key k in u to return the value v of the key. This is equivalent to the unification $u = \{k/v\}$ in declarative sense. No argument place of u is created. This predicate may have backtrack points. The key k does not need to be ground. This predicate is similar to the predicate *locate* below. In the case that k is known to be ground, the predicate *locate* is more efficient than this. The execution $x = \{a/1, b/2\}$, $getRole(x, k, v)$ yields $k = a$, $v = 1$ as the first solution and then $k = b$, $v = 2$ as the second one.

locate(u,k,v) If k is bound to a ground first-order term and $k \in dom(u)$ then solve $u = \{k/v\}$ otherwise fails. *locate* is similar to *getRole* above. However k must be ground. The call will fail if u has not the argument place named k. The call $locate(\{a/b\}, a, x)$ yields $x = b$. The call $locate(\{a/y\}, b, x)$ fails, where $a \neq b$. The call $locate(\{a/y\}, a, x)$ yield the unification $x = y$.

setOfKeys(u, s) This call makes the list of keys in the record u and return it to s. The call $setOfKeys(\{a/x, b/y, c/z\}, s)$ yields $z = [a, b, c]$.

role(k, u, v) This is a constraint version of *getRole*, which declares that $getRole(u, k, v)$ is executed when k is bound to a ground first-order term. For example, the call $x = \{a/1, b/2\}$, $role(k, x, 3)$, $k = c$ yields $k = c$, $x = \{a/1, b/2, c/3\}$.

delete(k,u,v) This call generates the conjunctive constraint of $v = u'$ and 'u' is the restriction to the $dom(u) \setminus \{k\}$' which is to be solved when the value of k is grounded. Intuitively this call deletes the k-field from u. For example, the call $delete(a, \{a/1, b/2\}, z)$ yields $z = \{b/2\}$.

partial(u) This call succeeds if u is a record otherwise fails.

record(u,v) If $[\![u]\!] = \{a_1/b_1, \ldots, a_n/b_n\}$, then this call generates and solves the following constraints:

$$\begin{aligned} v &= [(a_1, b_1)|v_1] \\ v_1 &= [(a_2, b_2)|v_2] \\ &\vdots \\ v_{n-1} &= [(a_n, b_n)|v_n] \end{aligned}$$

That is, this call makes the list consisting of pairs (p, w) such that $u = \{p/w\}$ and return it to v. This predicate is similar to *buffer* below. v is generated as a stream from the record u. This predicate is used as a stream generator. For example, the call $record(\{a/1, b/2\}, r)$ yields $r = [(a, 1), (b, 2)]$. This predicate must be used only when there is no possibility that the record u will grows in the further state.

buffer(u, v) This call generates the following constraints, where $u = \{a_1/b_1, \ldots, a_n/b_n\}$:

$$\begin{aligned} v &= v_0 \\ v_0 &= [w_1|v_1] \\ &\vdots \\ v_{r-1} &= [w_r|v_r] \\ w_j &= (a_{j+1}, b_{j+1}) \quad \text{if } 0 \leq j \leq n-1 \text{ and } j \leq r. \\ w_n &= end\_of\_list \quad \text{if } n \leq r. \end{aligned}$$

These constraints are solved in an incremental way that every time when any parameter v_i gets instantiated, the corresponding unification in the above is performed.

That is, the call $buffer(u, v)$ converts the record u to the buffered list v. This is similar to the predicate *record*. Each pair (k, s) in u, i.e., $u = \{k/r\}$, is put on v as the last element of v while there is room in the list v. The *end_of_list* marker is put when the pairs in u is exhausted. If the current tail of v is unbound then the producing the rest is suspended. The call $buffer(\{a/1, b/3\}, x)$, $x = []$ succeeds. The call $buffer(\{a/1, b/3\}, x)$, $x = [y|z]$, $z = [u|v]$ yields $x = [(a, 1), (b, 3)|v]$. The call $buffer(\{a/1, b/3\}, [x, y, z, u])$ yields $x = (a, 1)$, $y = (b, 3)$, $z = end\_of\_list$.

glue(u,v) This call executes the unification $u = \{k/z\}$ and $v = \{k/z\}$ for each common key k of u and v. The call $x = \{a/1, b/z\}$, $y = \{b/2, c/3\}$, $glue(x,y)$ yields $x = \{a/1, b/2\}$, $y = \{b/2, c/3\}$, $z = 2$.

merge(u, v) This call is equivalent to the unification $v = [\![u]\!]$. That is, this call adds to v each element of u. The execution of $x = \{a/1\}$, $y = \{b/2, c/3\}$, $merge(x,y)$ yields $x = \{a/1\}$, and $y = \{a/1, b/2, c/3\}$.

d_merge(u, v) This call is the unification $v = u'$ for a maximal restriction u' of u as a function such that the unification $v = u'$ is successful. That is, this call adds to v each element of u which is unifiable with the counter part in v if any. For example, the call $u = \{a/1, b/2\}$, $v = \{a/3\}$, $d\_merge(u,v)$ yields $u = \{a/1, b/2\}$, $v = \{a/3, b/2\}$.

subpat(u,v,d) This call checks $dom(u) \subseteq dom(v)$ and unifies d with the list of the triples $(k, u(k), v(k))$, where $k \in dom(u)$. That is, this call creates onto d the list of triples (k, r, s) such that (k/r) is in u and (k/s) is in v. If u is not a subrecord of v in the sense that each key of u is also the one of v, this call fails. The call $subpat(\{a/1, b/x\}, \{a/y, b/2, c/3\}, z)$ yields $z = [(a, 1, y), (b, x, 2)]$.

extend(u,v,d) This call unifies d with the list of triples $(k, u(k), v(k))$, where $k \in dom(u) \cap dom(v)$. Also this call unifies v with the restriction of u to $dom(u) \setminus dom(v)$.

That is, this call adds each element of u to v and return in d the difference list between u and v. The call $x = \{a/1, b/2\}$, $y = \{b/z, c/3\}$ yields $y = \{a/1, b/z, c/3\}$, $d = [(b, 2, z)]$.

meet(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto the D-list d for $k \in dom(u) \cap dom(v)$. For example, the call $meet(\{a/1, b/2\}, \{b/3, c/4\}, x\text{-}[])$ yields $x = [(b, 2, 3)]$.

frontier(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto a D-list d for $k \in dom(u) \cap dom(v)$. This predicate fails if $u(k)$ and $v(k)$ are a conflicting pair for some k. It is similar for TSTs. For example, the call $frontier(f(a, g(b)), f(x, y), z\text{-}[])$ yields $z = [a = x, g(b) = y]$. The call $frontier(a, b, z\text{-}[])$ fails, where $a \neq b$ are constants.

match(u,v,d) This call creates the triples $(k, u(k), v(k))$ onto the D-list d for $k \in dom(u) \cap dom(v)$ such that $u(k) \neq v(k)$. This predicate always succeeds. The execution of

$$match(f(a, x, c), f(b, x, z), u\text{-}[])$$

yields $u = [a = b, c = z]$.

t_subpat(u,v) This call checks whether u is a *hereditary* subrecord of v, i.e, u is a pattern of v, provided that both u and v are ground record. More formally, it is checked that $dom(u) \subseteq dom(v)$ and for any x in $dom(u)$ it is the case that either $u(x)$ and $v(x)$ are the same first-order term or $t\_subpat(u(x), v(x))$. This call will fail if it is not the case. This predicate is a transitive version of *subpat*, and checks the record subsumption relation. The call

$$t\_subpat(\{a/\{b/1\}\}, \{b/1, a/\{c/2, b/3\}\})$$

succeeds. The call $t\_subpat(\{a/\{b/3\}, c/4\}, \{b/1, a/\{c/2, b/3\}\})$ fails.

t_merge(u,v) This call merges u into v in the transitive way. This predicate is a transitive version of *merge*. That is, this call generates the constraints $t\_merge(u(a), v(a))$ for each $a \in dom(u) \cap dom(v)$, and execute $v = u'$ where u' is the restriction of u to $dom(u) \setminus dom(v)$. For example, the call $t\_merge(\{b/1, a/\{c/2, b/3\}\}, \{a/\{b/y\}\})$ yields $y = 3$.

masked_merge(u,m,v) This call creates u minus m and then merge them into v. The execution of $v = \{a/2\}$, $masked\_merge(\{a/1, b/1, c/1\}, \{a/\_, b/\_\}, v)$ yields $v = \{a/2, c/1\}$.

5.5.4 Extra Predicates

bound(u) This call succeeds if u is already instantiated.

unbound(u) This call succeeds if u is not bound.

freeze(x, g) This call suspends the goal g while x is unbound.

freeze(x,y,g) This call suspend the goal g while both x and y are unbound.

if(t, g) This call solves g if $t = true$, else fails.

if(t,y,z) This call solves y if $t = true$, else solves z.

ifBound(x, g) This call solves g if x is bound, else succeeds.

ifUnbound(x, g) This call solves g, if x is unbound, else succeeds.

wif(x,y) This call suspends the goal y while x is unbound. When x is bound, y is called if $x = true$, otherwise fails.

wif(x,y,z) This call suspends goals y and z while x is unbound, and, when x is bound, executes y if $x = true$, otherwise executes z.

pv(f,g) This call performs $unify(f, true)$ if f is unbound, else this call succeeds.

solve(g) This call executes g.

when(a, g) This call suspends the goal g while a is not ground.

5.6 Linguistic Analysis in CIL

In this section, we show various kinds of applications of CIL to linguistic analysis, in particular, uses of PSTs.

5.6.1 Using Partially Specified Terms

We show a CIL program for linguistic analysis, using PSTs and constraints. It also includes uses of constraints. The program is based on the idea of situation semantics [18] in that the meaning of a sentence is a relation between discourse situations and described situations. PSTs are used for linguistics information as feature structures.

Imagine the following piece of discourse between two persons, say Jack and Betty:

(1) *Jack: I love you.*
(2) *Betty: I love you.*

The two sentences are the same, but interpretations of (1) and (2) are different as in the following (3) and (4), respectively:

(3) *Jack loves Betty.*
(4) *Betty loves Jack.*

This difference is an example of language efficiency [18]. How is this kind of language efficiency analyzed in CIL? We demonstrate the power of PSTs by giving a program which analyzes the discourse. The complete listing of the program is in the appendix.

The name of the top level predicate is `discourse_constraint`.

```
?- discourse_constraint([(1), (2)], [X,Y]).
```

The query for the program will produce answer interpretations `X=` (3) and `Y =` (4) for (1) and (2), respectively. In this illustration, *discourse constraints* are simplified as the following (5) and (6):

(5) *The speaker and hearer turn their roles at every sentence utterance.*
(6) *The successive discourse locations are numbered sequentially.*

The main clause of the program is the following:

```
(7) discourse_situation({sit/S, sp/I, hr/ You, dl/ Here, exp/ Exp}):-
        member(soa(speaking, (I, Here),yes),S),
        member(soa(addressing,(You, Here),yes),S),
        member(soa(utter,(Exp, Here),yes), S).
```

This clause declares a type of called *discourse-situation* that a discourse situation has parameters: situations (`sit`), speaker (`sp`), hearer (`hr`), discourse location (`dl`), and expression (`exp`). In other words, an object x is a discourse situation if the `sit` component of x has the three state of affairs as indicated in the body of the clause. The membership definition is as usual.

The following is the discourse constraint clauses:

```
(8) discourse_constraint([],[]):-!.
    discourse_constraint([X],[Y]):-!,meaning(X,Y).
    discourse_constraint([X,Y|Z], [Mx,My|R]):-
        meaning(X,Mx),
        turn_role(X, Y),
        time_precedent(X, Y),
        discourse_constraint([Y|Z],[My|R]).
```

The first and second arguments are a list of discourse situations and a list of described situations, respectively. The clauses constrain discourse situations and described situations with Rule (5) and (6) above. The constraint (5) is coded in the clause:

```
(9) turn_role({hr/X,sp/Y},{hr/Y,sp/X}@discourse_situation).
```

According to the context of the program, this clause presupposes that the first argument is a discourse situation. The second argument `{hr/Y,sp/X}@discourse_situation` means that the actual argument is a *discourse-situation* with `{hr/Y, sp/Y}`. Note that `X` and `Y` turn their roles in the first argument.

The constraint (6) is coded in the clause (10):

```
(10) time_precedent({dl/loc(X)},{dl/loc(Y)}):- constr(X+1=:=Y).
```

The CIL call `constr(X+1=:=Y)` constrains `X` and `Y` so that the latter is greater than the former by one.

The sentence interpretation is described in DCG form. The following clause is an interface between the discourse situation level and sentence level.

```
(11) meaning(X#{exp/E},Y):-sentence(E-[],{ip/Y,ds/X}).
```

The sentence model is very simplified as follows. A sentence consists of a noun, verb, and another noun in order. There are only four nouns, i.e., *jack*, *betty*, *i(I)*, *you*. The word *love* is the only verb here. The feature system is taken after GPSG [32]. The control agreement principle is illustrated using subcategorization features. By checking the features agreement between the subject and verb, (12) is legal, but (*13) is illegal.

(12) *I love you.*

(*13) *Jack love you.*

The verb *love* has several semantic parameters: *agent* , *object*, *location*, and so on. The first and last nouns are unified with *agent* and *object* parameters, respectively. The location comes from the given discourse situation parameter. The agreement processing and role unification are coded in the following two clauses (14), (15) using PSTs, where **ip** stands for *interpretation*.

```
(14) sentence({ip/SOA,ds/DS})-->
       noun({ip/Ag,ds/ DS, syncat/{head/F}}),
       verb({ip/SOA, ds/DS, ag/Ag, obj/ Obj, syncat/{subcat/F}}),
       noun({ip/Obj, ds/ DS}).

(15) verb({ ip/ soa(love,(X, Y, Loc), yes),
            ds/ {dl/Loc},
            ag/ X,
            obj/Y,
            subcat/ {head/{minor/{agr/
                ({plu/P, per/N}: (P=(+), N= (@per);
                                  P=(-),(N=1; N=2)))@agr}}@category})
     --> [love].              % love
```

The pronoun *I* and proper name *Betty* are described as follows. The agreement features of *I* are the first person and singular. The agreement features of *Betty* are *the third person* and *singular*. The interpretation of the pronoun *I* is the hearer of the given discourse situation.

```
(16) noun({ip/betty,
          syncat/{head/{minor/{agr/{plu/(-),per/3}@agr}}}@category})
       -->[betty].             % Betty
     noun({ip/X,
          ds/{sp/X},
          syncat/{head/{minor/{agr/{plu/(-),per/1}@agr}}@category})
       -->[i]                  % I
```

The system of syntax categories in this example is coded as follows:

```
(17) category({bar/ @bar, head/ @head}).
```

This clause says that an object which contains {bar/B, head/H} is a category, where B and H are a bar category and head category.

The following is an example of the category specification in PST notation:

```
(18)    {bar/2,
         head/ {major/ {n/ +, v/ -},
                minor/ {agr/ {per/1, plu/ -},
                        case/ acc    }}}.
```

Take query (19), to the above defined constraint, for example.

```
(19) ?- discourse_constraint(
        [{sit/ [soa(speaking,  (jack, _), yes),
                soa(addressing,(betty, _),yes)|_],
          exp/ [i,love,you],
          dl/  loc(1)}@discourse_situation,
         {exp/ [i,love,you]}@discourse_situation],
        Interpretation).
```

Note that no parameter other than expression parameter is specified in the second discourse situation in this query. The other parameters are determined by the discourse constraint. Then, the exact output of this query is (20):

```
(20)    Interpretation =
             [soa(love,(jack,betty,loc(1)),yes),
              soa(love,(betty,jack,loc(2)),yes)].
```

5.6.2 Feature Co-occurrence Restrictions

The predicate *constr* can be used to put constraints on linguistic information. Here is such a simplified example from linguistic constraint. The constraint says that if REFL feature of x is $(+)$ then the GR feature of x must be SBJ, where x is a feature set. This is called a Feature Co-occurrence Restriction (FCR) in GPSG and written

$$\langle \text{REFL } + \rangle \Longrightarrow \langle \text{GR SBJ} \rangle.$$

The following call causes the feature set x to have this agreement feature constraint.

$$constr((x!\text{REFL} = (+) \rightarrow x!\text{GR} = \text{SBJ})).$$

The feature set x will automatically get SBJ as the value of GR feature immediately when the value $(+)$ is generated at the REFL field of x. For example, the execution (1) yields (2).

(1) $constr((x!\text{REFL} = (+) \rightarrow x!\text{GR} = \text{SBJ})),\ x!\text{REFL} = (+),\ a = x!\text{GR}.$

(2) $a = \text{SBJ},\ x = \{\text{REFL}/(+), \text{GR}/\text{SBJ}\}.$

5.6.3 PSTs as Partial Assignments

A *state of affairs* (*soa*, for short) is a triple which has the form

$$\ll R, a, p \gg$$

where R, a, and p are a relation, partial assignment, and polarity, respectively. Each relation R is given a set $arg(R)$ of argument places. A partial assignment for R is a partial function from $arg(R)$ which assigns objects to argument places. Each argument places has a condition on objects assigned by the partial assignment. A partial assignment is *appropriate* if it respects these constraints [12].

In situation semantics [18], soas play important roles in semantics of natural languages. So, the partial assignment is also important. Note that a partial assignment is just a record in our sense since it is a function, and hence a PST. Also a merge operation can be defined on the assignments so that the domain of partial assignments is isomorphic to the set of the records. Pollard [77] proposed *anadic relation* that are relations with no fixed arity. Anadic relation is an alternative to the state of affairs, that is, a relation whose arguments may not be fully saturated. It is straightforward to apply PSTs to represent both soas and anadic relations.

5.6.4 Remarks on Complex Indeterminates

The general form of a complex indeterminate is a basic indeterminate (parameter), x, with a constraint (condition), C, on x, written

$$x : C.$$

Complex indeterminates are basic materials in situation semantics [18] to construct various kinds of semantic objects expressed in language uses. The central idea of situation semantics is that context dependencies are an essential aspect of natural language uses. As suggested in Barwise and Perry [18], complex indeterminates can also play a role of dynamic cognitive objects such as discourse representation structures in the discourse representation theory of Kamp [42]. Accordingly, the first target was to introduce the complex indeterminate into logic programming, aiming at natural language analysis.

However, trying to give an appropriate inner representations of the complex indeterminate and to define an extended unification over them, we found that the standard term structures are not suitable into which complex indeterminates are coded. To illustrate the trouble, let $x\!:C(x,u,v)$ and $y\!: D(y,w,t)$ be two complex indeterminates, where u, v, w, and t are the "free" indeterminates appearing in the condition C or D or both as indicated. It was our policy that the extended unification rule for complex indeterminates should be as the simplest extension of the standard unification as possible. So, the unification for the two complex indeterminates above should go like this:

$$x : C(x,u,v) = y : D(y,w,t) \Longrightarrow x = y \wedge C(x,u,v) \wedge D(y,w,t).$$

There is, however, no way to identify free indeterminates u, v, w, and t with some of the others, for the list of free variables in a complex indeterminates is not given explicitly. For the more worse, even if the list of free variables is given as $[x,u,v]$ and $[y,w,t]$, for instance, it still remains a problem that there is no natural way to indicate the order of the indeterminates on the list in such a way that the standard list unification causes an intended unification of indeterminates on the list. To see this in the example, notice that $[x, v, u]$ is another possible list of free indeterminates, and unification may cause quite different effects from that for $[x, u, v]$.

Thereby it was noticed that the source of the trouble are the position-based fixed arity of the standard term structures. Then record structure came to us as a solution to the trouble. Now the complex indeterminates $x: C(x, u, v)$ and $y: D(y, w, t)$ of situation semantics is written

$$\{a/x, b/u, c/v\} : C(x, u, v)$$

and

$$\{a/x, b/w, c/t\} : D(y, w, t)$$

where a, b and c are appropriate "attribute names" given by the user. Thus, the problem was solved by extending the standard unification to record structures as we have seen in Chapter 3 and Chapter 4.

There is another historical remark. Introducing PSTs after fixing the trouble above, complex indeterminates were represented as a triple, $h(x, y, z)$, where h is a distinguished function symbol for parametric object, x is the object to be parameterized, y is a PST for the list of the parameters, and z is a condition on the parameters. An example looked like this:

$$h(x, \{age/y\}, (man(x) \wedge age(x, y) \wedge y \leq 30)).$$

Then, we noticed that, from implementation point of view, it is only convention to separate the prime and the other parameters. By introducing the conditioned term, we achieved more homogeneous representation of parametric objects. Now the above old example is simplified to

$$(\{self/x, age/y\}, (man(x), age(x, y), y \leq 31)).$$

Thus, the records represent complex indeterminates, parametric objects, and feature sets in the uniform way.

In the current version, a complex indeterminate is written $x : c$ or $x@c$, and is given the following interpretation rule:

$$a = (x : c) \Longrightarrow unify(a, x), solve(c).$$
$$a = (x@c) \Longrightarrow freeze(x, c), unify(a, x).$$

This rule covers only a small part of the aspects of complex indeterminates. We must leave it for the further interesting development of CIL.

5.6.5 Multiple Inheritance with Records

Records and the built-in *d_merge* in CIL apply to a simplified multiple-inheritance system. We first describe the problem using the language developed in Section 3.6.6. A *class system* (T, π) is an ordered pair of a set T of *classes* and a PST-valued function π defined on T. T is assumed to be partially ordered by $\leq$. The ordered set T represents an 'is-a' hierarchy and π gives prototype attributes for each class. Fix T and π. For $c \in T$, define $H(c) = \{d \in T \mid c \leq d\}^*$. The merge operation $(-)^*$ was defined in Example ex:merge. Define *create*(c) to be a *maximal* PST, p, such that $\pi(c) \sqsubseteq p \sqsubseteq H(c)$, where $\sqsubseteq$ is the record subsumption.

Now, let us take an example inheritance system $(\{bird, penguin, swallow\}, \pi)$ as follows:

$$\begin{array}{rcl} penguin & \leq & bird. \\ swallow & \leq & bird. \\ \pi(bird & = & \{can\_fly/yes\}. \\ \pi(penguin & = & \{can\_fly/no\}. \\ \pi(swallow) & = & \emptyset. \end{array}$$

This inheritance system can be written in the following CIL clauses.

$$\begin{array}{l} is\_a(penguin, bird). \\ is\_a(swallow, bird). \\ bird(\{can\_fly/yes\}). \\ penguin(\{can\_fly/no\}). \\ swallow(\_). \end{array}$$

By a simple calculation, each class has the attributes as follows.

$$\begin{array}{rcl} create(bird) & = & \{can\_fly/yes\} \\ create(penguin) & = & \{can\_fly/no\} \\ create(swallow) & = & \{can\_fly/yes\} \end{array}$$

Thus, we have a penguin instance with $\{can\_fly/no\}$ and a swallow with $\{can\_fly/yes\}$ as 'common sense reasoning' might expect. As *d_merge* collect default values, it is evident that a simple use of *d_merge* implements this multiple-inheritance specification.

5.6.6 Attitudes in PSTs

We show a simplified idea toward implementation of the attitudes theory in Barwise and Perry [18]. According to them, an attitude (mental state) is an ordered pair of a *frame* and a *setting*. A frame is a parametric type, and a setting is an assignment or anchor. Barwise and Perry [18] solves semantic paradoxes using this representations. The proposed data structure of mental states is close to that of the *closure* in LISP or the *molecule* in Prolog of structure sharing implementation. The record in CIL can be used for representation of the mental state. as illustrated below. For example, suppose the following two belief contexts:

(1) *Jack: I believe Taro beats Hanako.*
(2) *Betty: I believe Hanako beats Taro.*

We represent the mental states of (1) and (2) in (3) and (4), where *beater* and *beaten* are indeterminates.

(3) *believe(jack, {frame/beat(beater, beaten), beater/taro,beaten/hanako }).*
(4) *believe(betty, {frame/beat(beater, beaten), beater/hanako, beaten/taro }).*

We would like to say that basic unification and utilities on records gives a useful model to search for information in the mental state representation given that mental states are represented in records. We show this by giving queries to the above two beliefs.

(5) *Who believes taro is the beater?*
(6) $?\text{-}believe(x, \{beater/taro\}) \Longrightarrow x = jack.$

(7) *Who does jack believe is beaten?*

(8) $?\text{-}believe(jack, \{beaten/x\}) \Longrightarrow x = hanako.$

(9) *What does jack believe taro does?*

(10) $?\text{-}m = \{frame/z, a/taro\}, believe(jack, m) \Longrightarrow a = beater,\ z = beat(beater, beaten),$
$m = \{frame/beat(beater, beaten), beater/taro,\ beaten/hanako\}.$

Note that, in (10), this answer contains information *taro* is the *beater*.

5.6.7 A Tiny Discourse Analysis in CIL

The following is the program list in CIL for the example discussed in Section 5.6.1.

```
/* Abbreviations
        sit: situation
        sp:  beaker
        ip: interpretation
        hr: hearer
        dl: discourse location
        exp: expression
        soa: state of affairs
        ag: agent
        ob: object
        syncat: syntactical category
        head: head feature
        subcat: subcategorization */

% Discourse Situation
discourse_situation({sit/S, sp/I, hr/ You, dl/ Here, exp/ Exp}):-
        member(soa(speaking, (I, Here),yes),S),
        member(soa(addressing,(You, Here),yes),S),
        member(soa(utter,(Exp, Here),yes), S).

% Membership
member(X, [X|Y]).
member(X,[Y|Z]):-member(X,Z).

% Discourse Constraint
discourse_constraint([],[]):-!.
discourse_constraint([X],[Y]):-!,meaning(X,Y).
discourse_constraint([X,Y|Z], [Mx,My|R]):-
        meaning(X,Mx),
        turn_role(X, Y),
        time_precedent(X, Y),
        discourse_constraint([Y|Z],[My|R]).

turn_role({hr/X,sp/Y$}$,{hr/Y,sp/X}@discourse_situation).

time_precedent({dl/loc(X)},{dl/loc(Y)}):- constr(X+1=:=Y).

meaning(X#{exp/E},Y):-sentence(E-[],{ip/Y,ds/X}).

% DCG (Definite Clause Grammar)
sentence({ip/SOA,ds/DS})-->
        noun({ip/Ag,ds/ DS, syncat/{head/F}),
        verb({ip/SOA, ds/DS, ag/Ag, obj/ Obj,
syncat/{subcat/F}),
        noun({ip/Obj, ds/ DS}).

% Lexical Items
noun({ip/jack,
      syncat/{head/{minor/{agr/{plu/(-),per/3}@agr}}}@category})
        -->[jack].          % Jack
noun({ip/betty,
      syncat/{head/{minor/{agr/{plu/(-),per/3}@agr}}}@category})
        -->[betty].         % Betty
noun({ip/X,
      ds/{sp/X},
      syncat/{head/{minor/{agr/{plu/(-),per/1}@agr}}}@category})
        -->[i]              % I
noun({ip/X,
      ds/{hr/X},
      syncat/{head/{minor/{agr/{plu/@plu, per/2}@agr}}}@category})
        -->[you]            % you
verb({ip/ soa(love,(X, Y, Loc), yes),
      ds/ {dl/Loc},
      ag/ X,
      obj/Y,
      subcat/ {head/{minor/{agr/({plu/P, per/N}:
                              (P=(+), N= (@per);
                               P=(-), (N=1; N=2)))@agr}}}@category})
        --> [love].         % love

% Syntax  Categories
category({bar/ @bar, head/ @head}).
head( {major/ @major, minor/ @minor}).
major({n/ @n, v/ @v}).
minor({agr/ @agr, case/ @case}).
agr({per/ @per, plu/ @plu}).

case(accusative).
case(nominative).
bar(1).
bar(2).
bar(3).
n((+)).
n((-)).
v((+)).
v((-)).
plu((+)).
plu((-)).
per(1).
per(2).
per(3).
```

Chapter 6

Summary and Conclusion

First, CIL, an extension of Prolog, has been described. CIL has built-in record structures and an extended unification over them. CIL has freeze-based constraints for arithmetic, term, and Boolean domains. Record structures represent structured objects such as parametric objects of situation semantics. Various uses of records have been demonstrated for structured objects in linguistic analysis and knowledge representation. Thus, this work has shown a prototype system of logic programming for structured objects.

Second, we have proposed a partial algebra model, the record algebra, for feature structures and, in particular, partially specified terms, PSTs, in CIL. The record algebra is also a model for partiality of linguistic information and an extension of an existing model, the feature algebra. Viewing CIL as a constraint logic programming language over the record algebra, we have shown that CIL is an integration of the unification grammar and the definite clause grammar.

Finally, we have found a family of constraint languages over hypersets which are solution compact and satisfaction complete. The family includes Colmerauer's infinite tree unification with unequations and some of feature logics such as the record algebra. Declarative and operational semantics of the constraint logic programming over the domains have become essentially the same one seen as a form of coinductive definitions. Thus, this work puts a hyperset-theoretical foundation for the record-based languages such as unification grammars and object-oriented languages.

A decision problem in the hyperset domain remains open for the full class of constraints with equations, subsumptions, and their negations.

Bibliography

[1] P. Aczel. *Non-well-founded Sets*. CSLI lecture notes Number 14. CSLI Publications, Stanford University, 1988.

[2] P. Aczel. A theory of structured objects. September 1988. Unpublished draft, produced for the STASS workshop, Edinburgh.

[3] P. Aczel. Replacement systems and the axiomatisation of situation theory. In R.Cooper, K. Mukai, and J. Perry, editors, *Situation Theory and Its Applications*. CSLI Publications, Stanford University, 1990.

[4] P. Aczel and R. Lunnon. Universes and parameters. In J. Barwise, G. Plotkin, and J.M. Gawron, editors, *Situation Theory and Its Applications, volume II*. CSLI Publications, Stanford University, 1991. to appear.

[5] P. Aczel and N. Mendler. A final coalgebra theorem. In P.H. Pitts, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, number 389 in LNCS. Springer-Verlag, 1989.

[6] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. 1974.

[7] H. Aït-Kaci. *A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*. PhD thesis, Computer and Information Science, University of Pennsylvania, 1984.

[8] T. Amanuma, T. Suzuki, T. Okunishi, and K. Mukai. The CIL programming environment (in Japanese). In *The Proceedings of Second Annual Conference of Japanese Society for Artificial Intelligence*, 1988. Also appears in ICOT TM-534.

[9] J. Barwise. Branch points in situation theory. chapter 11. In Barwise[14].

[10] J. Barwise. Situations, facts, true propositions. chapter 10. In Barwise[14].

[11] J. Barwise. Situations, sets, and the axiom of foundation. chapter 8. In Barwise[14].

[12] J. Barwise. The situation in logic III: Situations, sets and the axiom of foundation. In Wilkiw et al, editor, *Proceedings of the 1984 Logic Summer School (Studies in Logic)*. Amsterdam: North-Holland, 1986. Also Report No. CSLI-85-26, Stanford: CSLI Publications.

[13] J. Barwise. Noun phrases, generalized quantifiers and anaphora. In P. Gärdenfors, editor, *Generalized Quantifier*. Dordrecht: Reidel, 1987.

[14] J. Barwise. *The Situation in Logic.* CSLI Lecture Notes 17. Stanford: CSLI Publications, 1989.

[15] J. Barwise and J. Etchemendy. Visual information and valid reasoning. In W. Zimmerman, editor, *Visualization in Mathematics.* Washington DC: Mathematical Association of America.

[16] J. Barwise and J. Etchemendy. *The Liar: An Essay on Truth and Circular Propositions.* Oxford Univ. Press, 1987.

[17] J. Barwise and L. Moss. Hypersets. manuscript, 1990.

[18] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, 1983.

[19] C. Chevalley. *Fundamental Concepts of Algebra.* Academic Press, 1956.

[20] G. Chierchia, B. H. Partee, and R. Turner, editors. *Properties, Types, and Meaning, II.* Kluwer Academic Publishers, 1989.

[21] A. Colmerauer. Prolog II: Reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Universite d'Aix-Marseille II, 1982.

[22] A. Colmerauer. Prolog and infinite trees. In S.A. Tarnlund, editor, *Logic Programming,* APIC Studies in Data Processing No. 16. Academic Press, 1982.

[23] A. Colmerauer. Equations and unequations on finite and infinite trees. In *Proceedings of the Second International Conference on Fifth Generation Computer Systems,* Tokyo, 1984.

[24] R. Cooper, K. Mukai, and J. Perry, editors. *Situation Theory and Its Applications.* Stanford: CSLI Publications, The University of Chicago Press, September 1990.

[25] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science,* 25:95–169, 1983.

[26] K Devlin. *Logic and Information, Volume I: Situation Theory.* Cambridge U.P., 1991 To appear.

[27] M. Dincbas, H. Simonis, and P. van Hentenryck. Extending equation solving and constraint handling in logic programming. Technical Report IR-LP-2203, ECRC, 1987.

[28] J. Dörre. Feature logic with weak subsumption constraints. Technical Report IWBS tech. Report 101, IBM Deutschland,Stuttgart, April 1990.

[29] J. Dörre and W. Rounds. On subsumption and semiunification in feature algebra. In *Proceedings of Logic in Computer Science,* 1990.

[30] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework, 1988. a lecture note at Stanford Univ.

[31] T. Fernando. On substitutional recursion over non-well-founded sets. In *Fourth Annual Symposium on Logic in Computer Science, Proceedings.* IEEE Computer Society Press, Washington, 1989.

[32] G. Gazdar, E. Klein, G.K. Pullum, and I.A. Sag. *Generalized Phrase Structure Grammar.* Cambridge: Blackwell, and Cambridge, Mass.: Harvard University Press, 1985.

[33] R. Goebel. The design and implementation of DLOG, a Prolog-based knowledge representation language. *New Generation Computing*, 3(4), 1985.

[34] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Partial and overloaded operators, errors and inheritance. Technical report, SRI International and CSLI, 1985.

[35] N. C. Heintze, J. Jaffar, C. S. Lim, S. Michaylov, P. J. Stuckey, R. Yap, and C. N. Yee. The CLP(R) programmers manual. Technical Report Technical Report 73, Dept. of Computer Science, Monash University, 1986.

[36] M. Höehfeld and G. Smolka. Definite relations over constraint languages. Technical Report LILOG-Report 53, Stuttgart: IBM Deutschland Gmbh, 1988.

[37] D. Israel and J. Perry. What is information? In *Proceedings of the Conference on Information, Language and Cognition, Vancouver, B.C.*, February 1988. to appear.

[38] S. Iyanaga and K. Kodaira. *Introduction to Modern Mathematics (I).* Tokyo: Iwanami Shoten, 1961. (in Japanese).

[39] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 1987.

[40] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *International Conference on Logic Programming*, 1987.

[41] M. Johnson. *Attribute Value Logic and the Theory of Grammar.* CSLI Lecture Notes 16. Stanford: CSLI Publications, University of Chicago Press, 1988.

[42] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk et al. editor, *Formal Method in the Study of Language*. Mathematical Center, 1981.

[43] R.T. Kasper and W. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the ACL, Columbia University.* ACL, 1986. New York, N.Y.

[44] F. Landman. Towards a theory of information: The status of partial objects in semantics. In *Foris-(Groningen-Amsterdam Studies in Semantics; 6).* Foris Publications – Dordrecht, 1986.

[45] S. Mac Lane. *Categories for the Working Mathematician.* Springer Verlag, 1971.

[46] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[47] (Manual) *Prolog III.* Luminy. Case 919–13288 Marseille Cedex 09-France: ProloGIA, 1990.

[48] R. Lunnon. Many sorted universes, srds, and injective sums. In J. Barwise, G. Plotkin, and J.M. Gawron, editors, *Situation Theory and Its Applications, volume II.* CSLI Publications, Stanford University, 1991. to appear.

[49] M. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees, 1988. draft.

[50] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[51] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[52] R. Montague. The proper treatment of quantification in English. In J. Hintikka, J. Moravcsik, and P. Suppes, editors, *Approaches to Natural Languages: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics.* Dordrecht: Reidel., 1970. Also appeared in *Formal Philosophy*, ed. R. Thomason. New York: Yale U.P. (1974).

[53] R. Montague. Pragmatics. In R. Thomason, editor, *Formal Philosophy.* New York: Yale U.P., 1974.

[54] L.S. Moss. Completeness theorems for logics of feature structures. In Yiannis N. Moschovakis, editor, *Proceedings of The MSRI Workshop on Logic From Computer Science.* Springer-Verlag, 1990. to appear.

[55] K. Mukai. Partially specified term in logic programming for linguistic analysis. In *Proceedings of International Conference on the Fifth Generation Computer Systems.* Institute for New Generation Computer Technology, 1988. (Also appears as ICOT-TM 568, 1988).

[56] K. Mukai. Coinductive semantics of Horn clause with compact constraint. Technical Report TR-562, ICOT, 1990.

[57] K. Mukai. Merge structures with an operator domain and its unification theory. *Journal of Japan Society for Software Science and Technology,* 7(2), 1990. (in Japanese, also appears as ICOT-TR 480, 1989).

[58] K. Mukai. A system of logic programming for linguistic analysis. Technical Report TR-540, ICOT, 1990. To appear also from SRI Tokyo series.

[59] K. Mukai. CLP(AFA): Coinductive semantics of horn clauses with compact constraints. In J. Barwise, G. Plotkin, and J.M. Gawron, editors, *Situation Theory and Its Applications, volume II.* CSLI Publications, Stanford University, 1991. to appear.

[60] K. Mukai. On synthesis of feature structures as constraint satisfaction problems over hypersets. Number Preprints, LA90–14(1991–2), February 1991. (in Japanese).

[61] K. Mukai. Record algebra model for feature structures. In J. Wedekind and C. Rohrer, editors, *Unification in Grammar.* MIT Press, 1991. to appear.

[62] K. Mukai. Unification in final coalgebras. Technical report, ICOT, 1991. in preparation.

[63] K. Mukai, T. Okunishi, and T. Amanuma. CIL language manual. Technical Report TM-242, ICOT, 1986.

[64] K. Mukai and H. Yasukawa. Complex indeterminates in prolog and its application to discourse models. *New Generation Computing,* 3(4), 1985.

[65] K. Mukai and H. Yasukawa. On the modelling discourse process based on type inference. In *The Proceedings of Sixth Annual Conference of Japan Society for Software Science and Technology,* 1989. (in Japanese).

[66] R. Muskens. Meaning and partiality. Amstel 292H NL 1017 AN Amsterdam, The Netherlands, 1989.

[67] R. Muskens. A relational formulation of the theory of types. *Linguistics and Philosophy*, 12, 1989.

[68] R. Muskens. Going partial in Montague grammar. In R. Bartsch, F.F.A.K. van Benthem, and P. van Emde Boas, editors, *Semantics and Contextual Expression*. Sixth Amsterdam Colloquium, Foris, Dordrecht, 1990.

[69] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6), 1987.

[70] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI Conference*, number 104 in Springer Lecture Notes in Computer Science. Springer, 1981.

[71] F.C.N. Pereira and S.M. Shieber. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, 1984.

[72] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis*. CSLI, 1987.

[73] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.

[74] G.D. Plotkin. Towards a model for situation theory. presented at Workshop on Semantic Issues on Human and Computer Languages, 1987.

[75] G.D. Plotkin. A theory of relations for situation theory, 1988. Paper presented at the Situation Theory and Information Processing Workshop, Edinburgh.

[76] G.D. Plotkin. Illative theory of relations. In N. Braisby and R. Cooper, editors, *Situation Theoretic Studies in Psychology, Language and Logic*, volume 3 of *Edinburgh Working Papers in Cognitive Science*. 1989. Appears also in Cooper[24].

[77] C.J. Pollard. Toward anadic situation semantics. Manuscript, 1985.

[78] C.J. Pollard and I.A. Sag. *Information Based Syntax and Semantics, Volume 1 Fundamentals*. CSLI Lecture Notes 13. Stanford: CSLI publications, 1987.

[79] W. Rounds. Set values for unification based grammar formalisms and logic programming. Technical Report Report No. CSLI-88-129, CSLI, 1988.

[80] K. Sakai and A. Aiba. Cal: Constraint logic programming language with Buchberger algorithm. Technical report, ICOT, 1988.

[81] Y. Sato, K. Sakai, and S. Menju. SetCAL a solver of set constraint in CAL. Technical report, ICOT, to appear.

[82] P. Sells. *Lectures on Contemporary Syntactic Theories: An Introduction to Government-Binding Theory, Generalized Phrase Structure Grammar, and Lexical–Functional Grammar*. CSLI Lecture Notes 3. CSLI, Stanford University, 1985.

[83] S.M. Shieber. *An Introduction to Unification–based Approaches to Grammar*. CSLI Lecture Notes No.4. CSLI publications, Stanford University, 1986.

[84] S.M. Shieber, F.C.N. Pereira, L. Karttunen, and M. Kay. A compilation of papers on unification-based grammar formalisms, Parts I and II. Technical Report CSLI-86-48, CSLI, April 1986.

[85] G. Smolka. Feature constraint logics for unification grammars. Technical Report IWBS report 93, IBM Deutschland GmbH, 1989.

[86] G. Smolka. Feature logic with subsorts. Technical Report LILOG Report 33, IWBS, IBM Deutschland, Postfach 80 08 80, 7000 Stuttgart 80, W. Germany, May 1989.

[87] G. Smolka and H. Aït-Kaci. Inheritance hierarchies: Semantic and unification. *Journal of Symbolic Computation*, 7, 1989.

[88] P. J. Stuckey. On the foundation of constraint logic programming. Technical report, Department of Computer Science Monash University, Victoria 3168, Australia, August 1987.

[89] R. Sugimura, H. Miyoshi, and K.Mukai. Constraint analysis on Japanese modification. In *the proceedings of natural language understanding and logic programming*. North Holland, 1987.

[90] J. van Benthem. The relational theory of meaning. *Jogique et Analyse*, 1986.

[91] J. van Benthem. Categorial grammar meets unification, 1988.

[92] J. van Benthem. Semantic parallels in natural language computation. In M. Garrido, editor, *Logic Colloquium, Granada 1987*. North-Holland, 1988. to appear.

[93] J. van Benthem. Categorical grammar and type theory. *Journal of Philosophical Logic (to appear)*, 1989.

[94] B.L. van der Waerden. *Algebra I, II*. Springer Verlag, 1955.

[95] H. Yasukawa and K. Yokota. Labeled graphs as semantics of objects. Technical report, ICOT, November 1990. SIGDBS& SIGAI of IPSJ.