Intensional Set Constraints

G. Richard - F. Saubion - A. Tellez-Arenas
LIFO, Université d'Orléans (France)
{ richard, saubion, tellez }@lifo.univ-orleans.fr

Abstract

Existing approaches to deal with sets in a CLP framework generally assume the finiteness of the subjacent sets: that is a reason why intensional sets are often forbidden. In this paper, we propose a new compilation method to treat intensional sets in a general setting. Our representation relies on a simply typed λ-calculus: sets are considered as Boolean-valued functions that may involve union and intersection. This set expression language can be manipulated through suitable rules added to a λ-calculus with explicit substitutions. Using the fact that we get a first order mechanism derived from a calculus with explicit substitutions, efficient implementations are intended taking advantage of existing specific abstract machines for such calculus. Our method is a way to overcome the standard finiteness restrictions concerning sets in *CLP* languages.

1 Introduction

Set constraints are naturally involved in many computer science areas and there is a large agreement about their value in programming languages. For instance, considering variables as sets of possible values for program analysis leads to systems of inclusion constraints [11, 12]. In functional programming, some type inference mechanisms use the notion of subtypes and thus, the associated typing algorithms are in fact set inclusion constraint systems ([3]). On another hand, an increasing number of works has been developed for embedding sets as a native data structure in high level programming languages, especially logic programming languages [7, 9, 14, 18]: their aim is to increase the expressive power of the host language. In [13] for instance, finite intensional sets are allowed : given a predicate definition containing intensional set variables, the resolution mechanism tries to give an explicit representation of these variables. Nevertheless, most previous works deals with finite sets represented by an enumeration of their elements, the so-called *extensional sets*. In the current implementations, data structures such as lists or arrays can be easily used. Specific procedures are needed to deal with the lack of ordering and with multiple occurrences of elements which are inherent in set-based definitions. But, as soon as one considers infinite sets, this representation is not available and one must turn to an implicit representation without enumeration.

Many kinds of such representations can be suggested: sets as solutions of fixed point equations, tree automata [10], etc... But, as is explained in [7], to keep close to the standard logic syntax, the most appropriate syntax for set expressions, inspired by a common practice in mathematics, is an *intensional* notation of the form $\{x \mid P(x)\}$ specifying the set of elements x with the property P, where P is generally defined using

first order logic. This kind of set expression, using an abstract set former $\{\cdots \mid \cdots\}$, becomes a logic term and is allowed to occur in any position where a standard term can occur. Let us examine a simple example, extracted from [16], to highlight our discussion. *A box is sterile if all the bugs in it are dead. A bug in a boiled box is dead. A box b is boiled.* The question is: "is b a sterile box ?" or "find the sterile boxes y". We argue that a natural way to formally translate this specification is to adopt a CLP-like formalism:

$sterile(y) \leftarrow bug\_in(y) \subseteq dead\_bug.$

$dead(x) \leftarrow bug(x),\ in(x, y),\ boiled(y).$

$boiled(b).$

where $bug\_in(y) = \{x \mid bug(x) \wedge in(x, y)\}$ and $dead\_bug = \{x \mid bug(x) \wedge dead(x)\}$. The inclusion $bug\_in(y) \subseteq dead\_bug$ is considered as a set constraint and we have to reduce it. But, of course, we have no information about the cardinality of the two sets involved in the constraint. Typically, none of the known works could deal with such a specification. For instance, [5] is based on a mechanism which constructs an intensional set by enumerating all its elements: the sets are assumed to be finite and abstract set formers are replaced by the corresponding extensional set terms. But, since the finiteness of a first order predicate is a well-known undecidable problem, either a careful programming style is necessary to avoid generation of infinite sets or the class of admissible programs is restricted using syntactic properties.

The main novelty in our paper is to reduce constraints without any assumption about the set cardinality. We propose an alternative representation of intensional sets together with a term rewriting system to solve constraints on these sets. Our main idea is to consider a predicate p, previously used to define the property characterizing the elements of an intensional set $\{x \mid p(x)\}$, as a boolean-valued function encoded as a λ-expression $\lambda x.p(x)$. Back to the sterile box example, we get (informally at this stage) a representation such as $bug\_in(y) \equiv \lambda x.bug(x) \wedge in(x, y)$ and $dead\_bug \equiv \lambda x.bug(x) \wedge dead(x)$. As in the standard approach, we deal with set expressions involving union, intersection and complement set operators. Then a with explicit substitutions [15, 1, 6, 8] provides a method to compute over λ-expressions by using first-order term rewriting system. We extend one of these systems, namely the $\lambda\sigma$-calculus [1, 6], with rules especially designed to handle intensional set definitions. We only consider two kinds of constraints: inclusion constraints between set expressions $se_1 \subseteq se_2$ (including by the way emptiness test) and membership constraints such as $x \in se$. Such constraints are embedded into a λProlog-like ([16]) framework, where functions and formulas can be treated with the same λ-calculus formalism. Such a framework extends classical constraint Horn clauses since universal quantifiers and implicative formulas become allowed goals. Concerning intensional sets, this is a necessary extension since an inclusion $\{x \mid p(x)\} \subseteq \{x \mid q(x)\}$ immediately leads to a goal $\forall x, p(x) \Rightarrow q(x)$, which is not a strict Prolog goal. For lack of space, we do not recall here the full λ-Prolog setting and we focus on the set expression manipulations.

2 Set representation

We present now the syntax associated with our choice: the intensional representation, using the constructor $\{\ldots \mid \ldots\}$, formalized here as a λ-expression using explicit substitutions.

2.1 Explicit Substitutions: a Brief Review

In classical λ-calculus [4], the substitution mechanism relies on an external formalism which is not at the same conceptual level than the reduction rules. To overcome this problem, new systems have been developed using an explicit treatment of the substitution process. These systems are expressed as sets of first-order rewrite rules which internalize the substitutions by the use of *closures*. Our approach is based on the system presented in [8] for simply typed lambda-calculus. We describe our system in a De Bruijn's setting [17] where natural numbers play the role of variables. To avoid confusion, we use underlined integers: $\underline{1}, \underline{2}, ...$ and the set of variables is denoted $\underline{\mathbb{N}}$. For instance, the λ-term $\lambda xyz.f(x,y,z)$ is represented by the term $\lambda\lambda\lambda f(\underline{3},\underline{2},\underline{1})$. Informally, the integer denoting a variable corresponds to the number of λ symbols one has to cross to reach this variable. In this context, a substitution is represented as a list built using the cons "." constructor . The identity substitution is denoted id. For instance, $a.b.id$ corresponds to the substitution: $\underline{1} \mapsto a; \underline{2} \mapsto b; \underline{3} \mapsto \underline{1}; \ldots; \underline{n+2} \mapsto \underline{n}$. We comply here with the notations of [1] and their calculus: $\lambda\sigma$. A particular substitution denoted $\uparrow$ is defined by: $\underline{1} \mapsto \underline{2}; \underline{2} \mapsto \underline{3}; \ldots$ and allows to consider $\underline{n}$ as a notation for $\underline{1}\underbrace{[\uparrow]...[\uparrow]}_{n-1}$ (of course, this is not a substitution in the usual first order setting where a finite support is assumed). The application of a substitution s to a term t is denoted $t[s]$: this is a *closure*, a term of a kind not present in the standard lambda-calculus. We give here the grammar and the rewrite rules for a calculus in the $\lambda\sigma$ ([1]) family. We consider typed terms and we begin with the grammar of types. Given any set of basic types $\mathcal{BT}$, types are built as follows :

- **Types** $A ::= K \mid A \to B$

where K is an element of $\mathcal{BT}$ and A and B are types. Contexts are lists of types, used to record types of free variables. Moreover, we introduce a set of high-order variables $\mathcal{X}$, considered as meta-variables, and independent from the β-reduction mechanism. A unique context $?_X$ and a unique type T_X is associated with each metavariable X in $\mathcal{X}$.

- **Contexts** : $? ::= nil \mid A.?$

We give then the typed $\lambda\sigma$-terms grammar :

- **Terms** : $a ::= \underline{1} \mid X \mid (a \;\; a') \mid \lambda_A a \mid a[s]$
- **Substitutions** : $s ::= Id \mid \uparrow \mid a.s \mid s \circ t$

where the composition operator is denoted $s \circ t$. We recall in the following tables the typed $\lambda\sigma$-calculus. Substitutions are introduced by the use of the $Beta$ rule, which is the $\lambda\sigma$ translation of the classical β-reduction, and are eliminated thanks to the twelve other rules. We use the notation $s \triangleright ?$ to express the fact that the substitution s has type $?$. $? \vdash a : K$ denotes the fact that a has the type K in the context $?$.

$\lambda\sigma$	
$Beta$	$(\lambda_A a \quad b) \rightarrow a[b.id]$
App	$(ab)[s] \rightarrow (a[s])(b[s])$
$VarCons$	$\underline{1}[a.s] \rightarrow a$
Id	$a[id] \rightarrow a$
Abs	$(\lambda_A a)[s] \rightarrow \lambda_A(a[\underline{1}.(s \circ \uparrow)])$
$Clos$	$a[s][t] \rightarrow a[s \circ t]$
IdL	$id \circ s \rightarrow s$
$ShiftCons$	$\uparrow \circ (a.s) \rightarrow s$
Ass	$(s_1 \circ s_2) \circ s_3 \rightarrow s_1 \circ (s_2 \circ s_3)$
IdR	$s \circ id \rightarrow s$
$Varshift$	$\underline{1}. \uparrow \rightarrow id$
$Scons$	$\underline{1}[s].(\uparrow \circ s) \rightarrow s$

Typing rules
$A.\Gamma \vdash \underline{1} : A \quad \Gamma_X \vdash X : T_X$
$\dfrac{A.\Gamma \vdash b : B}{\Gamma \vdash \lambda_A b : A \rightarrow B}$
$\dfrac{\Gamma \vdash a : A \rightarrow B \quad \Gamma \vdash a' : A}{\Gamma \vdash (a \quad a') : B}$
$\dfrac{\Gamma \vdash s \triangleright \Gamma' \quad \Gamma' \vdash a : A}{\Gamma \vdash a[s] : A}$
$\Gamma \vdash Id \triangleright \Gamma \quad A.\Gamma \vdash \uparrow \triangleright \Gamma$
$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash s \triangleright \Gamma'}{\Gamma \vdash a.s \triangleright A.\Gamma'}$
$\dfrac{\Gamma \vdash s'' \triangleright \Gamma'' \quad \Gamma'' \vdash s' \triangleright \Gamma'}{\Gamma \vdash s' \circ s'' \triangleright \Gamma'}$

Table 2.1

2.2 $\lambda\sigma$S-calculus

We present now our extension, $\lambda\sigma$S, of the $\lambda\sigma$-calculus. We first give the representation of predicates defining intensional sets, then syntax and rewriting rules associated with new set operators.

2.2.1 User's predicates

Clearly, an intensional set defined as $\{x \mid p(x)\}$ is based on the definition of the property p to be satisfied by its elements. In order to obtain a practical definition, we adopt a definition of predicates using a logic language with higher order features like λ-Prolog ([16]). A property can be described through a logic program, and the set $\{x \mid p(x)\}$ denotes the set of solutions of the query $p(x)$?. Back to the sterile box example, the set $dead\_bug$ is just the set of solutions of the query $bug(x) \wedge dead(x)$, where the predicates bug and $dead$ are defined by a logic program.

Given a set $\mathcal{F}$ of function symbols, a set $\mathcal{V}$ of first-order variables, $T(\mathcal{F}, \mathcal{V})$ denotes the set of terms constructed on $\mathcal{F}$ and $\mathcal{V}$, $T(\mathcal{F})$ denotes the set of ground terms. We work with possibly infinite intensional sets which are subsets of $T(\mathcal{F})$ or $\underbrace{T(\mathcal{F}) \times \ldots \times T(\mathcal{F})}_{n \ times}$ for a suitable integer n. Given a set $\mathcal{P}$ of predicate symbols, $Atom$ denotes the standard set of atomic formulas built with $\mathcal{P}$ and $T(\mathcal{F}, \mathcal{V})$. In the following, $\{(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n)\}$ will be an abbreviation for $\{(t_1, \ldots, t_n) \in T(\mathcal{F}) \times \ldots \times T(\mathcal{F}) \mid p(t_1, \ldots, t_n)\}$. Furthermore, in our formalism, since we use the De Bruijn notation, the set of variables $\mathcal{V}$ is $\underline{\mathbb{N}}$ and we will write equally $\mathcal{V}$ or $\underline{\mathbb{N}}$.

2.2.2 Set representation

The introduction of some boolean operations makes possible set manipulations such as union or intersection. We introduce two new specific atomic type constant denoted by E and U, respectively used for typing elements of $\mathcal{T}(\mathcal{F}, \underline{\mathbb{N}})$ and for typing user's predicates (typing rules are in table 2.2.2). The abstract grammar of intensional sets is the following, where se is a *set expression*, pe is a formula defining a set and p is an atom. We have to introduce a new set of variables $\mathcal{X}$, named *meta-variables* (denoted with capital letters),

which are *set variables*, different from $\mathcal{V}$ whose elements are first-order variables.

- $se ::= \underline{1} \mid \lambda_E se \mid \lambda_E pe \mid se \cup se' \mid se \cap se' \mid X \mid se[s]$
- $pe ::= pe \vee pe' \mid pe \wedge pe' \mid p \mid \top \mid \bot$

where s is a substitution following the syntax of $\lambda\sigma$, $\top$ and $\bot$ denote respectively true and false. Note that this syntax allows to express set depending from first-order variables, such as $bug\_in(y) = \{x \mid in(x,y) \wedge bug(x)\} = \lambda in(\underline{1},\underline{2}) \wedge bug(\underline{1})$, which can be instantiated by using the notation $se[s]$ into $bug\_in(b) = \{x \mid in(x,b) \wedge bug(x)\} = (\lambda in(\underline{1},\underline{2}) \wedge bug(\underline{1}))[b.id]$. In the following, we impose se to be free of meta-variables and reduce the term thanks to the $\lambda\sigma$S-calculus. We present then a set of rewrite rules to describe the interaction between the initial $\lambda\sigma$ terms and the new set operators introduced to express sets. The $\lambda\sigma$S-system is the union of $\lambda\sigma$ and of the *Set* system which shows how substitutions are applied to new $\lambda\sigma$S-terms.

Set	
App$\cup$	$(se \cup se')[s] \rightarrow se[s] \cup se'[s]$
App$\cap$	$(se \cap se')[s] \rightarrow se[s] \cap se'[s]$
$\lambda se \cap \lambda se' \rightarrow \lambda(se \cap se')$	$\lambda se \cup \lambda se' \rightarrow \lambda(se \cup se')$
$\lambda p \cap \lambda p' \rightarrow \lambda(p \wedge p')$	$\lambda p \cup \lambda p' \rightarrow \lambda(p \vee p')$

Typing rules	
$\dfrac{\Gamma \vdash se : A \quad \Gamma \vdash se' : A}{\Gamma \vdash se \cup se' : A}$	$\dfrac{\Gamma \vdash se : A \quad \Gamma \vdash se' : A}{\Gamma \vdash se \cap se' : A}$
$\dfrac{E.\Gamma \vdash p : U}{\Gamma \vdash \lambda_E p : E \rightarrow U}$	

Table 2.2.2

We may notice that, using the previous system, $((\lambda p \cap \lambda q) \ t)$ reduces to $(\lambda(p \wedge q)t)$, then to $(\lambda((p \wedge q)[t.id])$,then to $(\lambda(p[t.id] \wedge q[t.id])$. The rules to reduce $p[s]$ are similar to a first-order substitution process. Properties of $\lambda\sigma$S-system are the same ones as $\lambda\sigma$ properties ([8]) and the main result is that $\lambda\sigma$S is weakly normalizing and confluent on substitution-closed terms.

3 Constraints

By using λ-calculus for intensional sets representation and manipulation, we want to test membership and set inclusion without enumerating elements as is classically done in [5]. Note that other constraints such as equality or emptiness can be expressed using inclusion: $se = se'$ can be expressed by $se \subseteq se' \wedge se' \subseteq se$, whereas $se = \emptyset$ is equivalent to $se \subseteq \emptyset$.

3.1 Membership constraints

Given an intensional set S, a membership constraint $(t_1, \ldots, t_n) \in S$ will be expressed as an application $(\ldots(se \ t_1)\ldots \ t_n)$, where se denotes the abstraction encoding S, and $t_i \in T(\mathcal{F}, \mathcal{V})$. The main idea is to reduce the $\lambda\sigma S$-terms using the $\lambda\sigma S$-system and to solve the resulting formula w.r.t. the logic language defining predicates (λ-Prolog). But in fact, there are several cases: se could be ground or could contain meta-variables. If se contains meta-variables, then we cannot assert the consistency of the constraints, but in some cases, we can prove the inconsistency: it is sufficient to check the constraint after

replacing meta-variables by ground terms. For example, using the following definition of *even* predicate:

$even(zero)$.

$even(s(s(x))) :\perp even(x)$.

we cannot infer inconsistency from $s(zero) \in \{x \mid even(x)\} \cup X$, but $s(zero) \in \{x \mid even(x)\} \cap X$ is always false, that can be proved in replacing X by $T(\mathcal{F})$. In order to test the consistency of a ground membership constraints, it is sufficient to reduce it using the $\lambda\sigma$S-system, then test the validity of its normal form, w.r.t predicates definition. The resolution will generate a first-order substitution (on free variables), that we will propagate in the system. Of course, if the given query is ground, then the λ-Prolog resolution process acts only as a test. For example, checking $zero \in \{x \mid even(x)\}$ is solved by reducing $((\lambda x.even(x))\ \ zero)$, whose normal form is $even(zero)$ which is considered as the initial goal for a λ-Prolog interpreter. Thanks to the simply typed λ-calculus underlying our mechanism, we avoid some standard paradoxes using expression such as $X \notin X$.

3.2 Inclusion constraints

The main idea is to structure the solving mechanism in two distinct parts. The first part allows one to extract from a system of inclusion constraints a list of equations, from which simple questions such as membership, emptiness can be easily checked, and generate a new system of inclusion, but where all meta-variables have been removed. The consistency test of this new system is performed in a second part, with a mechanism like λ-prolog resolution, since inclusions of intensional sets can be transformed into logical formulas, built on $\Rightarrow$, $\forall$, $\wedge$, $\vee$, and predicates defined by logical program.

Definition 1 *Let P be a conjunction of set constraints of the form $se_1 \subseteq se_2$, where se_1 and se_2 are typed set expressions, and contain typed meta-variables $X_1, \ldots, X_p$. A solved form is a conjunction of equations of the form $X_i = T_i$, for $1 \leq i \leq p$, where T_i is an expression containing $\cap$, $\cup$,$-$ operators, closed set expressions, and no meta-variable of type with a depth greater than X_i (the depth n of a type T is the number of arrows which form it).*

Note that, if the complementary operator is not allowed in the grammar of set expressions, it may appear in an intermediary step, and will be denoted by $\overline{se}$. This solved form is completed by formulas without meta-variables, called consistency constraints and whose validity has to be checked by the system:

Definition 2 *A consistency constraint is a constraint of the form $\forall \ldots \forall (p_1 \wedge \ldots \wedge p_m \Rightarrow p'_1 \vee \ldots \vee p'_m)$ where p_i are user's predicates.*

The second part of our mechanism is the validity test of a system of such constraints. In order to insure termination of the algorithm, the next meta-variable to be resolved is the maximal one w.r.t. the depth of its type: each step eliminates the greatest meta-variable and does not add any new variable. We could prove that, if a given set is a subset of $\underbrace{T(\mathcal{F},\mathcal{V}) \times \ldots \times T(\mathcal{F},\mathcal{V})}_{n \text{ times}}$, then its type is of the form $\underbrace{E \rightarrow \ldots \rightarrow E \rightarrow U}_{n \text{ times}}$, and the depth is n. Our algorithm is an adaptation of the one presented in [2]. The main difference is that we deal with typed expressions, and different levels of constraints have to be treated. In [2], sets are subsets of $T(\mathcal{F},\mathcal{V})$, and tuples are not treated. Furthermore, we can deal

with sets depending from free variables (such as $bug\_in(y)$ in the sterile box example), and universal quantifiers $\forall$ are introduced, for instance in $\forall y(bug\_in(y) \subseteq dead\_bug(y))$. These quantifiers are internal and do not appear neither in the initial constraint system nor in the final one.

Simplification	$\dfrac{P \wedge \forall \ldots \forall(se_1 \subseteq se_2 \cap se_3)}{P \wedge \forall \ldots \forall(se_1 \subseteq se_2) \wedge \forall \ldots \forall(se_1 \subseteq se_3)}$ $\dfrac{P \wedge \forall \ldots \forall(se_1 \cup se_2 \subseteq se_3)}{P \wedge \forall \ldots \forall(se_1 \subseteq se_3) \wedge \forall \ldots \forall(se_2 \subseteq se_3)}$ $\quad$ $\dfrac{P \wedge se_1 \subseteq se_2}{P \wedge se_1 \downarrow_{\lambda\sigma S} \subseteq se_2 \downarrow_{\lambda\sigma S}}$
Decomposition	$\dfrac{P \wedge \forall_1 \ldots \forall_n(\lambda se_1 \subseteq \lambda se_2)}{P \wedge \forall_1 \ldots \forall_n \forall_{n+1}(se_1 \subseteq se_2)}$
$\forall$-**Elim**	$\dfrac{P \wedge \forall_1 \ldots \forall_{n_j}(b_j \subseteq X_i)}{P \wedge ((\cup_1 \ldots \cup_{n_j} b_j) \subseteq X_i)}$ $\quad$ $\dfrac{P \wedge \forall_1 \ldots \forall_{n_j}(X_i \subseteq b_j)}{P \wedge (X_i \subseteq (\cap_1 \ldots \cap_{n_j} b_j))}$
X-**Elim**	$\dfrac{P \wedge \forall \ldots \forall(se_1 \cap X_i \cap se_2 \subseteq se_3)}{P \wedge \forall \ldots \forall(X_i \subseteq \overline{se_1} \cup \overline{se_2} \cup se_3)}$ $\quad$ $\dfrac{P \wedge \forall \ldots \forall(se_1 \subseteq se_2 \cup X_i \cup se_3)}{P \wedge \forall \ldots \forall(se_1 \cap \overline{se_2} \cap \overline{se_3} \subseteq X_i)}$
complement-**Elim**	$\dfrac{P \wedge \forall \ldots \forall(se_1 \cap \overline{se} \cap se_2 \subseteq se_3)}{P \wedge \forall \ldots \forall(se_1 \cap se_2 \subseteq se \cup se_3)}$ $\quad$ $\dfrac{P \wedge \forall \ldots \forall(se_1 \subseteq se_2 \cup \overline{se} \cup se_3)}{P \wedge \forall \ldots \forall(se_1 \cap se \subseteq se_2 \cup se_3)}$
Final decomposition	$\dfrac{P \wedge \forall_1 \ldots \forall_n(\lambda se_1 \subseteq \lambda se_2)}{P \wedge \forall_1 \ldots \forall_n \forall_{n+1}(se_1 \subseteq se_2)}$ $\dfrac{P \wedge \forall_1 \ldots \forall_n(\lambda pe_1 \subseteq \lambda pe_2)}{P \wedge \forall_1 \ldots \forall_n \forall_{n+1}(pe_1 \Rightarrow pe_2)}$ $\quad$ $\dfrac{P \wedge \forall_1 \ldots \forall_n(\lambda se_1 \subseteq pe_2)}{Fail}$

Table 3.2

- **First step:** Iterate the three simplification rules. *Note that $\forall$ quantifiers do not appear in the initial system, but after elimination step.*
- Iterate the decomposition rule. *Indeed, intuitively, $\{(x,y) \mid p(x,y,z)\} \subseteq \{(x,y) \mid q(x,y)\}$ is equivalent to $\forall x(\{y \mid p(x,y,z)\} \subseteq \{y \mid q(x,y)\})$. Note that we use the De Bruijn notation in such a way that in the expression $\forall \lambda p(\underline{1},\underline{2}) \subseteq q(\underline{1},\underline{2})$, $\underline{1}$ is bounded with the λ, and $\underline{2}$ with $\forall$.*
- Iterate the simplification rules.
- *Now the system is a conjunction of constraints of the form $\forall_1 \ldots \forall_n(X_1 \cap \ldots \cap X_n \cap se_1 \subseteq X'_1 \cup \ldots \cup X'_m \cup se_2)$, where se_i do not contain any meta-variables λ-free.* Choose the greatest meta-variables $X_i \in \mathcal{X}$ with regard to the depth criterium (i.e. $\forall X_j \in \mathcal{X}, ((X_j \neq X_i) \Rightarrow (depth(X_j) < depth(X_i)$ *or* $(depth(X_j) = depth(X_i) and i > j)$.
- In order to obtain X_i as a single variable on the left hand side or the right hand side of an equation, apply the X-elim rules ($\overline{b}$ denotes the complementary of b).
- *Constraints containing X_i are now of the form $\forall_1 \ldots \forall_{n_j}(X_i \subseteq b_j)$ or $\forall_1 \ldots \forall_{n_j}(b_j \subseteq X_i)$, where b_j contains only meta-variables smaller than X_i.* Apply the $\forall$-elim rules, where we use the De Bruijn notation in the same way that with λ and $\forall$. Now, we can write: $X_i = (\bigcup_{(a \subseteq X_i)} a) \quad \cup \quad (Y_i \cap \bigcap_{(X_i \subseteq a')} a')$ where Y_i is a new meta-variable.
- Apply the following procedure which apply the transitivity of inclusion operator, in order to transform P in an equivalent system on variables smaller than X_i:

for all $(X_i \subseteq (\cup_1 \ldots \cup_{n_j} b_j))$ in P do
  for all $((\cap_1 \ldots \cap_{n'_j} b'_j) \subseteq X_i)$ in P do
    add to P the constraint $\forall_1 \ldots \forall_{n_j+n'_j}(b'_j[\underline{1}.\underline{2}.\ldots.\underline{n'_j}.\uparrow^{n_j}] \subseteq b_j[\uparrow^{n'_j}])$ od
  remove $(X_i \subseteq (\cup_1 \ldots \cup_{n_j} b_j))$ from P od

The substitution is a simple renaming, and we reduce then $X[s]$ in X.

- In order to suppress the negations, apply the *complement*-elim rules.
- *We have obtain a new smaller system denoted by P', where we have removed the metavariable X_i.* If meta-variables appear in P', then go to the **first step**.
- *We finally obtain a system of equations which is the solution of the problem if it exists, and a system of constraints, of the form $\forall \ldots \forall(se_1 \subseteq se_2)$, where se_i contains no*

meta-variables. In order to obtain consistency constraints, iterate simplification rules, then final decomposition rules. *If the number of λ is different between right and left side of the inclusion, then the constraint is ill-typed.* □

Of course, all along this mechanism, we implicitly apply over each side of equations, simplification rules like $X \cup X \rightarrow X, \ldots$. We get now a set of equations which is the solved form of the initial problem, and a consistent set of implications if and only if the initial system is consistent. We can also have a set of membership constraints: in that case, we have to replace meta-variables X_i on the right hand side of a membership constraint, by T_i, according to their order w.r.t the depth of their respective types. When there are no meta-variables left, we use the mechanism presented in section 3.1. In order to check the validity of the system combining atoms (membership constraints) and implications (inclusion constraints), we use a λ-Prolog interpreter. For instance, it is straightforward to test this mechanism over the sterile box example: starting from the query $sterile(b)$?, we get *yes* and starting from the query $sterile(y)$?, we get as answer the first order substitution $y = b$. This means that, during the resolution process, the inclusion constraint $bug\_in(y) \subseteq dead\_bug$ involving a set parameterized by the free variable y, we instantiate the variable to get a solution. Then, thanks to the λ-Prolog mechanism, we are able to handle sets depending on free-variables and to give answers in some cases. Of course, it remains to define a class of sets where such manipulations always terminate i.e. a class of intensional sets where the problem we deal with is decidable. To identify such a class might be easier in $\lambda\sigma$-calculus than in the CLP framework. In appendix , we give an example where we see how our algorithm works.

4 Related works, future works and conclusion

In this paper, we address the problem of finite representation of infinite sets and we focus on the so-called *intensional sets*. In the field of constraint logic programming, we may mention the works of [5, 7]: in their approach, sets are assumed to be finite and thus the compilation method consists in replacing abstract set-formers by the corresponding extensional set terms. To effectively implement such an approach, the host language is supposed to provide a *collect-all* mechanism (which in fact could be implemented using negation): we do not assume such a facility to be provided.

Starting with the intuition that intensional sets could be considered as higher order objects, we have developed a scheme to represent such sets with specific lambda-terms. Our set expressions are built from set variables, function symbols and some operators: union, intersection, complement and an abstract set former $\{\cdots \mid \cdots\}$ for intensional sets. Thus, starting from $\lambda\sigma$, a calculus with explicit substitutions, we extend its grammar with set operators and the reduction system with specific rules to capture set semantics. Keeping the standard properties expected from such a calculus (local confluence, weak termination), our solving mechanism is derived from a first order unification algorithm w.r.t. the theory $\lambda\sigma S$. This system includes some standard inference rules in set constraints solving and provides an uniform treatment for membership, emptiness and inclusion constraints. The host language of such a system is a λ-Prolog like language where the semantics is defined via a natural deduction system.

Among the various extensions we have in mind, two ones seem to be easily obtainable. First to include new set formers like $f(X)$ to denote the set $\{x \mid x = f(y) \text{ for a } y \in X\}$), Secondly, to allow recursive set expressions such as $Even = \{0\} \cup \{Even + 2\}$, which is also a way to represent infinite sets. Of course, it is easy to incorporate treatment

of finite extensional sets since a singleton set $\{a\}$ is just represented with $\lambda x.equal(x, a)$ where *equal* predicate is defined with the unique rule $equal(x, x)$. A finite set is just a finite union of such singleton sets. Concerning nested sets, our mechanism might allow to deal with them without specific difficulty because of the higher order features included in λ-Prolog, but we have not yet explored this issue. From an operational point of view, we are currently developing a prototype written in a high level framework (ELAN ([19])) well-suited to experiment various strategies. In order to get an efficient mechanism, we could also take advantages of various abstract machines ([15]) defined for lambda-calculus with explicit substitutions.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 4(1):375–416, 1991.

[2] A. Aiken and E.L. Wimmers. Solving Systems of Set Constraints. In 7^{th} *Symposium on LICS*, 1992.

[3] A. Aiken and E.L. Wimmers. Type Inclusion Constraints and Type Inference. In *Conference on Functional Programming Language and Computer Architecture*, pages 31–42. ACM press, 1993.

[4] H.P. Barendregt. *The Lambda-Calculus and its Syntax and Semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Sciences, north-holland edition, 1984. Second Edition.

[5] P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling Intensional Sets in CLP. In P. Van Entenryck, editor, *Logic Programming :* Proceedings of the eleventh International Conference, pages 647–661. MIT Press, june 1994.

[6] P.-L. Curien, T. Hardin, and A. Ríos. Normalisation Forte du Calcul des Substitutions. Technical Report 16, LIENS, 1991.

[7] A. Dovier, E Omodeo, E Pontelli, and G. Rossi. {log} : A Language for Programming in Logic with Finite Sets. In *Journal of Logic Programming*, volume 28, pages 1–44, july 1996.

[8] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 366–374, june 1995.

[9] C. Gervet. Conjuncto: Constraint Logic Programming with Finite Set Domains. In *ILPS'94*, November 1994.

[10] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated susbset relationships. Technical Report 247, Laboratoire d'Informatique Fondamentale de Lille, 1993.

[11] N. Heintze. *Set Based Progam Analysis.* PhD thesis, Carnegie Mellon University, 1992.

[12] N. Heintze and J. Jaffar. A Decision Procedure for a Class Herbrand Set Constraints. In *Proceedings of the* 5^{th} *Symposium on Logic in Computer Science*, pages 42–51, Philadelphia, June 1990.

[13] P. M. Hill and John W. Lloyd. The Gödel Programming Language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992.

[14] D. Kozen. Set Constraint and Logic Programming. In *CCL95*, 1995.

[15] P. Lescanne. From $\lambda\sigma$ to $\lambda\nu$ a journey through calculi of explicit substitutions. In Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL), pages 60–69, 1994.

[16] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[17] De Bruijn N. Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 5(34):381–392, 1972.

[18] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets, an introduction to SETL*. Springer-Verlag, 1986.

[19] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes*. PhD thesis, These de l'Universite de Nancy 1, Novembre 1994.

A Example

Let p, q, r and t be predicates symbols defined by:

$$\begin{cases} p(f(x,y)). \\ q(f(x,x),x). \\ r(x,h(x,x)). \\ t(g(x,x),x). \end{cases}$$

We want to test consistency and to solve the following set of constraints, where the typed meta-variables are $(X_1, E \to U)$, $(X_2, E \to U)$, and $(X_3, E \to E \to U)$:

$$\begin{cases} (1) & X_1 \cap \lambda p(\underline{1}) \subseteq \lambda q(\underline{1},\underline{2}) \\ (2) & \wedge \quad \lambda X_1 \cup \lambda\lambda r(\underline{2},\underline{1}) \subseteq \lambda X_2 \\ (3) & \wedge \quad X_3 \subseteq \lambda\lambda q(\underline{2},\underline{1}) \cup \lambda X_1 \\ (4) & \wedge \quad \lambda\lambda r(\underline{2},\underline{1}) \subseteq X_3 \cup \lambda\lambda t(\underline{2},\underline{1}) \end{cases}$$

whose meaning is: (x_1 is a free first-order variable)

$$\begin{cases} (1) & X_1 \cap \{x \mid p(x)\} \subseteq \{x \mid q(x,x_1)\} \\ (2) & \wedge \quad \{(x,y) \mid y \in X_1\} \cup \{(x,y) \mid r(x,y)\} \subseteq \{(x,y) \mid y \in X_2\} \\ (3) & \wedge \quad X_3 \subseteq \{(x,y) \mid q(x,y)\} \cup \{(x,y) \mid y \in X_1\} \\ (4) & \wedge \quad \{(x,y) \mid r(x,y)\} \subseteq X_3 \cup \{(x,y) \mid t(x,y)\} \end{cases}$$

The following table shows the progress, of the first part of our solving mechanism:

$(1) \quad \wedge \quad (2) \quad \wedge \quad (3) \quad \wedge \quad (4)$
$(1) \quad \wedge \quad (21) : \lambda X_1 \subseteq \lambda X_2 \quad \wedge \quad (22) : \lambda\lambda r(\underline{2},\underline{1}) \subseteq \lambda X_2 \quad \wedge \quad (3) : X_3 \subseteq \lambda(\lambda q(\underline{2},\underline{1}) \cup X_1) \quad \wedge \quad (4)$
$(1) \quad \wedge \quad (21) : \forall(X_1 \subseteq X_2) \quad \wedge \quad (22) : \forall(\lambda r(\underline{2},\underline{1}) \subseteq X_2) \quad \wedge \quad (3) \quad \wedge \quad (4)$
$solve\ X_3$
$(1) \quad \wedge \quad (21) \quad \wedge \quad (22) \quad \wedge \quad (3) \quad \wedge \quad (4) : \lambda\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda\lambda t(\underline{2},\underline{1})} \subseteq X_3$
$X_3 = (\lambda\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda\lambda t(\underline{2},\underline{1})}) \cup (Y_3 \cap (\lambda(\lambda q(\underline{2},\underline{1}) \cup X_1))$
$(1) \quad \wedge \quad (21) \quad \wedge \quad (22) \quad \wedge \quad (34) : \lambda\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda\lambda t(\underline{2},\underline{1})} \subseteq \lambda(\lambda q(\underline{2},\underline{1}) \cup X_1)$
$(1) \quad \wedge \quad (21) \quad \wedge \quad (22) \quad \wedge \quad (34) : \lambda\lambda r(\underline{2},\underline{1}) \subseteq \lambda\lambda t(\underline{2},\underline{1}) \cup \lambda(\lambda q(\underline{2},\underline{1}) \cup X_1)$
$(1) \quad \wedge \quad (21) \quad \wedge \quad (22) \quad \wedge \quad (34) : \lambda\lambda r(\underline{2},\underline{1}) \subseteq \lambda(\lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1}) \cup X_1)$
$(1) \quad \wedge \quad (21) \quad \wedge \quad (22) \quad \wedge \quad (34) : \forall(\lambda r(\underline{2},\underline{1}) \subseteq \lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1}) \cup X_1)$
$solve\ X_2$
$(1) \quad \wedge \quad (21) : (\cup X_1) \subseteq X_2 \quad \wedge \quad (22) : (\cup\lambda r(\underline{2},\underline{1})) \subseteq X_2 \quad \wedge \quad (34)$
$X_2 = (\cup X_1) \cup (\cup\lambda r(\underline{2},\underline{1})) \cup Y_2$
$(1) \quad \wedge \quad (34)$
$solve\ X_1$
$(1) : X_1 \subseteq \overline{\lambda p(\underline{1})} \cup \lambda q(\underline{1},\underline{2}) \quad \wedge \quad (34) : \forall(\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1})} \subseteq X_1)$
$(1) \quad \wedge \quad (34) : (\cup(\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1})}) \subseteq X_1)$
$X_1 = (\cup(\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1})}) \cup (Y_1 \cap \overline{\lambda p(\underline{1})} \cup \lambda q(\underline{1},\underline{2})$
$(134) : \forall(\lambda r(\underline{2},\underline{1}) \cap \overline{\lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1})} \subseteq \overline{\lambda p(\underline{1})} \cup \lambda q(\underline{1},\underline{3}))$
$(134) : \forall(\lambda r(\underline{2},\underline{1}) \cap \lambda p(\underline{1}) \subseteq \lambda t(\underline{2},\underline{1}) \cup \lambda q(\underline{2},\underline{1}) \cup \lambda q(\underline{1},\underline{3}))$
$(134) : \forall(\lambda(r(\underline{2},\underline{1}) \wedge p(\underline{1})) \subseteq \lambda(t(\underline{2},\underline{1}) \vee q(\underline{2},\underline{1}) \vee q(\underline{1},\underline{3})))$
$(134) : \forall\forall(r(\underline{2},\underline{1}) \wedge p(\underline{1}) \Rightarrow t(\underline{2},\underline{1}) \vee q(\underline{2},\underline{1}) \vee q(\underline{1},\underline{3}))$

The meaning of the solution is:

$$\begin{cases} X_3 = \{(x,y) \mid r(x,y)\} \cap \overline{\{(x,y) \mid t(x,y)\}} \cup (Y_3 \cap \{(x,y) \mid q(x,y) \vee y \in X_1\} \\ X_2 = X_1 \cup \bigcup_{x \in T(\mathcal{F})} \{y \mid r(x,y)\} \cup Y_2 \\ X_1 = \bigcup_{x \in T(\mathcal{F})} (\{y \mid r(x,y)\} \cap \overline{\{y \mid t(x,y)\} \cup \{y \mid q(x,y)\}}) \cup \\ \qquad\qquad (Y_1 \cap \overline{\{x \mid p(x)\}} \cup \{x \mid q(x,x_1)\}) \end{cases}$$

with the consistency constraint:

$$\forall x \forall y (r(x,y) \wedge p(y) \Rightarrow t(x,y) \vee q(x,y) \vee q(y,x_1))$$

which is true, since $(r(x,y) \wedge p(y))$ is always false.