A Formal Approach to Deductive Synthesis of Constraint Logic Programs

Kung-Kiu Lau[1]
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

Mario Ornaghi
Dipartimento di Scienze dell'Informazione
Universita' degli studi di Milano, Via Comelico 39/41, Milano, Italy
ornaghi@hermes.mc.dsi.unimi.it

Abstract

Formal program synthesis is concerned with deriving programs that are (formally) correct wrt their specifications, and is therefore of crucial importance to formal development of verifiably correct software. Declarative paradigms such as logic programming are particularly suitable for program synthesis, and we have formulated a formal approach to synthesising standard logic programs. Our approach is object-oriented in the sense that it allows the synthesis of modular, reusable, and hence maintainable programs. In this paper, we propose a formal approach to synthesising constraint logic programs based on our current work. We will show that this approach can be used to synthesise modular and reusable CLP programs. An integral part of this paper is the introduction of an abstract semantics to reason about the synthesis of CLP programs.

1 Introduction

Program synthesis is concerned with deriving programs that are *correct* wrt their specifications. A formal approach to synthesis guarantees the formal correctness of the derived programs. This means that formal program synthesis will be at the heart of any system for developing verifiably correct software, in particular software for safety-critical applications. For practical software engineering, however, we need techniques that can synthesise modular, reusable, and hence maintainable software.

Declarative paradigms such as logic programming provide particularly suitable formalisms for program synthesis. Indeed in [13], Lloyd cites synthesis as 'the ultimate goal' of declarative programming. We share this view, and have formulated a formal approach to the synthesis of standard logic programs. Our approach can indeed be used to synthesise modular and reusable (and hence maintainable) programs. Moreover, the formal framework that underlies our approach is very general, and it should be equally applicable to the synthesis of *constraint logic programs* [8, 9]. In this paper, we propose a formal approach to deductive synthesis of CLP programs, based on our work in standard LP.

Our approach to synthesis in standard LP is *deductive* [10, 12, 11], i.e. it iteratively derives clauses for a program from its specification. The *synthesis process* starts from a *specification* that is defined in the context of a (first-order) theory

[1]The first author was partially supported by the European Union HCM project on Logic Program Synthesis and Transformation, contract no. 93/414, and by a grant from Consiglio Nazionale delle Ricerche (CNR), Italy, and the British Council, Rome.

that characterises all the knowledge of the problem domain. (We call such a theory a *specification framework*, denoted by $\mathcal{F}$). The synthesis process then must employ a *synthesis method*[2] to derive a program P that is totally correct wrt the specification. This means that the completion of P, $Comp(P)$, is a sub-theory of the specification framework $\mathcal{F}$. Thus the synthesis process can be seen as deriving a *small* theory $Comp(P)$ from a *big* theory $\mathcal{F}$ of which it is a part, in such a way that the axioms of the small theory can be run as a logic program.

More interestingly, from the point of view of object-oriented synthesis, specification frameworks may be *open*, i.e. not completely defined and hence parametric. In such frameworks, we can synthesise *steadfast*, or modular and reusable, programs [12]. A steadfast program can be correctly (re)used in a class of computation domains. Therefore steadfastness provides a basis for modular programming.

Furthermore, specification frameworks (both open and closed) can be constructed, and thus maintained, in an incremental manner by using the result of synthesis itself, as we showed in [11]. Thus, altogether our approach can provide a basis for object-oriented LP software engineering.

In this paper, we extend our approach to the synthesis of CLP programs. We define specification frameworks so that they can contain *constraint domains* or constraint theories, and develop an abstract (isoinitial) semantics for specification frameworks and constraint theories (Section 3). The semantics that we used in our previous work in standard LP becomes a particular case.[3]

We use our semantics to define the meaning of specifications of CLP programs and program correctness, and explain how our general approach to synthesis in LP can be soundly applied to CLP (Section 4). In particular, we will show that we can synthesise *steadfast* CLP programs. In this paper, however, we do not consider synthesis methods for CLP programs. Such methods are a research topic in their own right, which we intend to pursue in future.

To begin with, we describe specification frameworks that we have been using for standard LP, and the meaning of synthesis in these frameworks (Section 2).

2 Specification Frameworks for Standard LP

In our work on the synthesis of standard logic programs, a specification framework $\mathcal{F}$ is a first-order theory that axiomatises a general knowledge domain. $\mathcal{F}$ should be a sufficiently rich theory for specifying different problems on the domain. It should therefore define not only the abstract data types (ADT's) involved, but also their salient properties, e.g. induction axioms, for reasoning purposes.

Furthermore, $\mathcal{F}$ should have a good semantics to allow us to reason precisely and soundly about specifications, synthesis and programs. We have used *isoinitial semantics* [1] since we believe it is the right choice for this purpose. In this section, we give a brief description of the main features of specification frameworks with isoinitial semantics. More details can be found in [11].

We distinguish between *closed* and *open* frameworks. Intuitively a closed framework completely characterises a unique intended model (under isoinitial semantics), whereas an open framework has a class of intended models. However, in this paper, for lack of space, we will concentrate on closed frameworks, and only briefly consider open frameworks in Section 5.

[2] A survey of such methods can be found in [5].

[3] Our semantics is a contribution to the semantics of CLP programs. In this paper, however, we will consider only constraint domains without limit elements.

2.1 Closed Frameworks

A first-order *theory* $\mathcal{T}_\Sigma$ with *signature* Σ is a set of closed formulas of the (first-order) *language* L_Σ generated by Σ. We shall assume that the *identity* '$=$' always belongs to Σ, and that the usual *identity axioms* belong to $\mathcal{T}_\Sigma$. Normally, Σ will be understood and we will omit it.

A *closed* framework $\mathcal{F}$ is a (first-order) theory with the following key properties:
• *Freeness*. A subset of the constant and function symbols of $\mathcal{F}$, which we shall call the *construction symbols*, satisfy the *freeness axioms* [14].
• *Reachability*. There is at least one model of $\mathcal{F}$ reachable by the construction symbols. This means that every element of the domain of the model is represented by a ground term using the construction symbols. We will call such a term a *construction*.
• *Atomic completeness*. For every *closed atomic* formula A, $\mathcal{F} \vdash A$ or $\mathcal{F} \vdash \neg A$.
Reachability and atomic completeness ensure that a closed framework completely characterises a unique intended model under *isoinitial semantics*. Reachability by construction symbols together with freeness guarantee that the domain of the intended model is the Herbrand universe generated by the construction symbols.

In isoinitial semantics, the intended model of a theory $\mathcal{F}$ is the *isoinitial model*[4] (when it exists), namely a model I of $\mathcal{F}$ such that for every other model M of $\mathcal{F}$, there is a unique *isomorphic embedding* (see e.g. [3]) of I in M. This semantics is thus similar to the more well-known *initial semantics* popularly used in the algebraic approach to ADT's.[5] An *initial model* J of a theory $\mathcal{F}$ is such that for every other model M of $\mathcal{F}$, there is a unique *homomorphism* of J in M. In other words, whereas isoinitial semantics uses isomorphic embeddings, initial semantics uses homomorphisms.

From a semantics point of view, isoinitial semantics has the advantage that negation of atomic formulas behaves properly. This is due to the fact that isomorphic embeddings preserve relations and their negations.

Furthermore, isoinitial semantics has the advantage that we have the following criterion for the existence of isoinitial models of *full* first-order theories (see [1]):

Theorem 2.1 If a theory $\mathcal{F}$ has at least one reachable model, then $\mathcal{F}$ admits an isoinitial model if and only if it is atomically complete.

An analogous criterion for initial models (of full first-order theories) cannot be given, even though an initial model always exists for any algebraic or Horn theory.

Example 2.1 Consider the well-known theory $\mathcal{PA}$ of Peano Arithmetic, with signature containing '0', 's', '$+$', '$\cdot$', and '$=$'. Here, only '0' and 's' are *construction symbols*. $\mathcal{PA}$ can be used as a closed specification framework for specifying and reasoning about standard logic programs that compute over natural numbers.

Its construction symbols satisfy the freeness axioms. $\mathcal{PA}$ is reachable because every natural number n can be represented by the numeral $s^n 0$. It is atomically complete, since, for every closed atomic formula $t_1 = t_2$, either $\mathcal{PA} \vdash t_1 = t_2$ or $\mathcal{PA} \vdash \neg t_1 = t_2$. The isoinitial model of $\mathcal{PA}$ is just the standard structure of natural numbers $\mathcal{N}$. $\diamond$

2.2 Reasoning about Synthesis

In a given specification framework $\mathcal{F}$, we can specify many different problems, i.e. relations to be computed.

[4] More precisely, all isoinitial models are isomorphic, and the intended model is one such model.
[5] See [15] for a survey, and [1] for an extensive comparison of the two kinds of semantics.

A *specification* consists of a *definition axiom* D_r of the form $\forall x(r(x) \leftrightarrow R(x))$, and a set $G = \{r(\tau_1), \ldots, r(\tau_j), \ldots\}$ of *admissible goals*, where:
• r is a new predicate symbol that denotes the relation to be computed, and R is a formula defining the intended meaning of r in $\mathcal{F}$;
• G is the set of instances of $r(x)$ that can be used as goals; the *answers* (in $\mathcal{F} \cup D_r$) of a goal $r(\tau_j)$ are the substitutions σ such that $\mathcal{F} \cup D_r \vdash r(\tau_j)\sigma$.

A deductive synthesis process then generates a (possibly infinite) sequence of programs $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_k \subseteq \cdots$ where, for every P_j, there is a corresponding set D_j of definition axioms, introducing the new predicate symbols of P_j. The program P_n that results from the process must be *totally correct* wrt the specification, i.e. P_n must compute, for every goal $r(\tau_j)$, the corresponding ground answers in $\mathcal{F} \cup D_r$.

Thus synthesis can be seen as the extraction from a large theory (a specification framework), wherein many different problems can be specified, of a small sub-theory (a program) that suffices for the purpose of computing all instances of a (single) specified problem. This means that we can reason about synthesis precisely and soundly by comparing the two theories, e.g. by comparing their models.

Note, however, that at the level of semantics, instead of *proof-theoretic answers* (as defined above), we should use *semantic answers* defined as follows. Let I be the isoinitial model of $\mathcal{F}$, and I^r the expansion of I by the new relation r, interpreted according to D_r. A *semantic answer* of $r(\tau)$ in $\mathcal{F} \cup D_r$ is a substitution σ such that $I^r \models r(\tau)\sigma$. The distinction is necessary, because in general, proof-theoretic answers are a subset of semantic answers. They coincide for ground substitutions if I^r is an isoinitial model of $\mathcal{F} \cup D_r$ (since in this case, for every ground σ, $I^r \models r(\tau)\sigma$ iff $\mathcal{F} \cup D_r \models r(\tau)\sigma$).

Now we can compare the isoinitial model of the specification framework $\mathcal{F}$ and the Herbrand model of the program P_j we are synthesising in $\mathcal{F}$, in the following manner. First, in $\mathcal{F}$, we use the set of constructions (i.e. the ground terms generated by the construction symbols) as its Herbrand universe $\mathcal{H}$, and we interpret '=' as the identity, and the function symbols as maps from ground terms to ground terms, in the usual way. Since $\mathcal{F}$ satisfies *reachability* and *freeness*, this interpretation isomorphically coincides with that of any isoinitial model of $\mathcal{F}$.

Then, for the synthesised program P_j, we interpret its predicate symbols in the domain $\mathcal{H}$. The meaning of a program is its minimum Herbrand model. Since every predicate symbol p of P_j has been introduced by a definition axiom, we can compare the meaning of p in P_j and that stated by the definition axiom D_p; if they coincide, then P_j is *semantically correct*.[6]

Example 2.2 Consider again Peano Arithmetic $\mathcal{PA}$. The construction symbols are '0' and 's', and the set of constructions is the set of numerals.

We can use the symbols '0', 's', and '=', in a program P and preserve their semantics, since the Herbrand universe of P with these symbols is the set of numerals.

In contrast, the semantics of the non-construction symbols '+' and '$\cdot$' is not preserved by a program, since they do not satisfy (in $\mathcal{PA}$) the freeness axioms. For example $\mathcal{PA} \vdash s0 + s0 = ss0$. However, in the Herbrand universe of any program with these symbols, $s0 + s0$ and $ss0$ are different terms (trees).[7]

A program P_{sum} to compute '+' could be synthesised in $\mathcal{PA}$ from the definition axiom D_{sum}: $\forall x, y, z\,(sum(x, y, z) \leftrightarrow x + y = z)$. Since we are interested in computing $\mathbf{m} + \mathbf{n}$, for every pair $\mathbf{m}, \mathbf{n}$ of numerals, the admissible goals for P_{sum} are of

[6] We remark that semantic correctness is equivalent to total correctness wrt ground goals, though in real programs they may diverge. In this paper we will not be concerned with this issue.

[7] Note that in CLP systems over natural numbers it may be possible to use '+' and '$\cdot$' in constraints.

the form $sum(\mathbf{m}, \mathbf{n}, X)$. For example, $sum(s0, s0, X)$ is an admissible goal, and its answer in $\mathcal{PA}$ is $X = ss0$.

To be totally correct, P_{sum} must compute the answers in $\mathcal{PA}$ of the admissible goals. Now the diagram of sum, $diag_{\mathcal{PA}}(sum) = \{sum(0,0,0), sum(0,s0,s0), \ldots\}$, is completely characterised by $\mathcal{PA} \cup \{D_{sum}\}$. If the minimum Herbrand model of P_{sum} coincides with $diag_{\mathcal{PA}}(sum)$, then P_{sum} is semantically correct. $\diamond$

Even at the level of proofs of semantic correctness, i.e. proofs of equivalence between models, e.g. between the Herbrand model of P_{sum} and $diag_{\mathcal{PA}}(sum)$ in this example, the specification framework $\mathcal{F}$ has an important part to play. In general, to prove semantic properties of programs, we should codify them by suitable formulas and prove them in $\mathcal{F}$. For example, if $\mathcal{F}$ is closed, then the semantic correctness of a terminating program P is equivalent to the consistency of the completion $Comp(P)$ with $\mathcal{F} \cup D_P$,[8] where D_P are the definition axioms of the predicate symbols of P. Therefore, if we can prove $Comp(P)$ from the axioms of $\mathcal{F} \cup D_P$, then P is semantically correct.

This shows why frameworks are important for the formal synthesis of standard logic programs. For the synthesis of CLP programs, they are equally important, for the same reasons. Therefore we will insist on using axiomatisations of constraint domains rather than just the domains themselves. In Section 3.2, we will discuss such axiomatisations based on generalisations of reachability and atomic completeness.

3 Specification Frameworks for CLP

For the synthesis of CLP programs, we need to incorporate *constraint domains* in specification frameworks. In this section we will adapt our notion of specification frameworks for the synthesis of standard logic programs to that of CLP programs. To this end, we shall define *constraint theories*, i.e. theories that correspond to constraint domains. Then we define *constraint specification frameworks* as extensions of constraint theories.

3.1 CLP Terminology

First we recall some basic CLP terminology. We will follow Jaffar and Maher [9], but we will be very brief and somewhat informal.

In the scheme CLP($\mathcal{X}$), $\mathcal{X}$ is a 4-tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$, where

• Σ is a (first-order, possibly many-sorted) signature containing the pre-defined constant, function, and predicate symbols used in the constraints.
• $\mathcal{D}$ is a Σ-structure. It represents the computation structure.
• $\mathcal{L}$ is a class of formulas of the language L_Σ generated by Σ.[9] It represents the class of constraints that can be used, and is called the *constraint language*.
• $\mathcal{T}$ is a Σ-theory. It is an axiomatisation of (some) properties of $\mathcal{D}$. It is normally assumed that $\mathcal{D}$ is a model of $\mathcal{T}$ and for every constraint $c \in \mathcal{L}$, $\mathcal{D} \models \tilde{\exists}\, c$ iff $\mathcal{T} \models \tilde{\exists}\, c$, where $\tilde{\exists}$ denotes existential closure.

We will call $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ a *CLP-instance*. In a CLP-instance, $\mathcal{T}$ is *satisfaction complete* if for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \tilde{\exists}\, c$ or $\mathcal{T} \models \neg\tilde{\exists}\, c$.

$(\mathcal{D}, \mathcal{L})$ is called a *constraint domain*. It is usually assumed that every element of $\mathcal{D}$ can be represented by a (possibly infinite) set of constraints. Elements of $\mathcal{D}$ that can only be represented by an infinite set of constraints are called *limit elements*.

[8] This is briefly discussed in [12].

[9] L_Σ is the whole language, not to be confused with $\mathcal{L}_\Sigma$ in [9] which is a sub-language of L_Σ.

3.2 Constraint Theories

In this section, we will state the CLP counterpart to Theorem 2.1, namely an existence criterion for the isoinitial models of the theory $\mathcal{T}$ in a CLP-instance. We then define constraint theories to be such $\mathcal{T}$'s that do indeed have isoinitial models.

First we introduce the notion of *$\mathcal{L}$-reachability*, where $\mathcal{L}$ is a set of constraints.

Definition 3.1 A structure $\mathcal{D}$ is *$\mathcal{L}$-reachable* if for every element d of $\mathcal{D}$, there is at least one constraint $\delta(x) \in \mathcal{L}$, such that $\delta(x)$ is true in $\mathcal{D}$ only for the valuation $x = d$. We will say that $\delta(x)$ *represents* d.

A theory $\mathcal{T}$ is *$\mathcal{L}$-reachable* if there is a $\mathcal{L}$-reachable model $\mathcal{D}$ of $\mathcal{T}$ and every element d of $\mathcal{D}$ is represented by at least one $\delta(x)$ such that $\mathcal{T} \vdash \exists! x\, \delta(x)$.

We call $\delta(x)$ a *data constraint* in $\mathcal{T}$.[10] $\diamond$

If $\mathcal{T}$ is $\mathcal{L}$-reachable wrt a model $\mathcal{D}$, then the constraint domain $(\mathcal{D}, \mathcal{L})$ has no limit elements. In this paper, we will consider only such domains. (We are currently investigating possible ways of extending our work to domains with limit elements. We will comment on this briefly in Section 6.)

$\mathcal{L}$-reachability thus means reachability by the constraints rather than by the ground terms, as in specification frameworks for standard logic programs (Section 2.1), of the language.

We can now give a criterion for determining if a $\mathcal{L}$-reachable theory $\mathcal{T}$ has an isoinitial model. We recall from [9] that $\mathcal{L}$ is normally assumed to be the smallest subset of L_Σ that contains the atomic formulas and is closed under $\wedge$ and $\exists$. It is easy to see that in this case every formula $c \in \mathcal{L}$ is logically equivalent to an existential formula of the form $\exists \underline{y}\, \Gamma$, where Γ is a conjunction of atomic formulas.

For our purpose, it is sufficient to require only that $\mathcal{L}$ is *any* subset (not necessarily the smallest) of L_Σ such that it is closed under $\wedge$ and $\exists$, it contains the atomic formulas, and for every $c \in \mathcal{L}$ there is an existential formula $\exists \underline{x}\, \Delta$, where Δ is *any* quantifier free formula, such that $\mathcal{T} \vdash c \leftrightarrow \exists \underline{x}\, \Delta$. We say such a set of constraints is *regular* in $\mathcal{T}$.

To state the CLP counterpart to Theorem 2.1, we shall use $\mathcal{L}$-reachability as a generalisation of reachability. In addition, we shall make use of the notion of *atomic decidability* to generalise atomic completeness.

Definition 3.2 A theory $\mathcal{T}$ is *atomically decidable* if for every n-ary relation r, and every n-tuple $\delta_1, \ldots, \delta_n$ of data constraints,

$$\begin{array}{ll} \text{either} & \mathcal{T} \vdash \exists y_1, \ldots, y_n\, (\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge r(y_1, \ldots, y_n)) \\ \text{or} & \mathcal{T} \vdash \exists y_1, \ldots, y_n\, (\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge \neg r(y_1, \ldots, y_n)). \end{array}$$

Now we have the following theorem:

Theorem 3.1 Let $\mathcal{T}$ be an $\mathcal{L}$-reachable theory, where the set of constraints $\mathcal{L}$ is regular in $\mathcal{T}$. Then $\mathcal{T}$ has an isoinitial model iff $\mathcal{T}$ is atomically decidable.

The proof is based on the following construction, which associates the theory $\mathcal{T}$ with a structure $\mathcal{D}_\mathcal{T}$ (as in [1]). This structure is built in a similar manner to Herbrand models, but it is based on data constraints instead of ground terms.

In the construction of $\mathcal{D}_\mathcal{T}$, we do not make use of $\mathcal{L}$-reachability. We simply consider constraints $\delta(x) \in \mathcal{L}$ such that $\mathcal{T} \vdash \exists! x\, \delta(x)$. For now, we call these constraints $\mathcal{T}$-constraints.[11] If the set of $\mathcal{T}$-constraints is not empty, then we can

[10] A data constraint thus identifies a single element of $\mathcal{D}$, and we use $\delta(x)$ to distinguish it. In [9], x is said to be determined or grounded by δ.

[11] They become data constraints when we prove $\mathcal{D}_\mathcal{T}$ to be a model.

construct the structure $\mathcal{D}_\mathcal{T}$ as described below. In general, such a $\mathcal{D}_\mathcal{T}$ may well not be a model of $\mathcal{T}$, but Theorem 3.1 tells us when it is an isoinitial model of $\mathcal{T}$.

We construct $\mathcal{D}_\mathcal{T}$ as follows. Whereas in a Herbrand model, function symbols are interpreted as maps from ground terms to ground terms, in $\mathcal{D}_\mathcal{T}$ we interpret them as functions from (equivalence classes of) $\mathcal{T}$-constraints to (equivalence classes of) $\mathcal{T}$-constraints as follows:

• We say that two $\mathcal{T}$-constraints δ_1 and δ_2 are equivalent if $\mathcal{T} \vdash \exists x\,(\delta_1(x) \wedge \delta_2(x))$. We use $[\delta]$ to denote the equivalence class of constraints containing a $\mathcal{T}$-constraint δ, and $\mathcal{H}_\mathcal{T}$ to denote the set of such equivalence classes.

• Every n-ary function symbol f is interpreted by $f_\mathcal{T} : \mathcal{H}_\mathcal{T}^n \to \mathcal{H}_\mathcal{T}$ defined thus:

$$f_\mathcal{T}([\delta_1],\ldots,[\delta_n]) = [\exists y_1 \cdots \exists y_n\,(x = f(y_1,\ldots,y_n) \wedge \delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n))].$$

We remark that $\exists y_1 \cdots \exists y_n\,(\ldots)$ belongs to $\mathcal{L}$ (since $\mathcal{L}$ is closed under $\exists$ and $\wedge$, and it contains the atomic formulas. In particular, $x = f(y_1,\ldots,y_n)$ belongs to $\mathcal{L}$). It is easy to see that it is a $\mathcal{T}$-constraint, i.e. $\mathcal{T} \vdash \exists! x\, \exists y_1 \cdots \exists y_n\,(x = f(y_1,\ldots,y_n) \wedge \delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n))$.

• Every k-ary relation symbol r is interpreted by the relation $r_\mathcal{T}$ defined thus: $r_\mathcal{T}([\delta_1],\ldots,[\delta_k])$ is true iff $\mathcal{T} \vdash \exists y_1 \ldots \exists y_k\,(r(y_1,\ldots,y_k) \wedge \delta_1(y_1) \wedge \cdots \wedge \delta_k(y_k))$.

As we already mentioned, $\mathcal{D}_\mathcal{T}$ may not be a model of $\mathcal{T}$ in general. However, if $\mathcal{T}$ is $\mathcal{L}$-reachable and atomically decidable, then $\mathcal{D}_\mathcal{T}$ is indeed an isoinitial model.

Proof. Let $\mathcal{D}_\mathcal{T}$ be the canonical structure defined in the above manner, and let $\mathcal{D}$ be any model such that $\mathcal{T}$ is $\mathcal{L}$-reachable wrt $\mathcal{D}$ (at least one such $\mathcal{D}$ exists, by hypothesis).

First we prove that $\mathcal{D}_\mathcal{T}$ is isomorphic to $\mathcal{D}$. Since every element d of $\mathcal{D}$ is represented by at least one data constraint $\delta_d(x)$, we build the map ϕ : domain of $\mathcal{D}$ $\to$ domain of $\mathcal{D}_\mathcal{T}$, such that $\phi(d) = [\delta_d(x)]$. We remark that, if $d_1 \neq d_2$, then $[\delta_{d_1}(x)] \neq [\delta_{d_2}(x)]$, i.e. ϕ is a bijective map.

Now we prove that it is an isomorphism. Let $f_\mathcal{D}$ and $f_\mathcal{T}$ be the interpretations of a (n-ary) function symbol f, in $\mathcal{D}$ and $\mathcal{T}$ respectively. Consider $d = f_\mathcal{D}(d_1,\ldots,d_n)$, and let $\delta_d(x), \delta_{d_1}(x), \ldots, \delta_{d_n}(x)$ be data constraints corresponding to $d, d_1, \ldots, d_n$; then $[\delta_d(x)]$ contains

$$\exists y_1 \cdots \exists y_n\,(x = f(y_1,\ldots,y_n) \wedge \delta_{d_1}(y_1) \wedge \cdots \wedge \delta_{d_n}(y_n)).$$

Indeed such a formula is a data constraint satisfied only by $x = d$. Therefore $[\delta_d(x)] = f_\mathcal{T}([\delta_{d_1}] \cdots [\delta_{d_n}])$, as required.

Now we consider the interpretations $r_\mathcal{T}, r_\mathcal{D}$ of a (k-ary) relation symbol r and prove that $d_1,\ldots,d_n \in r_\mathcal{D}$ iff $\phi(d_1),\ldots,\phi(d_n) \in r_\mathcal{T}$.

For the if-part, assume that $d_1,\ldots,d_n \in r_\mathcal{D}$, then

$$\exists y_1, \ldots, \exists y_n\,(\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge r(y_1,\ldots,y_n))$$

is satisfiable. Since $\mathcal{T} \vdash \exists y_1 \ldots \exists y_n\,(\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge r(y_1,\ldots,y_n))$ or $\mathcal{T} \vdash \exists y_1 \ldots \exists y_n\,(\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge \neg r(y_1,\ldots,y_n))$, we have that

$$\mathcal{T} \vdash \exists y_1 \ldots \exists y_n\,(\delta_1(y_1) \wedge \cdots \wedge \delta_n(y_n) \wedge r(y_1,\ldots,y_n)) \qquad (*)$$

Then by definition, $\langle [\delta_{d_1}] \cdots [\delta_{d_n}] \rangle \in r_\mathcal{T}$.

For the only-if-part, assume $\langle [\delta_1] \cdots [\delta_n] \rangle \in r_\mathcal{T}$, then (*) holds. Therefore, since $\mathcal{D}$ is a model of $\mathcal{T}$, there are $d_1^*, \ldots, d_n^*$ that satisfy (*); since every data constraint represents a unique element of $\mathcal{D}$, $d_i^* = d_i$, and we have our assertion.

Now we can prove that an $\mathcal{L}$-reachable $\mathcal{T}$ has an isoinitial model if and only if it is atomically decidable. We shall make use of a theorem in [1] (Theorem 5.4, p.158) which essentially says that a theory $\mathcal{T}$ has an isoinitial model iff $\mathcal{D}_\mathcal{T}$ is a model of $\mathcal{T}$, and $\mathcal{T}$ is atomically decidable. To apply this theorem, each equivalence class $[\delta(x)]$ must be representable by an existential formula (see end of Section 3.2).

Assume that $\mathcal{T}$ has an isoinitial model. Then, by the theorem in [1], $\mathcal{D}_{\mathcal{T}}$ is an isoinitial model and $\mathcal{T}$ is atomically decidable.

Now, assume that the hypotheses of our theorem are satisfied. Then $\mathcal{D}_{\mathcal{T}}$ can be constructed, since the set of $\delta(x)$ such that $\mathcal{T} \vdash \exists! x\, \delta(x)$ is not empty. Since $\mathcal{D}$ and $\mathcal{D}_{\mathcal{T}}$ are isomorphic and $\mathcal{D}$ is a model of $\mathcal{T}$, $\mathcal{D}_{\mathcal{T}}$ is a model of $\mathcal{T}$ and, again by the theorem in [1], it is an isoinitial model. □

An important corollary of Theorem 3.1 is that for every constraint domain $(\mathcal{D}, \mathcal{L})$ such that $\mathcal{T}$ is $\mathcal{L}$-reachable wrt to $\mathcal{D}$, $\mathcal{D}$ and $\mathcal{D}_{\mathcal{T}}$ are isomorphic (isoinitial) models. Therefore, $\mathcal{D}$ can be seen as an implementation of the ADT axiomatised by $\mathcal{T}$, and $\mathcal{D}_{\mathcal{T}}$ a kind of canonical representation.[12] In other words, we can use $\mathcal{T}$ as an abstract axiomatisation of $\mathcal{D}$, and use $\mathcal{D}_{\mathcal{T}}$ as a basis to build Herbrand $\mathcal{D}_{\mathcal{T}}$-models, which is independent from the implementation ($\mathcal{D}$ may be one of many possible isomorphic implementations).

We are particularly interested in $\mathcal{T}$ as an abstract axiomatisation of $\mathcal{D}$, since it represents our knowledge of the constraint domain, and is used in the synthesis process, or more generally to prove properties of programs such as semantic correctness. In this context, the abstraction from the implementation becomes an important feature.

Example 3.1 Let K be a set of constant and function symbols, and Σ_K the signature containing just '=' and K. Now consider the theory $\mathcal{T}(K)$ containing the identity axioms and the freeness axioms for K.

An isoinitial model of $\mathcal{T}(K)$ is the Herbrand structure $\mathcal{H}_K$ defined as follows. The domain of $\mathcal{H}_K$ is the Herbrand base generated by K, '=' is identity, and the symbols of K are interpreted in the usual way in Herbrand models.

$\mathcal{T}(K)$ is $\mathcal{L}_K$-reachable, where $\mathcal{L}_K$ is the smallest subset of formulas that is closed under $\wedge$ and $\exists$, and that contains all the equalities $v = \tau$, where v is a variable and τ a term. $\mathcal{L}_K$ is thus regular, and hence the canonical structure $\mathcal{D}_{\mathcal{T}(K)}$ is an isoinitial model of $\mathcal{T}(K)$ that is isomorphic to $\mathcal{H}_K$. This means that the model $\mathcal{H}_K$, used as a basis in standard Herbrand models, is isomorphic to the canonical data-constraint structure $\mathcal{D}_{\mathcal{T}(K)}$. Moreover, every data constraint $\delta(x)$ has a solved form $x = \tau$ (i.e. $\mathcal{T}(K) \vdash \exists x\, (x = \tau \wedge \delta(x))$, and hence $x = \tau \in [\delta(x)]$). ◇

Now we define constraint theories as follows:

Definition 3.3 A *constraint theory* for a constraint domain $(\mathcal{D}, \mathcal{L})$ is any theory $\mathcal{T}$ that is $\mathcal{L}$-reachable wrt $\mathcal{D}$, satisfaction complete wrt $\mathcal{L}$, and atomically decidable.

Example 3.2 Consider Presbürger Arithmetic $\mathcal{PR}$, which is the restriction of $\mathcal{PA}$ to the signature containing only '0', 's', '+', '=' (i.e. '$\cdot$' is omitted). It has an isoinitial model $\mathcal{D}_{\mathcal{PR}}$, which is the reduct of the isoinitial model $\mathcal{N}$ of $\mathcal{PA}$ to the restricted language of $\mathcal{PR}$. Since it is a decidable theory, $\mathcal{PR}$ is satisfaction complete wrt any constraint language, in particular $\mathcal{L}_{\mathcal{PR}}$, the language that is closed under $\exists$ and $\wedge$, and that contains all the atomic formulas. Thus $\mathcal{PR}$ is a constraint theory for the constraint domain $(\mathcal{D}_{\mathcal{PR}}, \mathcal{L}_{\mathcal{PR}})$. ◇

Thus constraint theories axiomatise constraint domains without limit elements, e.g. rational numbers and rational trees. In general, for any constraint domain $(\mathcal{D}, \mathcal{L})$, if $\mathcal{D}$ is $\mathcal{L}$-reachable and $\mathcal{L}$ contains *all* the atomic formulas and is regular, then a constraint satisfaction algorithm (if it exists) for the domain implicitly provides (computes) a constraint theory (for which $\mathcal{D}$ is an isoinitial model).

Theorem 3.1 requires that $\mathcal{L}$ is regular in $\mathcal{T}$. In the common constraint domains $\mathcal{L}$ is regular in the empty theory (hence in any $\mathcal{T}$), thus this requirement is not too

[12] $\mathcal{D}_{\mathcal{T}}$ is the CLP counterpart to the Herbrand base for a standard logic program.

restrictive. Nevertheless, it can be weakened: it is sufficient that every equivalence class $[\delta(x)]$ is representable by an existential formula $\exists \underline{y} \Delta(x, \underline{y})$ (not necessarily belonging to $\mathcal{L}$), that is, $[\delta(x)]$ contains a $\delta^*(x)$ such that $\mathcal{T} \vdash \forall \underline{x}(\delta^*(x) \leftrightarrow \exists \underline{y} \Delta(x, \underline{y}))$.

If the constraint domain we want to axiomatise has a constraint language $\mathcal{L}$ that does not satisfy the weaker requirement, we can only prove that $\mathcal{D}_{\mathcal{T}}$ is a *prime model* of $\mathcal{T}$. That is, for every model M of $\mathcal{T}$ there is an isomorphic embedding $\mu : \mathcal{D}_{\mathcal{T}} \to M$, but the unicity of μ is no longer guaranteed. This causes the loss of abstractness (see [1]).

However, we can recover isoinitiality by expanding the language of $\mathcal{T}$ as follows. For each equivalence class $[\delta(x)]$ that is not representable by an existential formula, we introduce a new atomic formula $r(x)$, by a definition axiom $\forall x(r(x) \leftrightarrow \delta(x))$. We get a conservative extension $\mathcal{T}^*$ of $\mathcal{T}$, with isoinitial model $\mathcal{T}_{\mathcal{D}^*}$, and we take $(\mathcal{T}_{\mathcal{D}^*}, \mathcal{L})$ as the axiomatised constraint domain. Since $\mathcal{L}$ does not contain the new formulas, they may not be computed in an implementation; however, every implementation must be isomorphic to the reduct of $\mathcal{D}_{\mathcal{T}^*}$ to the language of $\mathcal{T}$.

3.3 Constraint Specification Frameworks

Now we shall define constraint specification frameworks as extensions of constraint theories. Consider for example Peano Arithmetic $\mathcal{PA}$. In contrast to $\mathcal{PR}$, $\mathcal{PA}$ is not a constraint theory (if we use a constraint language closed under $\wedge$ and $\exists$ and containing all the atomic formulas). This is because $\mathcal{PA}$ is not satisfaction complete wrt formulas such as $\exists \underline{y}\, A$, with A atomic. However, $\mathcal{PA}$ is a rich and expressive theory (at least in principle we can specify any computable problem on natural numbers by a formula of $\mathcal{PA}$), and is thus suitable as a specification framework.

Definition 3.4 A *constraint specification framework* $\mathcal{F}_{\mathcal{T}}$ for a constraint theory $\mathcal{T}$ is a theory $\mathcal{T} \cup \mathcal{F}$, where $\mathcal{F}$ is a set of axioms of a language possibly larger than that of $\mathcal{T}$.

A constraint specification framework $\mathcal{F}_{\mathcal{T}}$ may be *closed* (with one intended isoinitial model), or *open* (with a class of intended models). In order for $\mathcal{T} \cup \mathcal{F}$ to have a clean semantics that agrees with that of the constraint theory $\mathcal{T}$, it must satisfy suitable *adequacy conditions*. We shall concentrate on closed frameworks, and only briefly consider open frameworks in Section 5.

Definition 3.5 Let $\mathcal{T}$ be a constraint theory (wrt a constraint language $\mathcal{L}$), and let $\mathcal{T} \cup \mathcal{F}$ be a *closed* constraint specification framework for $\mathcal{T}$. Let $\mathcal{D}^*$ be an isoinitial model of $\mathcal{T} \cup \mathcal{F}$.

We say that $\mathcal{T} \cup \mathcal{F}$ is *adequate* wrt $\mathcal{T}$ iff the reduct of $\mathcal{D}^*$ to the language of $\mathcal{T}$ is an isoinitial model of $\mathcal{T}$.

That is, for a closed framework $\mathcal{F}_{\mathcal{T}}$, we require that its isoinitial model (unique up to isomorphism) and that of $\mathcal{T}$ give the same interpretation to the symbols in the signature of $\mathcal{T}$, while there is no restriction on the interpretation of other, new symbols.

Example 3.3 As we hinted above, $\mathcal{PA}$ is a rich theory within which we can define many problems. In particular $\mathcal{PA}$ contains the constraint theory $\mathcal{PR}$. Moreover, the isoinitial model of $\mathcal{PA}$ is $\mathcal{N}$, and its reduct to the language of $\mathcal{PR}$ is just (isomorphic to) the isoinitial model $\mathcal{D}_{\mathcal{PR}}$ of $\mathcal{PR}$.

Thus $\mathcal{PA}$ is a closed constraint specification framework that is adequate wrt $\mathcal{PR}$. So we can use $\mathcal{PA}$ for specifying and synthesising CLP programs that compute over the constraint domain $(\mathcal{D}_{\mathcal{PR}}, \mathcal{L}_{\mathcal{PR}})$. $\diamond$

Note that this distinction between constraint specification frameworks and constraint theories is clear-cut if we use $\mathcal{L}$ that contains *all* the atomic formulas. It gets blurred if we allow $\mathcal{L}$ to be a more restricted language, as in some CLP systems.

Example 3.4 We could use $\mathcal{PA}$ as a constraint theory over the *restricted* constraint language $\mathcal{L}_{\mathcal{PA}}$ built *only* from the atomic formulas which represent linear equations. Indeed it is satisfaction complete wrt $\mathcal{L}_{\mathcal{PA}}$, and it has $\mathcal{N}$ as an isoinitial model. $\diamond$

However, we shall maintain the distinction between frameworks and theories in order to be able to reason clearly about synthesis in CLP.

4 Deductive Synthesis in CLP

Having defined constraint specification frameworks, we now describe the process of synthesising CLP programs from specifications defined in constraint specification frameworks. Our approach to synthesis is deductive, i.e. it iteratively generates clauses for the specified program until the entire program has been derived. The synthesis process generates a (possibly infinite) sequence of CLP programs $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_k \subseteq \cdots$ where, for every P_j, there is a corresponding set D_j of definition axioms, introducing the new predicate symbols of P_j. The program P_n that results from the process must be *totally correct* wrt the specification.

Clearly the synthesis process must employ some synthesis method to generate the P_j's. However, in this paper, we will not be concerned with specific synthesis methods. The design of such methods is a major topic in its own right. Rather we will concentrate on the criteria for correctness, and for halting the synthesis process successfully with a totally correct program. We will assume synthesis methods that meet (and employ) these criteria.

Let $\mathcal{T}$ be a constraint theory, and $\mathcal{F}_{\mathcal{T}}$ a closed specification framework that is adequate wrt $\mathcal{T}$, with constraint language $\mathcal{L}$ and isoinitial model $\mathcal{D}_{\mathcal{F}_{\mathcal{T}}}$. As in Section 2.2, we define a *specification* in $\mathcal{F}_{\mathcal{T}}$ to consist of a definition axiom D_r of the relation r to be computed, and a set $G = \{g_1, \ldots, g_k, \ldots\}$ of admissible goals. We require that D_r is *adequate*, i.e. $\mathcal{F}_{\mathcal{T}} \cup D_r$ is adequate wrt $\mathcal{T}$. This ensures that $\mathcal{D}^r_{\mathcal{F}_{\mathcal{T}}}$, the expansion of $\mathcal{D}_{\mathcal{F}_{\mathcal{T}}}$ by the new relation r (according to D_r), is the canonical isoinitial model of $\mathcal{F}_{\mathcal{T}} \cup D_r$.

Each goal g_k is of the form $c \wedge r(\tau)$, where c is a constraint, and an answer is a constraint. Again, as in Section 2.2, we distinguish between proof-theoretic and semantic answers. A constraint a is *proof-theoretic answer* of g_k if $\mathcal{F}_{\mathcal{T}} \cup D_r \vdash a \rightarrow g_k$. It is a *semantic answer* if $\mathcal{D}^r_{\mathcal{F}_{\mathcal{T}}} \models a \rightarrow g_k$. For data constraints, the two sets of answers coincide if D_r is adequate. Henceforth, unless otherwise indicated, we consider semantic answers only.

We say that a (possibly infinite) set A of answers for a goal g_k is *complete* in $\mathcal{F}_{\mathcal{T}}$ iff

$$\mathcal{D}^r_{\mathcal{F}_{\mathcal{T}}} \models \tilde{\forall}(\bigvee_{a_i \in A} a_i \leftarrow g_k)$$

where $\tilde{\forall}$ denotes universal closure.

Now we give an example of a specification in a constraint specification framework:

Example 4.1 Consider the constraint theory $\mathcal{PR}$ for the constraint domain $(\mathcal{D}_{\mathcal{PR}}, \mathcal{L}_{\mathcal{PR}})$, the constraint specification framework $\mathcal{PR} \cup \mathcal{PA}$, i.e. $\mathcal{PA}$. Then the following specification

$$\begin{array}{lcl} D_{prod} & : & \tilde{\forall}\,(prod(x,y,z) \leftrightarrow x \cdot y = z) \\ G_{prod} & : & \{c(m,n), prod(m,n,z) \mid c(m,n) \text{ has a finite set of solutions}\} \end{array}$$

specifies a CLP program P_{prod} to compute '$\cdot$' in $\mathcal{PA}$. D_{prod} is adequate. The isoinitial model of $\mathcal{PA} \cup D_{prod}$ is $\mathcal{N}^{prod}$, the expansion of $\mathcal{N}$ by the relation $prod$, interpreted as the graph of '$\cdot$'.

To compute $2 \cdot 2$, for example, we use the goal $m = ss0, n = ss0, prod(m, n, z)$.

As another example, a complete set of answers (in $\mathcal{PA}$) for the goal $m + n = ss0, prod(m, n, z)$ is $\{a_1, a_2, a_3\}$ where a_1, a_2, a_3 are respectively $(m = ss0 \wedge n = 0 \wedge z = 0)$, $(m = s0 \wedge n = s0 \wedge z = s0)$, and $(m = 0 \wedge n = ss0 \wedge z = 0)$. Indeed $\mathcal{N}^{prod} \models m + n = ss0 \wedge prod(m, n, z) \leftrightarrow a_1 \vee a_2 \vee a_3$. $\diamond$

In the synthesis process, we shall use the following notions of correctness of a program P wrt a specification in $\mathcal{F}_{\mathcal{T}}$:

- P is *partially correct* if for every goal g_k in the specification, every answer of g_k computed by P is an answer in $\mathcal{F}_{\mathcal{T}}$ of g_k.
- P is *totally correct* if the set of answers computed by P for a goal g_k is complete.

4.1 The Synthesis Process

Now we illustrate the synthesis process for CLP programs, starting from a specification in a (closed) constraint specification framework. We use the simple specification in Example 4.1 for the sake of brevity and clarity.

Example 4.2 A possible deductive synthesis process to synthesise P_{prod} would be the following:
(i) Synthesise a clause for the goal $prod(x, 0, z)$. In $\mathcal{PA} \cup D_{prod}$ we can prove:
$prod(x, 0, z) \leftrightarrow x \cdot 0 = z \leftrightarrow z = 0$.
and so we get the program P_1: $prod(x, 0, z) \leftarrow z = 0$.
(ii) Next synthesise a clause for $prod(x, s(y), z)$. In $\mathcal{PA}$ we can prove

$$\forall y \exists m (m + m = s(y) \vee m + s(m) = s(y)) \tag{2}$$

Since $m + m = s(y)$ and $m + s(m) = s(y)$ are in our constraint language and m is less than $s(y)$, we can use (2) to derive recursive (terminating) clauses as follows, where each step is provable in $\mathcal{PA} \cup D_{prod}$:

$$\begin{array}{rcl}
prod(x, s(y), z) & \leftrightarrow & \exists m (m + m = s(y) \vee m + s(m) = s(y)) \wedge prod(x, s(y), z) \\
& \leftrightarrow & \exists m (m + m = s(y) \vee m + s(m) = s(y) \wedge x \cdot s(y) = z) \\
& \leftrightarrow & \exists m \exists w (m + m = s(y) \wedge x \cdot m = w \wedge w + w = z \\
& & \vee\, m + s(m) = s(y) \wedge x \cdot m = w \wedge w + w + x = z)
\end{array}$$

In the first equivalence, constraints have been introduced by (2). The subsequent steps use some obvious logical equivalences, the identity axioms and well-known properties of $+$ and $\cdot$ provable in $\mathcal{PA}$. By replacing $x \cdot m = w$ with $prod(x, m, w)$ in the last equivalence, we get the program P_2:

$$\begin{array}{rcl}
prod(x, 0, z) & \leftarrow & z = 0 \\
prod(x, s(y), z) & \leftarrow & m + m = s(y), w + w = z, prod(x, m, w) \\
prod(x, s(y), z) & \leftarrow & m + s(m) = s(y), w + w + x = z, prod(x, m, w)
\end{array}$$

that is our final program P_{prod}. $\diamond$

It is worth noting that if instead we were synthesising a standard logic program for computing '$\cdot$' in $\mathcal{PA}$, then we would need to go on with the synthesis of clauses for '$+$' (as in Example 2.2). Moreover, to achieve reasonable efficiency, we would need a suitable program for computing the integer half m of $s(y)$; in contrast, using a constraint domain this would be left to the (hopefully efficient) constraint-solving algorithms. Thus, in a given specification framework, the synthesis process of CLP programs harnesses the computational resources of the constraint domain, and often it is even shorter than the synthesis of standard logic programs. This is of course to be expected, since a CLP language is more expressive.

Returning to the synthesis process, in each step we have to ensure that P_j is *partially correct*, and terminating (it suffices that for every goal g_k and data constraint $\delta(x)$, either $\delta(x) \wedge g_k$ is finitely failed, or $\delta(x) \wedge g_k$ has at least one successful derivation).[13] Moreover, we also need a criterion for determining if and when P_j has become *totally correct*, i.e. if and when the synthesis process can successfully halt. We call such a criterion a *halting criterion*. Now we state these criteria. They are based on the completion $Comp(P)$ of a program P [4, 9], and are similar to our criteria for synthesis in standard LP [12].

Consider synthesis in the constraint specification framework $\mathcal{F}_\mathcal{T}$ containing the constraint theory $\mathcal{T}$. Then:

• for *partial correctness*, we require that, for every P_j, $\mathcal{F}_\mathcal{T} \cup D_j \vdash Comp^+(P_j)$, where $Comp^+(P_j)$ is the *if*-part of $Comp(P_j)$. This follows easily from point 2 of Theorem 6.1 in [9].

For example, in Example 4.2, it is easy to see that after step 1, we have $\mathcal{PA} \cup D_{prod} \vdash Comp^+(P_1)$ and after step 2, we have $\mathcal{PA} \cup D_{prod} \vdash Comp(P_2)$.

• for *total correctness*, we use the following *halting criterion*:
For each predicate p of P_n, if $\mathcal{F}_\mathcal{T} \cup D_n \vdash Comp^-(P_n, p)$, then do not search for any more clauses with head p; if $\mathcal{F}_\mathcal{T} \cup D_n \vdash Comp^-(P_n, p)$ for every p, then the synthesis can successfully halt, where $Comp^-(P_n, p)$ is the *only-if*-part of the completed definition of p. This follows from points 2, 4 and 6 of Theorem 6.1 in [9], using the hypotheses of $\mathcal{L}$-reachability and termination.

Looking at Example 4.2, we see that $\mathcal{PA} \cup D_{prod} \vdash Comp^-(P_2, prod)$. Thus P_2 is totally correct, i.e. the synthesis process can successfully halt.

5 Synthesis in Open Frameworks

In our work in synthesis in standard LP, the single most important result, from the point of view of object-oriented synthesis, is the characterisation of *steadfast programs* [12]. Such a program can be synthesised in an open specification framework, and thus it contains parameters. However, it can be synthesised once and for all, and then reused correctly with any programs that correctly compute its parameters. In other words, a steadfast program can be (re)used correctly in a class of closed frameworks. Such reuse of programs in different frameworks is of course especially apt in CLP.

Steadfastness thus provides a basis for synthesising modular and reusable (and hence maintainable) programs. In this section, we briefly discuss the synthesis of steadfast CLP programs in open constraint specification framework, and thus show that our approach can be used to synthesise modular and reusable CLP programs.

An open constraint specification framework $\mathcal{F}_\mathcal{T}$ for a constraint theory $\mathcal{T}$ is any axiomatisation $\mathcal{T} \cup \mathcal{F}$, together with a *dependency* $\Delta \Leftarrow \Pi$, where Δ and Π are symbols of the signature of $\mathcal{F}_\mathcal{T}$. We call Δ *defined symbols* and Π *parameters*. Intuitively speaking, the meaning of the defined symbols is completely characterised by the axioms of the framework in terms of the parameters. The dependency contains all the symbols of the signature, except those of $\mathcal{T}$, since they are completely characterised by $\mathcal{T}$ and do not depend on parameters.

We will write $\mathcal{F}_\mathcal{T} : \Delta \Leftarrow \Pi$ to denote an open framework $\mathcal{F}_\mathcal{T}$ with dependency $\Delta \Leftarrow \Pi$. Parameters of Π can be *instantiated* by a closed framework $\mathcal{I}_\mathcal{T}$ with a

[13] In our synthesis process, currently we need to rely on independent tests for termination.

signature containing Π but not Δ. If $\mathcal{I}_\mathcal{T}$ is any such closed framework, and $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T}$ is consistent, we call it an *instance* of $\mathcal{F}_\mathcal{T}$ (corresponding to $\mathcal{I}_\mathcal{T}$).

As in the case of closed frameworks (Section 3.3), we require open frameworks to be adequate, where $\mathcal{F}_\mathcal{T} : \Delta \Leftarrow \Pi$ is *adequate* iff every instance $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T}$ is a closed framework that is adequate wrt $\mathcal{T}$. An example of an adequate open constraint specification framework is the following:

Example 5.1 We recall the constraint theory in Example 3.4, that is $\mathcal{PA}$ with constraint language $\mathcal{L}_{\mathcal{PA}}$. Consider the constraint specification framework $\mathcal{LIST}_{\mathcal{PA}}$ (for lists) defined as follows:

$$
\begin{array}{lll}
\text{SORTS:} & \multicolumn{2}{l}{Nat, Elem, List} \\
\text{FUNCTIONS:} & nil & : \ \to List; \\
 & | & : \ (Elem, List) \to List; \\
\text{RELATIONS:} & elemr & : \ (Elem, Elem); \\
 & elemi & : \ (List, Nat, Elem) \\
\text{AXIOMS:} & \multicolumn{2}{l}{\forall a : Elem, B : List(\neg nil = a|B);} \\
 & \multicolumn{2}{l}{\forall a_1, a_2 : Elem, B_1, B_2 : List(a_1|B_1 = a_2|B_2 \to a_1 = a_2 \wedge B_1 = B_2);} \\
 & \multicolumn{2}{l}{\forall e : Elem, L : List(elemi(L, 0, e) \leftrightarrow \exists B : List(L = e|B));} \\
 & \multicolumn{2}{l}{\forall e : Elem, L : List, i : Nat} \\
 & \multicolumn{2}{l}{(elemi(L, s(i), e) \leftrightarrow \exists a : Elem, B : List(L = a|B \wedge elemi(B, i, e)));} \\
 & \multicolumn{2}{l}{(H(nil) \wedge \forall x : Elem, J : List(H(J) \to H(x|J))) \to \forall L : List\, H(L)}
\end{array}
$$

Apart from the signature and the axioms of $\mathcal{PA}$, $\mathcal{LIST}_{\mathcal{PA}}$ contains the construction symbols 'nil' and '$|$' for lists; the relation $elemi(L, i, e)$ (meaning the element e occurs at position i in L); the freeness axioms for the construction symbols 'nil' and '$|$'; two axioms for the relation $elemi(L, i, e)$; and the first-order induction schema on lists, where H is any (first-order) formula of the language of lists.

$\mathcal{LIST}_{\mathcal{PA}}$ is *open* since there are no construction symbols or axioms characterising the domain of $Elem$ and the relation $elemr$ on its elements. Its dependency is[14]

$$=_{List}, nil, |, elemi \Leftarrow Elem, elemr, =_{Elem}$$

$\mathcal{LIST}_{\mathcal{PA}}$ is *adequate*, since whenever we give a complete framework $\mathcal{I}_{\mathcal{PA}}$ axiomatising the parameters $Elem$, $elemr$, $=_{Elem}$, we get a complete framework $\mathcal{LIST}_{\mathcal{PA}} \cup \mathcal{I}_{\mathcal{PA}}$. For instance $elemr$ might be instantiated by some ordering relation. $\diamond$

As discussed in [12, 11], in an adequate open framework $\mathcal{F}_\mathcal{T} : \Delta \Leftarrow \Pi$, we can specify and synthesise *open programs* of the form $P : p_1, \ldots, p_n \Leftarrow q_1, \ldots, q_m$, where $p_1, \ldots, p_n, q_1, \ldots, q_m$ are the predicate symbols of P. We call $p_1, \ldots, p_n \Leftarrow q_1, \ldots, q_m$ the *dependency* of P: $p_1, \ldots, p_n$ are new symbols introduced in $\mathcal{F}_\mathcal{T}$ by a set D_P of definition axioms, and $q_1, \ldots, q_m$ are parameters of P and must belong to the parameters Π of $\mathcal{F}_\mathcal{T}$.

P is *steadfast* in $\mathcal{F}_\mathcal{T}$ if it has the following property. Let $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T}$ be any instance of $\mathcal{F}_\mathcal{T}$, and let $diag(q_1, \ldots, q_m)$ be the diagram of $q_1, \ldots, q_m$ in $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T}$. Then $P \cup diag(q_1, \ldots, q_m)$ is semantically correct in $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T} \cup D_P$, i.e. the minimum $\mathcal{D}_\mathcal{T}$-model of P (as defined in [9]) interprets $p_1, \ldots, p_n$ in the same way as the isoinitial model of $\mathcal{F}_\mathcal{T} \cup \mathcal{I}_\mathcal{T}$ expanded according to the definition axioms D_P.

This means that we can instantiate P in a correct way, using a correct program Q for computing $q_1, \ldots, q_m$ in $\mathcal{I}_\mathcal{T}$. Moreover, if $\mathcal{I}_\mathcal{T}$ is a constraint theory expanding $\mathcal{T}$, and $q_1, \ldots, q_m$ are predicates that can be used in its constraints, then we do not need any program Q, and instead we can safely use the constraint solving system in $\mathcal{I}_\mathcal{T}$.

[14]Note that $\mathcal{LIST}_{\mathcal{PA}}$ has polymorphic identity '$=$'.

Example 5.2 Consider the following definition axiom D_{chain}:

$\forall x : List(chain(x) \leftrightarrow$
$\forall i : Nat, a, b : Elem(elemi(x, i, a) \wedge elemi(x, s(i), b) \rightarrow elemr(a, b))$

which specifies the property of being a chain wrt the relation $elemr$.

The following program P_{chain} is steadfast in $\mathcal{LIST}_{\mathcal{PA}}$:

$$\begin{array}{rcl} chain(nil) & \leftarrow & \\ chain(u|nil) & \leftarrow & \\ chain(u|v|l) & \leftarrow & elemr(u, v), chain(v|l) \end{array}$$

P_{chain} can be used correctly with any program that correctly computes $elemr$. For example, if $\mathcal{LIST}_{\mathcal{PA}} \cup \mathcal{I}_{\mathcal{PA}}$ is an instance in which the parameters $Elem$, $elemr$, $=_{Elem}$ are instantiated to Nat, $<$, $=_{Nat}$, and $Q_<$ a program that computes $<$ and is correct in $\mathcal{I}_{\mathcal{PA}}$. Then $P \cup Q_<$ is a correct program in $\mathcal{LIST}_{\mathcal{PA}} \cup \mathcal{I}_{\mathcal{PA}}$ for checking that lists of natural numbers are ordered (by $<$). Of course, if $\mathcal{I}_{\mathcal{PA}}$ is a constraint theory (for some constraint domain), then we do not need $Q_<$. $\diamond$

Our synthesis approach can be used to synthesise steadfast CLP programs such as P_{chain}. The halting problem here is to determine if and when a steadfast program has been obtained. For this, we can use a halting criterion very similar to that for total correctness in closed frameworks (Section 4.1), viz. if $Comp^-(P_j, p)$ is provable for every symbol p of P_j that is not a parameter in its dependency, then (P_j is steadfast and) the synthesis can halt successfully.
In Example 5.2, it is easy to verify that $\mathcal{LIST}_{\mathcal{PA}} \cup D_{chain} \vdash Comp^-(P_{chain}, chain)$. We have chosen this simple example for brevity. Nevertheless, it makes the important point that we can prove steadfastness in CLP. Moreover, steadfastness is the natural characterisation of correctness of CLP modules. In this respect, it is related to the compositional semantics of [2, 7].

Thus we have shown that our approach can be used to synthesise reusable CLP modules. More interestingly, parameters of an open constraint specification framework can themselves be instantiated by constraint domains. Our approach therefore provides a basis for object-oriented CLP software engineering.

6 Conclusion

To date, the only work (as far as we are aware) on constructing CLP programs from specifications is [6]. However, their approach is different from ours in that it is informal, whereas ours is completely formal, the advantage of a formal approach being that it can formally verify correctness.

In our approach, we start from constraint specification frameworks. These are specification frameworks that contain constraint theories. Constraint theories, in turn, are axiomatisations of constraint domains. As such they can be seen as a contribution to CLP semantics.

However, we have not treated constraint domains $(\mathcal{D}, \mathcal{L})$ with limit elements. We are currently investigating how best to formulate constraint theories for such domains. For instance, it may be possible to consider an axiomatisation with a (possibly non-denumerable) set of 'limit' constants for limit elements as a constraint theory, but $\mathcal{L}$ will not contain the 'limit' constants. Another possibility is to consider an $\mathcal{L}$-reachable constraint theory $\mathcal{T}$ with isoinitial model $\mathcal{D}_{\mathcal{T}}$ that is a sub-structure of $\mathcal{D}$ and is elementarily equivalent to $\mathcal{D}$. Whichever way we choose to axiomatise $(\mathcal{D}, \mathcal{L})$, we need suitable conditions for the validity of our halting criterion.

We are concerned with a good semantics for synthesis, not just for computation (e.g. [7]), which enables us to reason precisely and soundly about synthesis. To this end, our constraint specification frameworks are endowed with isoinitial semantics. This provides an abstract semantics for defining the meaning of specifications and program correctness. This semantics is thus a major contribution of this paper.

Our approach to synthesis is deductive and hence iterative, and we have formulated conditions that any synthesis method must meet in order to preserve correctness at each step. Furthermore, we have defined criteria for halting the iterative synthesis process successfully, i.e. when a totally correct program has been synthesised.

More importantly, from the point of view of practical software engineering, our approach can be used to synthesise steadfast, or modular and reusable, CLP programs. The reuse of steadfast CLP programs in different constraint domains is truly in the spirit of the original concept of the CLP(χ) scheme. Furthermore, it may well be possible to extend our approach to open constraint theories, i.e. open constraint domains. This will be quite a departure from the current practice of having only closed constraint domains. Anyone for CLP($\chi(\psi)$)?

References

[1] A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.

[2] M. Bugliesi, E. Lamma and P. Mello. Modularity in logic programming. *J. Logic Programming* 19,20:443–502, 1994. Special issue: Ten years of logic programming.

[3] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.

[4] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293-322. Plenum Press, 1978.

[5] Y. Deville and K.K. Lau. Logic program synthesis. *J. Logic Programming* 19,20:321–350, 1994. Special issue: Ten years of logic programming.

[6] Y. Deville and P. Van Hentenryck. Construction of CLP Programs. In D.R. Brough, editor, *Logic Programming: New Frontiers*, pages 112–135, Kluwer, 1992.

[7] M. Gabbrielli, G.M. Dore and G. Levi. Observable semantics for constraint logic programs. To appear in *J. Logic and Computation*.

[8] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987.

[9] J. Jaffar and M. Maher. Constraint logic programming: A survey. *J. Logic Programming* 19,20:503-581, 1994. Special issue: Ten years of logic programming.

[10] K.K. Lau and M. Ornaghi. An incompleteness result for deductive synthesis of logic programs. In D.S. Warren, editor, *Proc.* 10th *Int. Conf. on Logic Programming*, pages 456–477, MIT Press, 1993.

[11] K.K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *Lecture Notes in Computer Science* 883, pages 104-121, Springer-Verlag, 1994.

[12] K.K. Lau, M. Ornaghi and S.-Å. Tärnlund. The halting problem for deductive synthesis of logic programs. In P. van Hentenryck, editor, *Proc.* 11th *Int. Conf. on Logic Programming*, pages 665–683, MIT Press, 1994.

[13] J.W. Lloyd. Practical advantages of declarative programming. Invited talk at 1994 Joint Conference on Declarative Programming.

[14] J.C. Shepherdson. Negation in logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19-88. Morgan Kaufmann, 1988.

[15] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.