International Journal on Artificial Intelligence Tools

Mode Analysis During Program Development

--Manuscript Draft--

Manuscript Number:	IJAIT-D-13-00094R4
Full Title:	Mode Analysis During Program Development
Article Type:	Research Paper
Keywords:	Mode analysis; static analysis; abstract interpretation; minimal function graphs; logic program construction; program design schema; stepwise refinements.
Corresponding Author:	Emmanouil Marakakis, Ph.D. Technological Educational Institute of Crete Heraklion, Crete GREECE
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	Technological Educational Institute of Crete
Corresponding Author's Secondary Institution:	
First Author:	Emmanouil Marakakis, Ph.D.
First Author Secondary Information:	
Order of Authors:	Emmanouil Marakakis, Ph.D.
	Haridimos Kondylakis, Ph.D.
Order of Authors Secondary Information:	
Abstract:	Mode analysis in logic programs has been used mainly for code optimization. The mode analysis in this paper supports the program construction process. It is applied to partially complete logic programs. The program construction process is based on schema refinements and refinements by data type operations. Refinements by data type operations are at the end of the refinement process. This mode analysis supports the proper application of refinements by data type operations. In addition, it checks that the declared modes as defined by the Data Type (DT) operations are consistent with the inferred runtime modes. We have implemented an algorithm for mode analysis based on minimal function graphs. An overview of our logic program development method and the denotational semantics of the analysis framework are presented in this paper.
Response to Reviewers:	

International Journal on Artificial Intelligence Tools
Vol. XX, No. X (2015) 1–40

World Scientific
www.worldscientific.com

MODE ANALYSIS DURING PROGRAM DEVELOPMENT

EMMANOUIL MARAKAKIS

Department of Informatics Engineering, Technological Educational Institute of Crete
Heraklion, GR-71410, Greece
mmarak@cs.teicrete.gr

HARIDIMOS KONDYLAKIS

Institute of Computer Science, Laboratory, Address,
N. Plastira 100, V. Vouton, GR-71110, Heraklion, Greece
kondylak@ics.forth,gr

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Mode analysis in logic programs has been used mainly for code optimization. The mode analysis in this paper supports the program construction process. It is applied to partially complete logic programs. The program construction process is based on schema refinements and refinements by data type operations. Refinements by data type operations are at the end of the refinement process. This mode analysis supports the proper application of refinements by data type operations. In addition, it checks that the declared modes as defined by the Data Type (DT) operations are consistent with the inferred runtime modes. We have implemented an algorithm for mode analysis based on minimal function graphs. An overview of our logic program development method and the denotational semantics of the analysis framework are presented in this paper.

Keywords: Mode analysis; static analysis; abstract interpretation; minimal function graphs; logic program construction; program design schema; stepwise refinements.

1. Introduction

In logic programming, information about the instantiation modes of variables is used for code optimization and helps compilers to generate more efficient code by using different specialized unification algorithms for terms in different instantiation modes. Mode analysis is a formal framework for static analysis of programs in order to derive run-time properties without running them.

The mode analysis of this paper supports the program construction process by ensuring that the declared modes of data type (DT) operations are consistent with the

derived ones by the mode analysis procedure. So, given a partially constructed logic program, and a moded initial goal, a mode inference procedure computes call and success modes as far as possible. These inferred modes are used to check the call modes of DT operations for compatibility with the declared modes.

More specifically, in this paper, a mode analysis tool which supports a logic program construction methodology is presented. The logic program construction method is schema-based and it is supported by a set of tools in order to automate the construction process as much as possible and to confine the human involvement at the level of program design decisions and even higher at the level of sequences of design decisions. In order to satisfy these goals the method is supported by a set of tools. These tools are the following.

- A *partial evaluator* is used to transform a constructed program into an equivalent one with less clauses and possibly more efficient.
- An *interactive verifier* which supports the proof of correctness of logic programs constructed by this methodology [1].
- An *interactive KB update tool* which allows the programmer through a user friendly interface to update the knowledge base (KB), i.e. logic programs, theorems, axioms, lemmas, data type (DT) operations and First Order Logic (FOL) transformation rules.
- An *interactive program development tool* which allows the programmer through a user friendly interface, i.e. Windows-based and menu driven interface, to perform design decisions for constructing a logic program [2]. This tool assists the programmer to take the appropriate design decisions (or refinements) and to perform them. In addition, after each design decision the user can see graphically the refinement tree with annotations which assist him in the refinement process.
- A *static analysis tool* which is presented in this paper.

This development method constructs initially an understandable software system used for verification. The constructed program is partially evaluated to get an efficient and more complex program for running. Program reasoning in this method is performed at the level of design decisions. A programmer of this method is not expected to write programs in the target language. The focus of the programming task is on the design decisions that will be applied and on the order that they will be applied. A programmer is expected to manipulate refinement trees and to reason on the refinement trees which are high-level designs. Particular arrangements of refinements in refinement trees encode structural similarities between the programs that each refinement tree represents. The aim is as much as possible guidance to be given to programmers by the system. It is desirable design decisions to be performed by the system. The overall goal is software development to become as much as possible automatic.

Our novel mode analysis method is based on *minimal function graphs* (*mfg*).The *minimal function graphs* (*mfg*) method originally was developed in functional languages in [3] for program analysis. This method has been introduced in logic programming in [4] for analysis-based compiler optimizations and in [5] for program specialization. The *mfg* approach is designed to generate only the call patterns and their success patterns that are reachable from a given input query.

The main novelties of our work are the following.

a) The mode analysis is used during program development supporting the program construction process.
b) The mode analysis is applied to logic programs that have been constructed partially.

In more detail, our schema-based method constructs typed, moded logic programs by stepwise top-down design using five program schemata, data types (DTs) and modes [6], [7] [8]. Each schema consists of a set of clause schemata together with type and mode schemata for each predicate variable occurring in them. A set of built-in DTs is proposed which forms a kernel for defining and implementing user-defined DTs. User-defined DTs are implemented by using this schema-based method.

A program is constructed by this method by successively refining undefined predicates. The lowest refinement level involves refinement by DT operations. Modes in this method support the application of the refinement operations. The programs which are constructed by this method are polymorphic many-sorted programs. They satisfy the head condition and the transparency condition [9] which they ensure that no run-time type checking is needed. Finally, the programs satisfy declared input-output modes when run using the standard left-right depth-first computation rule. The logic programs which are constructed by this schema-based method are called *Schema-Instance Programs* or *SI-Programs.* Mode inference is meaningful only when the computation rule has been specified. The left to right computation rule of Prolog is assumed for SI-programs.

The specification of each DT operation defines its computation task. The implementation of each DT operation is assumed to satisfy its specification. The DT operations, in order to compute the task for which they have been designed require their arguments to be sufficiently instantiated when they are called. This instantiation requirement is expressed by assigning a mode to each DT operation. The modes that are used by this mode analysis method are *g* and *u* which stand for "ground" and "unknown" respectively.

For example, the DT operation *head*(*sq*, *el*) is defined to be true if *el* is the first element of the sequence *sq*. SI-programs are intended to satisfy the requirement that each DT operation should have inferred mode which is subsumed by the declared one. The (declared) mode of this DT operation is *Mode*(*head*) = (*g*, *u*) meaning that *sq* is supposed to be ground when *head*(*sq*, *el*) is called. Let us assume that *head*(*sq*, *el*) is part of a larger program which sorts the elements of sequence *sq* and the element el has to be compared with another element. Then, the comparison test will cause an instantiation error if *el* is not ground to ensure that this kind of error does not occur. The modes of all DT arguments are required to be ground after the completion of the DT operations.

During construction of an SI-program the arguments of a DT operation or equality predicate have to be matched with the arguments of the predicate in whose definition the DT operation or the equality appears. The argument association process is performed by the programmer. Static mode inference is performed dynamically after the matching of the arguments of each DT operation or equality predicate with the arguments of the

predicate it defines. If the inferred call mode of a DT operation is not subsumed by its declared one then mode incompatibility occurs. In case of mode incompatibility the system asks the designer to repeat the argument matching for the DT operation which caused the problem(s).

This paper is organized as follows. Related work is discussed in Section 2. An overview of our schema-based method is presented in Section 3. The abstract analysis framework is presented in Section 4. Section 5 presents the mode analysis. Section 6 presents an example which illustrates the features of this approach. Finally, Section 7 illustrates and discusses the implementation of this mode analysis system. Section 8 has discussion, conclusions and future research. The required background is presented in the appendix of the paper.

2. Related Work

A lot of work has been done in abstract interpretation the last 20 years. An overview of most of these works can be found on [10]. Systems in the area which use static analysis based on abstract interpretation include the ones which help to ensure software correctness on critical applications [11] and the ones which try to ensure correctly-moded predicate calls by performing run-time checking and delaying [9], [12]. A declarative language for logic programming with similar goals is GÖDEL [9]. Delay declarations in GÖDEL allow calls to proceed only if they are sufficiently instantiated thus avoiding unexpected failure. MU-PROLOG is an implementation of PROLOG with improved negation and control facilities [12]. MU-PROLOG delays negated calls until they are ground. In addition, the if-then-else predicate is delayed until the test is ground. Another interesting feature of GÖDEL is the abstract domains for polymorphically typed logic programs [13]. It is shown how specialized domains may be constructed for each type in the program. However, their formalism is rather complex, mainly due to the function declarations. More recent works such as the Ciao [14] present a multi-paradigm programming system that among others can verify that a program complies with its specification and can perform many optimizations.

However, our system differs from all the above in terms of both goals and techniques. More closely related works to ours are the following: [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] which recognize the necessity of optimizing Prolog implementations. The collected mode information in these analysis frameworks is intended to support optimizing compilers to generate efficient code. Mode inference in our approach supports the application of DT refinements to undefined predicates. The abstract domains range from simple two-value domains in [16] to more complex ones in [15], [17] and [24]. Aliasing has been considered as well by these approaches. Our abstract domain is a two-value domain without considering aliasing. Aliasing does not occur in the programs constructed by this schema-based methodology because the arguments of each predicate are distinct. That is why aliasing is not studied in this paper. This domain is sufficient for the needs of our program development method [6], [7] because only definite groundness is required by this development method. That is, definite groundness is required by some

arguments of the DT operations in order to compute successfully their task. Moreover, most of these works have little or no results implemented in current state of the art systems and there is more than one decade for these authors to present new research results.

The new features of our mode analysis compared to other works we are aware of are the following: first this mode analysis is used during program development supporting the program construction process and second this mode analysis is applied to logic programs that have been constructed partially. The mode inference procedure of this paper takes a partially constructed program and the expected call mode of the top-level predicate and computes a call and success mode for each program predicate. The call modes derived by the procedure are used to check that the DT operations are used consistently with their declared call modes.

3. Overview of our Schema-Based Logic Program Construction Method.

A short overview of the logic program construction method is presented in this section. The formal definitions of schemata and the instantiation operation are presented in appendix.

3.1. *The Idea of the Methodology*

The overall idea of our approach can be summarized as follows. The logic program construction methodology is based program design schemata and DTs. The program design schemata are abstractions of program design strategies. They are independent of the problem representation. This program construction method constructs highly structured programs. The structure of each program depicts the design decisions that have been taken for its construction. The constructed programs are called Schema-Instance Programs (SI-Programs). The size of such program is big even for small program but this is not disadvantage because the design philosophy of the methodology is to be supported by automatic and semi-automatic tools. The methodology assumes that the constructed programs can be made compact and faster automatically by a partial evaluation. The important gain from the construction of these highly structured program designs or programs is that the programs have clear structure so they can be updated easily by interactive tools and their correctness can be shown by semi-automatic tools as well. In other words each constructed program can be used for update and for proving its correctness and the one derived by partial evaluation will be used for running. Supporting tools for the methodology are a partial evaluator, an interactive verifier, a static analysis tool and an interactive update tool. Tests with partial evaluators [25], [26] have been made on the constructed programs with very good results [6]. Other tools like interactive verifiers have been constructed for the needs of the methodology [1].

In this paper small example programs have been used for illustration purposes in order to make clear to the reader the algorithm and in order to show the relation of the supporting theory with the method with simple and understandable examples. So the reader can see that this algorithm can be used in other problem domains as well. For

example, the method and the algorithm of this paper can be used for type inference. Large programs, i.e. non-trivial, programs have also been constructed with this methodology like unification algorithms, construction of depth-first search spanning tree of a directed graph, etc. [6].

A program construction methodology in order to be useful and practical has to be supported by a set of tools in order to make easy the development and the maintenance of large programs. The programmer can take the program design decisions and their implementation has to be done automatically or semi-automatically. In addition, the programmer would like to show the correctness of his program and his programs have to be efficient.

The program schemata of this logic program construction method satisfy the requirements which a schema-based approach must have [6]. These requirements are the following.

1. *Well-engineered schemata*. The schemata should not depend on the underlying data structures.
2. *Small set of schemata.* Small and fixed set of schemata are manageable by humans.
3. *Schema selection should be based on program design criteria.*
4. *Schemata which capture the expected mode and type of arguments.*
5. *Simple schema instantiation operation with instances without uninstantiated components.*
6. *Combination of instances of schemata.*
7. *Construction of non-trivial logic programs.*
8. *Construction of maintainable logic programs.*
9. *Correctness.* Provide support to the program correctness

The well-engineering of schemata is key to a schema-based approach. This approach consists of schemata independent of particular data types, while in most of the other methods, schemata are based on specific data types. In addition, the schemata in this approach represent algorithm design strategies that programmers follow in constructing logic programs. Each schema is an abstract representation of a specific algorithm design strategy. The application of a schema refinement to an undefined predicate corresponds to the realization of that strategy into the program. The design of such schemata has been based on semantic criteria, i.e. the algorithm design strategy that they represent. These algorithm design strategies are presented informally for the 5 schemata in Section 3.3. Another strong feature of our schemata that none of the previous methods has is the type and mode schemata of the predicate variables. We have also introduced into the schema-based method the well-known top-down stepwise refinement which we believe it is the programming style followed by logic programmers.

In summary, the general features of this method are the following [6].

1. This method is semi-automatic. We see it being supported by various development tools such as static analysis tools, partial evaluators, verification and other transformation tools.

2. This development method constructs initially an understandable software system.

This software system is used for verification. The constructed program is partially evaluated to get an efficient and more complex program for running.

3. The method uses 5 program schemata which are the main refinement rules. These refinement rules result in short and understandable derivations compared to derivations of other methods which are based on deductive synthesis and on formal transformations.

   - The *formal transformations* paradigm involves the development of a formal specification of the software system. The formal specification is transformed by correctness-preserving transformations into a program. *Formal transformations* have been applied in restricted application domains and at the level of individual programs. The fundamental difficulty with transformational systems is search control. At any point in the synthesis there are a number of possible transformations to apply. In addition, a derivation can require thousands of transformations to reach its solution.

   - The *deductive synthesis* paradigm is based on the observation that a program could be derived from a formal specification and the use of techniques such as resolution. The proof procedures in *deductive synthesis are* computationally very expensive. This constraint allows synthesis of short programs only. The derivation steps are small and as in the case of formal transformations the derivation steps may not correspond with the primitives that a programmer might use in writing a program

4. The method is built around abstract entities. The program schemata correspond to *procedural or structural abstractions.* Each program schema is a procedural or structural abstraction of the algorithm design strategy that it represents. The reusable components of this method are the program schemata and the DT operations. The program schemata are meta-level objects, i.e. they are terms in a meta-language for expressing polymorphic many-sorted first-order logic. Instances of schemata are derived when they are applied to undefined predicates. Therefore, the method reuses the program schemata, i.e. objects at the design level (meta-level), not their instances. The DT operations are reusable components at the code level. The DT operations are used for the refinement of undefined predicates as they have been specified.

5. We have developed a correctness proof method and a semi-automatic interactive tool for it that is based on formal specifications rather than testing.

6. Program schemata and DT operations enhance structure and maintainability of the constructed programs.

3.2. *Data types*

The construction of typed logic programs is considered in this method. The underlying language is *many sorted logic with parametric polymorphism* [9], [27]. The alphabet of the language contains disjoint infinite sets *V*, *F*, *P*, *A* and *C* representing respectively,

variables, functions, predicate symbols, parameters and constructors. Functions of arity 0 are constants and constructors of arity 0 are called *basic types*.

A *data type* is a set of data objects or values. Each type is associated with a number of operations which are applied to the values of the type. A DT in this method is considered to include both the set of values and the associated set of operations, each with its mode and type signature. The operations are formally defined by first-order logic (FOL) specifications.

In our language, types are denoted by type terms (or simply types). Type terms are defined similarly to first-order terms. That is, parameters correspond to individual *variables*, *basic types* correspond to *constants* and *type constructors* correspond to *functions*.

Definition 1. Let α be a *parameter*. Let B and C stand for *basic types* and *type constructors of arity* ≥ 1, respectively. A *type* is defined inductively as follows: A *parameter* α is a type. A *basic type* B is a type. If C is a type constructor of arity $n \geq 1$ and $\tau_1, \ldots, \tau_n$ are types, then $C(\tau_1, \ldots, \tau_n)$ is a type.

A type containing a variable is called a *parameterized* (or *polymorphic*) type. A type which does not contain parameters is called a *ground type*. A polymorphic type represents a collection of ground types. This collection of ground types is generated by instantiating its parameters to all possible ground types.

This method provides a set of DTs called *built-in DTs*. A programmer can define his own DTs called *user-defined DTs*. The available built-in DTs are *sequences*, *sets*, *multisets* (*bags*), *tuples* and the basic ones *Z* (*integers*), *N* (*natural numbers*), *Q* (*rational numbers*) and *strings*. The operations of these DTs have been implemented as predicates. A programmer can implement his own DTs by using this method. User-defined DTs are specified in terms of built-in DTs or previously defined user-defined DTs. The DT operations of user-defined DTs are implemented by using this method.

3.3. *Five design schemata*

A (*program design*) *schema* is a problem-independent algorithm design strategy. A design schema contains a fixed set of subparts or subgoals that have different instantiations for each problem to which they are applied. The modes which are used in this method are *g* and *u*. They stand for "ground" and "unknown" respectively. *g* identifies the arguments which are expected to be ground. These terms are characterized as "ground". An argument with any instantiation can be assigned mode *u*.

Definition 2. *A mode schema* for predicate variable P of arity n, denoted by *Mode(P)*, is $Mode(P) = m_1, \ldots, m_n$ where each m_i $(1 \leq i \leq n)$ is either *g* or *u*.

The mode schema for a predicate variable represents the expected use of the arguments, and it is used during schema refinement as a heuristic to partition the arguments into two groups [6].

Definition 3. A *typed, moded,* program *schema* is a typed program schema together with a mode schema for each predicate variable.

Five typed, moded program schemata have been designed in this logic program construction method. *Each program schema represents a problem-independent algorithm design strategy*. Their classification is based on the algorithm design strategy that each schema represents. That is, *Incremental*, *Divide-and-conquer*, *Subgoal*, *Case* and *Search*. These schemata can be applied to problems with different representations because they are defined to be independent of particular representations. *Logic programs in this method are constructed by composing instances of the 5 schemata*. This feature makes this method practical and flexible. We find this small set of schemata surprisingly expressive for constructing logic programs. Complex programs have been developed using this methodology like unification algorithms, construction of depth-first search spanning tree of a directed graph, etc.

Incremental. The *Incremental schema* assumes that the type τ of an argument in the tuple of the input data *u1* is defined inductively. The inductive definition of τ includes a constructor operation which builds an element of τ from another element of τ and some other data item. The *Incremental* schema processes one by one each piece of the input data and composes the results for each one to construct the solution. The schema is as follows.

Type Schemata
Type (Incr): α1 × α2
Type (Terminating): α1
Type (Initial _result):α1 × α2
Type (Deconstruction): α1 × α3 × α1
Type (Non_initial_result): α1 × α3 × α2 × α2
Mode Schemata
Mode (Incr): g, u
Mode (Terminating): g
Mode (Initial_result): g, u
Mode (Deconstruction): g, u, u
Mode (Non_initial_result): g, g, g, u
Incremental Schema
Incr(u1, u2) ← Terminating(u1) ∧ Initial _result (u1, u2)
Incr(u1, u2) ← ¬Terminating(u1) ∧ Deconstruction(u1, v1, v2) ∧
Incr(v2, v3) ∧ Non_initial_result (u1, v1, v3, u2)

In addition, each schema literal carries some semantic information which is related with its role in the schema. This semantic information is associated with informal guidance which supports the refinement of its instances. That is, design decisions are guided informally by the schema literals as well. For example, the informal guidance of the schema literals of *Incremental* schema is the following.

- *Terminating(u1):* Refining instances of this literal schema should aim at the construction of tests which determine if the primitive cases of the problem occur. The type of an argument from the tuple of input data, i.e. *u1*, must be defined inductively. The primitive cases of a problem are the ones which do not require further decomposition of the inductive DT in order to construct the desired solution.
- *Initial_result(u1, u2):* Refining instances of this literal schema should aim at the construction of solutions for the primitive cases of the problem. Instances of such refinements should construct the results into arguments of the tuple *u2* by using arguments from the tuple *u1*.
- *Deconstruction(u1, v1, v2):* Refining instances of this literal schema should aim at the decomposition of an argument from the tuple of input data, i.e. *u1*, whose type is defined inductively. This decomposition should split that argument into an element and the remaining part which are represented by the arguments *v1* and *v2* respectively. The remaining part, i.e. *v2*, is expected to be further decomposed. Note that the remaining part has type same as the type of the inductive argument, i.e. *u1*. On the other hand the component, i.e. *v1*, may have different type than the one of the inductive argument.
- *Non-initial-result(u1, v1, v3, u2):* Refining instances of this literal schema should aim at the construction of solutions for the general cases of the problem. These solutions are constructed into arguments of the tuple *u2* by using *u1*, *v1* and *v3*. Note, that the tuple of arguments *v3* represents the results from processing the remaining part of the argument which is decomposed, i.e. *v2*.

Divide-and-conquer. The *Divide-and-conquer* schema assumes that the type τ of an argument in the tuple of the input data is defined inductively. The *Divide-and-conquer* schema decomposes the problem representation in two subproblems of similar form to the initial one. The solution of the problem is constructed by composing the solutions of the subproblems together.

Subgoal. The *Subgoal* schema reduces the problem into a conjunction of two or more subproblems. The solutions of the simpler problems imply the solution of the original problem.

Case. The *Case* schema reduces a problem to two or more independent sub-problems. Each subproblem corresponds to a different case of the original problem. A solution to the initial problem is implied by each solution of its subproblems.

Search. A *state* of a problem is a particular configuration of the problem representation. The *space of states* of a problem is all possible configurations of a problem representation. *Operations* manipulate the problem representation. That is, they transform one state into another. The space of states is represented implicitly by a graph or tree. The *Search* schema performs search in the space of states of a problem. The search starts from the initial state, new states are produced by applying *operations* to it, and next *operations* are applied again to these states producing new states and so on. As the search proceeds a tree is constructed which is called the *search tree*. If there is no operation to be applied or if it is understood that an incorrect operation has been performed then control can backtrack to a previous node of the search tree and try another operation. The Search schema constructs the *search tree* in a stack called *search stack*. The search stack has an implicit representation of the state space which remains for searching. Backtracking is performed by using the search stack.

3.4. *The representation of schemata in knowledge base*

The program design schemata are presented in ground representation form [28]. The ground representation of *Incremental schema* follows. The predicate *design_schema*/2 has in the first argument the name of the schema "*incr*" and in the second argument the list of the identifiers of schema clauses, i.e. 1 and 2. The next two clauses of predicate *design_schema_clause*/2 have the ground representation of schema clauses of *Incremental schema* with identifier 1 and 2. In the ground representation of the *Incremental schema pvar(N1), var(N2)* and *tvat(N3)* where *N1, N2* and *N3* are positive integers stand for predicate variables, schema argument variables and type variables respectively.

```
design_schema(incr, [1,2]).
design_schema_clause(1,
     [[pvar(3), arg(var(1),g,tvar(1)), arg(var(2),u,tvar(2))],
      [pvar(1), arg(var(1),g,tvar(1))],
      [pvar(2), arg(var(1),g,tvar(1)), arg(var(2),u,tvar(2))]
     ]).
design_schema_clause(2,
    [[pvar(3), arg(var(1),g,tvar(1)), arg(var(2),u,tvar(2))],
     [not pvar(1), arg(var(1),g,tvar(1))],
     [pvar(4), arg(var(1),g,tvar(1)), arg(var(9),u,tvar(3)),
                                  arg(var(3),u,tvar(1))],
     [pvar(3), arg(var(3),g,tvar(1)), arg(var(4),u,tvar(2))],
     [pvar(5), arg(var(1),g,tvar(1)), arg(var(9),g,tvar(3)),
           arg(var(4),g,tvar(2)), arg(var(2),u,tvar(2))]
    ]).
```

3.5. *The construction process*

The construction process involves the application of a sequence of refinements. The sequence of refinements is obtained by traversing a tree structure, called a *refinement tree*, which is closely related to the program structure. Each refinement corresponds to a node of the tree. During the program construction process, the signatures and the modes of the program predicates are also derived.

Definition 4. Let *Prog* be a normal logic program. Let *p* be an n-ary predicate symbol in *Prog*. The predicate *p*/*n* is called an *undefined predicate* if there are no clauses whose heads contain the predicate symbol *p*.

Definition 5. A *refinement step* or simply a *refinement* is defined to be either of the following actions.
(i) The definition of an undefined predicate by instantiating a design schema. Such refinements are called *schema refinements*.
(ii) The definition of an undefined predicate by a DT operation or equality or the negation of one of these. Such refinements are called *DT refinements*.

Definition 6. Let *p*/*n* be a predicate with type *Type (p)* = $\tau_1 \times \ldots \times \tau_n$. Let $p(x_1, \ldots, x_n)$ be an atom where $x_1, \ldots, x_n$ are distinct variables. $p(x_1, \ldots, x_n)$ is called typed *completely general atom* (*cga*) for predicate *p*/*n*.

Let *p*/*n* be an undefined predicate with type $\tau_1 \times \ldots \times \tau_n$ and expected mode $m_1, \ldots, m_n$. Let $p(x_1, \ldots, x_n)$ be a typed cga where each x_i ($1 \leq i \leq n$) has type τ_i in $p(x_1, \ldots, x_n)$. Initially, the programmer gives a typed cga of the predicate that he wants to define, i.e. $p(x_1, \ldots, x_n)$, the type τ_i of each x_i in $p(x1, \ldots, xn)$ and the mode of *p*/*n*. The initial refinement is applied to this typed *cga*. The next refinements are applied to typed *cga*(*s*) of undefined predicates which are created by the initial refinement, and so on. Eventually, the undefined predicates are expected to be refined by DT refinements. The construction process is a successive top-down, left-to-right application of refinements until the construction of the desired SI-program is complete. A program is considered to be complete when all of its predicates are defined.

The construction process of a program is represented by a refinement tree. Each node of a refinement tree partial or complete represents either a schema-refinement or a DT refinement. The refinement tree is represented in ground form [28] using the predicates *node/2* and *node/3*. The predicates *node/2* and *node/1* are used to represent schema refinements and DT refinements respectively. They have the following general form.

- *node(refine(PredicateName, SchemaName), ListOfNewPredicateNames)*, where *PredicateName* is the name of the predicate to be refined. In the case of top level predicate, it has the representation in ground form of its arguments as well. For the other predicates it has just its name. *SchemaName* has the schema name which will be used for the refinement of the corresponding predicate.

ListOfNewPredicateNames is a list of new predicate names which will be used for the new predicates that will be created.

- *node(refine(PredicateName, dt_eq, GroundRepresentationOfDToperation))* where *PredicateName* is the name of the predicate to be refined. *dt_eq* is a constant which indicates that the refinement type is DT operation or equality. *GroundRepresentationOfDToperation* has the representation of the DT operation or the equality in ground form.

Fig. 2 illustrates the complete refinement tree in graphical form for predicate sum/2 which is used for illustrating the presentation of the method. The corresponding refinement tree for sum/2 in ground form is shown in Subsection 3.5. The construction of a program corresponds to an in-order, left-right traversal of the refinement tree. The ground representation of the refinement tree is input to a meta-interpreter which performs the in-order, left-right traversal of the refinement tree.

The constructed logic programs satisfy declared input-output modes when run using the standard left-right depth-first computation rule. Mode inference is performed during program constructions. The inferred modes of undefined predicates and the declared ones of the DT operations guide the argument matching of refinements by DT operations. This argument matching is also supported by the derived types of undefined predicates and the declared ones of the DT operations. Fig. 1 illustrates the program development process and the use of mode inference during program construction.

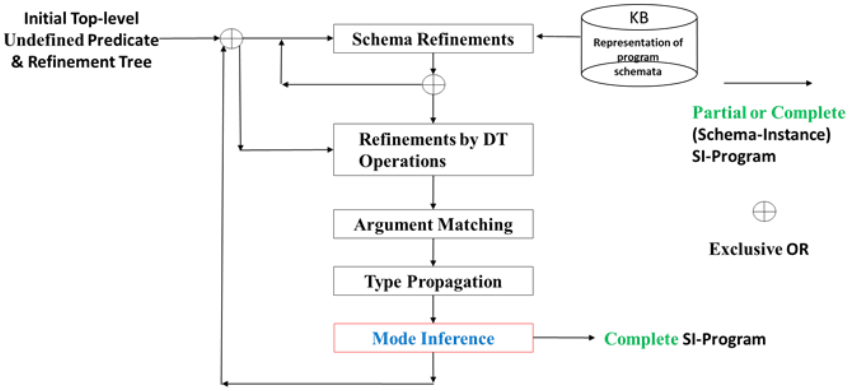

Fig. 1. Mode inference during program construction.

3.6. *Example of a Refinement Tree and the Constructed Program*

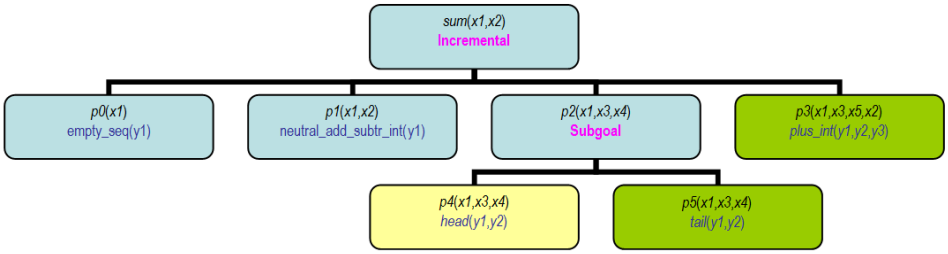

Fig. 2. Complete refinement tree for predicate *sum*/2**.**

Let *sum/2* be an undefined predicate with *Type(sum)* = *seq*($\alpha 1$) × $\alpha 1$ and expected mode *Mode(sum) = (g, u)*. Let us assume that a programmer wants to construct a program for *sum/2*. Let *sum(x1, x2)* be a typed *cga*. The predicate *sum(x1, x2)* is *true* iff *x2* is the sum of integers in sequence *x1*. The "*expected modes*" are modes which are not derived by the mode inference procedure. They are used to guide the application of schemata to undefined predicates. The top predicate variable of the 5 schemata is binary. Such a binary schema can be applied to an undefined predicate of any arity by using the "*expected modes*" of the arguments of the undefined predicate. "*Expected modes*" and inferred modes do not necessarily agree. In fact, an "*expected mode*" *g* might be *u* at run time, and an "*expected mode*" *u* might be *g* at run time. The "*expected modes*" only provide a heuristic for splitting arguments into two groups, i.e. input and output groups. However, the heuristic is safe in the sense that an argument that is expected to be *g* will either be inferred to be *g*, or not used at all.

The complete refinement tree in graphical form for this example is illustrated in Fig. 2. The ground representation of the refinement tree for *sum*/2 is shown bellow as well. In the ground representation of the refinement tree the argument of each predicate is represented by a term of the form *arg(Term, Mode, Type)* where *Term* is the argument term. *Mode* is the call mode in case of DT operation and the expected mode in case of the top-level predicate. *Type* is the argument data type. In addition, *var(N1)* and *tvat(N2)* where *N1* and *N2* are positive integers stand for variables and type variables respectively.

Schema refinements

```
node(refine(sum(arg(var(11),g,seq(tvar(1))),arg(var(12),u,tvar(1))),
                        incr), [p0,p1,p2,p3]).
node(refine(p2, subgoal_B), [p4,p5]).
```

DT refinements

```
node(refine(p0, dt_eq, empty_seq(arg(var(1),u,seq(tvar(1))))  )).
node(refine(p1, dt_eq, neutral_add_subtr_int(arg(var(1),u, int))  )).
node(refine(p4, dt_eq,
        head(arg(var(1),g,seq(tvar(1))),arg(var(2),u,tvar(1)))  )).
node(refine(p5, dt_eq,
```

tail(arg(var(1),g,seq(tvar(1))),arg(var(2),u,seq(tvar(1))))).
node(refine(p3, dt_eq,
plus_int(arg(var(1),g,int), arg(var(2),g,int), arg(var(3),u,int)))).

Note that the signatures and the modes of the DT operations *head*/2, *tail/2*, *empty_seq/1*, *neutral_add_subtr*/1 and *plus_int/3* are defined by the user. The DT operations are components of the construction methodology and they are built-in. The signature and the "expected" mode of the top-level predicate *sum/2* are given by the user because it is the one which the user wants to define. The expected and derived modes for this predicate are same. The predicates *p0*/1, *p1*/2, *p2*/3, *p3*/4, *p4*/3 and *p5*/3 are constructed by the system and their signatures and modes are derived by the system as well. The mode inference procedure derives modes for all predicates of the program. For the DT operations it simply verifies that the inferred modes are consistent with the declared ones. The constructed SI-program is as follows.

Signatures

tail: seq(α_1) × seq(α_1)
neutral_add_subtr_int: int
empty_seq: seq(α1)
head: seq(α1) × α1
plus_int: int × int × int
sum: seq(int) × int
p0: seq(int)
p1: seq(int) × int
p4: seq(int) × int × seq(int)
p2: seq(int) × int × seq(int)
p3: seq(int) × int × int × int
p5: seq(int) × int × seq(int)

Modes

tail: g, u
neutral_add_subtr_int: u
empty_seq: u
head: g, u
plus_int: g, g, u
sum: g, u
p0: g
p1: g, u
p4: g, u, u
p2: g, u, u
p3: g, g, g, u
p5: g, g, u

Refinements

1. Refinement 1: Incremental schema is applied to *sum*(*x1*, *x2*).
   sum(x1, x2) ← *p0(x1)* ∧ *p1(x1, x2)*
   sum(x1, x2) ← ¬ *p0(x1)* ∧ *p2(x1, x3, x4)* ∧ *sum(x4, x5)* ∧ *p3(x1, x3, x5, x2)*
2. Refinement 2: The DT predicate *empty_seq*/*1* refines *p0*(*x1*).
   p0(x1) ← *empty_seq(x1)*
3. Refinement 3: The DT *predicate_neutral_add_subtr_int*/*1* refines *p1*(*x1*, *x2*).
   p1(x1, x2) ← *neutral_add_subtr_int(x2)*
4. Refinement 4: Subgoal schema is applied to *p2(x1, x3, x4)*.
   p2(x1, x3, x4) ← *p4(x1, x3, x4)* ∧ *p5(x1, x3, x4)*
5. Refinement 5: The DT predicate *head*/*2* refines *p4(x1, x3, x4)*.
   p4(x1, x3, x4) ← *head(x1, x3)*
6. Refinement 6: The DT *predicate_tail*/*2* refines *p5(x1, x3, x4)*.
   p5(x1, x3, x4) ← *tail(x1, x4)*
7. Refinement: The DT predicate *plus_int*/*2* refines *p3(x1, x3, x5, x2)*.
   p3(x1, x3, x5, x2) ←*plus_int(x3, x5, x2)*

Definitions of DT and Equality Predicates

empty_seq([])
neutral_add_subtr_int(0)
head([h| t], h)
tail([h | t], t)
plus_int(x_1, x_2, x_3) ← x_3 *is* $x_1 + x_2$

3.7. *Partial Evaluation of the Constructed Programs*

Partial evaluation in logic programming is described in the following way in [29]. "Given a program *Prog* and a goal *G*, partial evaluation produces a new program *Prog*′ *which* is *Prog* specialized to goal *G*. *Prog* should have the same semantics as *Prog* with respect to *G that* is, the correct and computed answers for *G* in *Prog*′ should be equal to answers for *G* in *Prog*. It is also expected that *G* should be executed more efficiently in *Prog*′ than in Prog".

In this logic program construction methodology, partial evaluation is applied to SI-programs in order to reduce their size and to make them possibly more efficient. The effect of partially evaluating an SI-program with respect to a goal is to remove the design layer which this method creates thus intertwining it with the implementations of DT operations.

A partial evaluator, in general, takes as input a program and a partially instantiated goal. It returns a program specialized for all instances of that query. For the aims of this logic program construction methodology we do not like programs generated by a partial evaluator to be more specialized than the original one. We would like the residual program to be able to compute all queries that the initial one can compute. Let *Prog* be an

SI-program. Let *p/n* be the top-level predicate in *Prog*. The program *Prog* is partially evaluated with respect to a goal of the form $G = \leftarrow p(t_1,..., t_n)$ where $p(t_1,... ,t_n)$ is a typed cga. In this way the program *Prog΄* generated by partial evaluation of program *Prog* will be equivalent to the initial one. That is, *Prog* and *Prog΄* will have the same semantics. In addition, *Prog΄* is expected to have fewer clauses. It also expected to be more efficient.

The partial evaluator that we use for the needs of this logic program construction methodology is SP [25]. SP is designed to specialize declarative normal logic programs.

We partially evaluated using SP a set of SI-programs. Then we run each SI-program and the ones generated by partial evaluation using the same goal in order to measure their time and space requirements. For the SI-Program of predicate sum/2 shown in Section 3.6 and for goal $G = \leftarrow sum\ (Q, S)$ the residual program, *Prog'*, returned by the partial evaluator SP is the following program.

```
sum([],0) :- true.
sum([X1|X2],X3) :- \+fail, sum(X2,X4), X3 is X4+X1.
```

The program size and the time and space evaluation results of this residual program derived by SP and the one constructed by this methodology, shown in Section 3.6, are illustrated in Table 1. The time and space results have been derived by running both programs with the same goal.

Table 1. Number of clauses, time and space requirements of the residual program and the one constructed by our methodology for predicate sum/2.

	SI-Program	**SP Program**
N umber of clauses	13	2
Runtime (100 runs)	0.88 sec	0.03 sec
Global stack size (1 run)	6012 Kb	2748 Kb

The most likely reasons for these optimizations are the following.

1. The removal of single-clause definitions. That is, all predicates defined by Subgoal schema consist of single clauses.
2. The pruning of failing branches. This reduces the number of alternatives.
3. Better indexing of clauses because data structures (e.g. [] and [_|_]) are pushed into the clause heads. The clauses of predicates are indexed according to functors of the arguments in their heads.

4. Abstract Analysis Framework.

In this section, we present the abstract domain. The discussion is illustrated with an example. Partially ordered structures, least fixed points and abstract interpretation notions are presented in appendix.

Abstract interpretation is a formal framework for static analysis of the run-time properties of logic programs [15], [30]. The meaning or semantics of programs is analyzed in a finite abstract domain which approximates an infinite concrete domain. Sets of elements from the concrete domain are approximated by single elements of the abstract domain. An abstract interpretation is defined by an abstract domain and abstract semantic functions. The abstract semantic functions define an abstract denotation of the program, which should be a safe approximation to the concrete denotation.

4.1. *Abstract Domain.*

Definition 7. *Abstract terms* are the terms of the set $D_{aTerm} = \{g,u\}$

Let Pr be an SI-program excluding the definitions of DT operations. The program variables in the abstract interpretation of a logic program Pr are bound to abstract terms. The elements of the set D_{aTerm} form a total order with respect to the inclusion relation $\sqsubseteq_{aTerm}$. That is, g "*is included by*" u which is denoted by $g \sqsubseteq_{aTerm} u$.

Definition 8. $(D_{aTerm}, \sqsubseteq_{aTerm})$ is a pointed *cpo*. The *lub* ($\sqcup_{aTerm}$) and *glb* ($\sqcap_{aTerm}$) of (D_{aTerm}, $\sqsubseteq_{aTerm}$) are defined as $g \sqcup_{aTerm} u = u$ and as $g \sqcap_{aTerm} u = g$ respectively.

Definition 9. An abstract substitution λ is a finite set of the form $\{x_1/m_1, \ldots, x_n/m_n\}$ where x_i $(1 \leq i \leq n)$ are distinct variables and m_i are abstract terms.

In the following, λ and μ will denote abstract substitutions.

Definition 10. An abstract atom for predicate p/n is an atom of the form $p(m_1, \ldots, m_n)$ where $m_i \in D_{aTerm}$ $(1 \leq i \leq n)$.

In the following, I possibly superscripted will denote an abstract atom. An abstract atom $p(m_1, \ldots, m_n)$ for predicate p/n which occurs at call time is called an abstract call atom. An abstract atom $p(m_1, \ldots, m_n)$ for predicate p/n which occurs at success time is called abstract success atom. A partial ordering on the abstract atoms of a predicate p/n, denoted by $\sqsubseteq_p$, is defined as follows.

Definition 11. Let $p(m_1^1 \ldots, m_1^n)$ and $p(m_1^2, \ldots, m_n^2)$ be two abstract atoms for predicate p/n where $m_i^1, m_i^2 \in D_{aTerm}$ $(1 \leq i \leq n)$.

$$p(m_1^1, \ldots, m_n^1) \sqsubseteq_p p(m_1^2, \ldots, m_n^2) \textbf{ iff } m_1^1 \sqsubseteq_{aTerm} m_1^2 \wedge \ldots \wedge m_n^1 \sqsubseteq_{aTerm} m_n^2$$

Let D_p stand for the set of all possible abstract atoms for a predicate p/n. $(D_p, \sqsubseteq_p)$ is a pointed *cpo*. The top element of D_p is the atom with arguments only u's, i.e. $p(u, \ldots, u)$. The bottom element of D_p is the atom with arguments only g's, i.e. $p(g, \ldots, g)$. The *lub* ($\sqcup_p$) for any two elements in D_p is defined as follows.

Definition 12. Let $p(m_1^1 \ldots, m_1^n)$ and $p(m_1^2, \ldots, m_n^2)$ be two abstract atoms for predicate p/n where $m_i^1, m_i^2 \in D_{aTerm}$ $(1 \leq i \leq n)$.

$$p(m_1^1, \ldots, m_n^1) \sqcup_p p(m_1^2, \ldots, m_n^2) = p(m_1, \ldots, m_n)$$

where $m_i = m_i^1 \sqcup_{aTerm} m_i^2 (1 \leq i \leq n)$.

For example, $p(g, g, u, g, u) \sqcup_p p(g, u, u, g, g) = p(g, u, u, g, u)$. The disjoint union of D_p for all predicates that may occur in a logic program with an additional bottom element $\perp_{aAtom}$ is denoted by D_{aAtom}. *Psymb* stands for the set of predicate symbols of the predicates in logic program *Pr*.

Definition 13. Let I^1 and I^2 be two elements of the abstract domain D_{aAtom}. The ordering $\sqsubseteq_{aAtom}$ on D_{aAtom} is defined as follows.

$$I^1 \sqsubseteq_{aAtom} I^2 \textbf{ iff } I^2 = \perp_{aAtom} \text{ or } I^2 \sqsubseteq_p I^2 \text{ for some } p \in Psymb$$

D_{aAtom} is a pointed *cpo* with ordering $\sqsubseteq_{aAtom}$.

Definition 14. Let I^1 and I^2 be two elements of the abstract domain D_{aAtom}.
The *lub* $I^1 \sqcup_{aAtom} I^1$ is defined as follows.

$$I^1 \sqcup_{aAtom} I^2 = \begin{cases} I^1 & \text{if } I^2 \perp_{aAtom} \\ I^2 & \text{if } I^2 \perp_{aAtom} \\ I^1 \sqcup_p I^2 & \end{cases}$$

4.2. *Illustration of the Abstract Domain through a Simple Example.*

We will use a small program in order to illustrate the definitions and concepts of our abstract analysis framework. The predicate *p(x1, x2, x3, x5)* is true if *x5* is the concatenation of sequences *x1*, *x2* and *x3*. Let *Pr* be the following program for predicate p/4 which has been constructed by this logic program construction methodology.

Signatures

concat: seq(a1) × seq(a1) × seq(a1)
p: seq(a2) × seq(a2) × seq(a2) × seq(a2)
p0: seq(a2) × seq(a2) × seq(a2) × seq(a2) × seq(a2)
p1: seq(a2) × seq(a2) × seq(a2) × seq(a2) × seq(a2)

Modes

concat: g, g, u
p: g, g, g, u
p0: g, g, g, u, u
p1: g, g, g, g, u

Program clauses

$p(x1, x2, x3, x5) \leftarrow p0(x1, x2, x3, x4, x5) \wedge p1(x1, x2, x3, x4, x5)$
$p0(x1, x2, x3, x4, x5) \leftarrow concat(x1, x2, x4)$
$p1(x1, x2, x3, x4, x5) \leftarrow concat(x4, x3, x5)$

Abstract domain of *Pr*.
$D_{concat} = \{concat(g, g, g, g), concat(g, g, g, u), .., concat(g, u, u, u), concat(u, u, u, u)\}$,
$D_p = \{p(g, g, g, g), p(g, g, g, u), .., p(g, u, u, u), p(u, u, u, u)\}$,
$D_{p0} = \{p0(g, g, g, g, g), p0(g, g, g, g, u) .., p0(g, u, u, u, u), p0(u, u, u, u, u)\}$,
$D_{p1} = \{ p1(g, g, g, g, g), p1(g, g, g, g, u), .., p1(g, u, u, u, u), p1(u, u, u, u, u)\}$
$D_{aAtom} = \{\perp_{aAtom}\} \cup D_{concat} \cup D_p \cup D_{p0} \cup D_{p1}$
$(D_{concat}, \sqsubseteq_{concat})$, $(D_p, \sqsubseteq_p)$, $(D_{p0}, \sqsubseteq_{p0})$, $(D_{p0}, \sqsubseteq_{p0})$ and $(D_{aAtom}, \sqsubseteq_{aAtom})$ are pointed *cpo*. They are shown graphically in Fig. 3, Fig. 4 and Fig. 5.

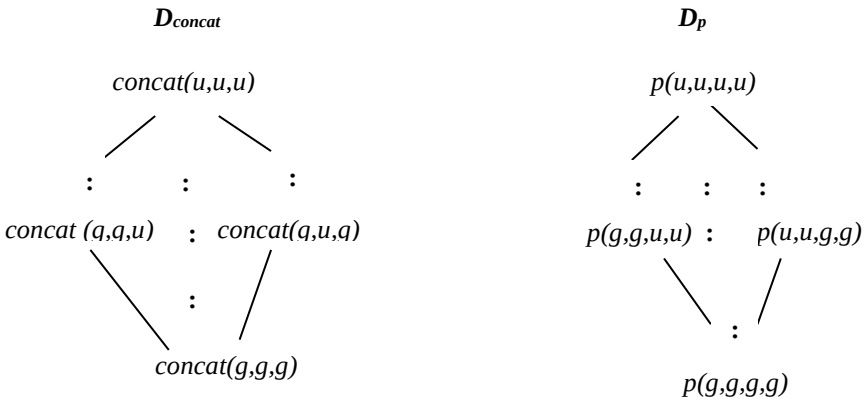

Fig. 3. $(D_{concat}, \sqsubseteq_{concat})$ and $(D_p, \sqsubseteq_p)$ are pointed *cpo*.

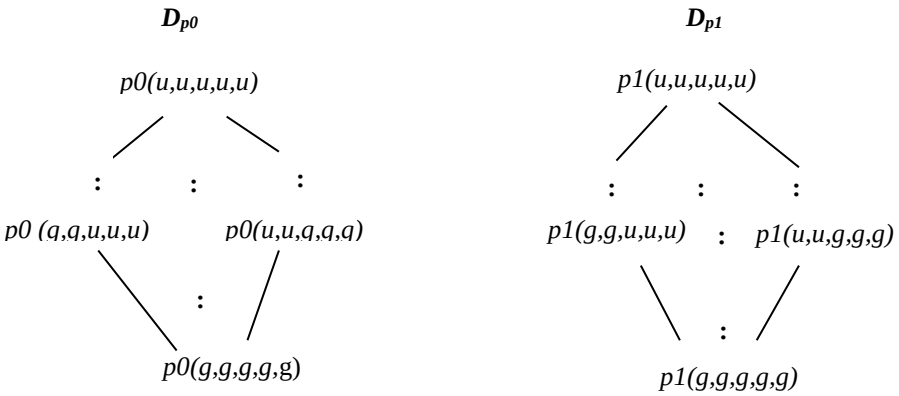

Fig. 4. $(D_{p0}, \sqsubseteq_{p0})$ and $(D_{p0}, \sqsubseteq_{p0})$ are pointed *cpo*.

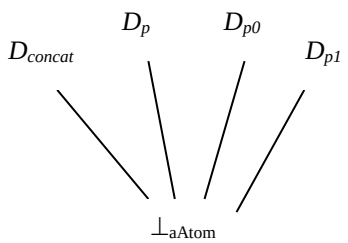

Fig. 5. (D_{aAtom}, $\sqsubseteq_{aAtom}$) is a pointed *cpo*.

5. Mode analysis.

First, the abstract operations of matching (unification) and substitution composition are presented. Then, the mode analysis algorithm is shown. Next, the program meaning is discussed through an example. Finally, it is shown that SI-programs are well-moded.

In this abstract analysis framework, a sequence of tables represents program descriptions. The initial table is $T_0 = calls(Pr) \cup answers(Pr)$ where $calls(Pr)$ consists of the call mode of the top-level predicate of program Pr and $answers(Pr)$ consists of the success mode of all DT operations of program Pr. $d \in D$ is the table description of program Pr which is not changed by the abstract evaluation of Pr with respect to the specified abstract goal of the top-level predicate.

The mode analysis algorithm computes a sequence of tables representing program descriptions. Each program description consists of two sets of abstract atoms, representing the call and success patterns computed so far. The initial table consists of the initial call pattern of the top-level predicate, and the success patterns for all DT operations including equality with all completely ground.

The notation $call(p(t_1, \ldots, t_n))$ where $p(t_1, \ldots, t_n)$ is an abstract atom, refers to the abstract *call* atom $call(p(m_1, \ldots, m_n))$ corresponding to p. Similarly, $ans(I)$ refers to the abstract success atom for I. With this notation, a program description is represented as a set of expressions of the form $call(I)$ and $ans(I)$ where a given predicate may appear in one call and one answer expression.

An operation for matching an atom to an abstract atom is needed

Definition 15. Let $p(t_1, \ldots, t_n)$ be an atom and $p(m_1, \ldots, m_n)$ be an abstract atom for predicate p/n. Terms $t_1, \ldots, t_n$ are distinct variables and $m_1, \ldots, m_n$ are abstract terms. $match(p(t_1, \ldots, t_n), p(m_1, \ldots, m_n))$ is an operation that is defined as follows.

$$match(p(t_1, \ldots, t_n), p(m_1, \ldots, m_n)) = \{t_1/m_1, \ldots, t_n/m_n\}$$

For example, let's assume the atom $p(x_1, x_2, x_3)$ and the abstract atom $p(g, g, u)$ then $match(p(x_1, x_2, x_3), p(g, g, u)) = \{x_1/g, x_2/g, x_3/u\}$.Matching substitutions can be composed, making use of the $\sqcap_{aTerm}$ operation.

Definition 16. Let $\lambda_1 = \{x_1/m_1^1, \ldots, x_{n1}/m_{n1}^1\}$ and $\lambda_2 = \{y_1/m_1^2, \ldots, y_{n2}/m_{n2}^2\}$ be two abstract substitutions. The composition of abstract substitutions is denoted by $\lambda_1 \circ_\alpha \lambda_2$. It is defined as follows.

$$\lambda_1 \circ_\alpha \lambda_2 = \{z/m \mid x_i/m_i^1 \in \lambda_1 \wedge y_j/m_j^2 \in \lambda_2 \wedge x_i = y_j = z \wedge m = m_i^1 \sqcap_{aTerm} m_j^2\}$$
$$\cup \{x_i/m_i^1 \mid x_i/m_i^1 \in \lambda_1 \wedge x_i \notin dom(\lambda_2)\}$$
$$\cup \{y_j/m_j^2 \mid y_j/m_j^2 \in \lambda_2 \wedge y_j \notin dom(\lambda_1)\}$$

where $(0 \le i \le n_1)$ and $(0 \le j \le n_2)$.

For example, if $\lambda_1 = \{x_1/g, x_2/u, x_3/u\}$ and $\lambda_2 = \{x_1/g, x_3/g\}$ then $\lambda_1 \circ_\alpha \lambda_2 = \{x_1/g, x_2/u, x_3/g\}$.

5.1. *Mode Analysis Algorithm.*

Let *Pr* be an SI-program. Let $p(m_1, \ldots, m_n)$ be the call mode of the top-level predicate. Let $dt_1/n_1, \ldots, dt_k/n_k$ be the DT operations in the SI-program *Pr*. T_0 is the initial program description. This algorithm computes successively program descriptions starting from T_0, and terminates when no change occurs in two successive program descriptions. The sets *calls*(*Pr*, T_i), *answers*(*Pr*, T_i) have the call and answer modes of the predicates that have changed in the ith iteration. The new program description represented by next table T_{i+1} is evaluated by the *lub* of the corresponding entries in the table T_i and the ones in the sets *calls*(*Pr*, T_i) and *answers*(*Pr*, T_i). The algorithm terminates when no changes have occurred in the ith iteration.

$T_0 := \{call(p(m_1, \ldots, m_n))\} \cup \{ans(dt_1(g, \ldots, g)), \ldots, ans(dt_k(g, \ldots, g))\}$
i := 0
repeat
  $T_{i+1} := lub(T_i, calls(Pr, T_i), answers(Pr, T_i))$
  $i := i + 1$
until $T_i = T_{i-1}$
where
calls(*Pr*, T_i) =

$$\left\{ call(p_j(m_1, ..., m_k)) \;\middle|\; \begin{array}{l} h \leftarrow l_1,\ldots,p_j(x_1,\ldots,x_k),\ldots,l_n \in Pr, \\ \quad \{call(h), ans(l_1), ... , ans(l_{j-1})\} \subseteq T_i \\ \Phi = match(h, call(h)) \circ_\alpha match(l_1, ans(l_1)) \\ \quad \circ_\alpha ... \circ_\alpha match(l_{j-1}, ans(l_{j-1})) \cup call(p_j(m_1, ..., m_k)) \\ \{x/u \mid x \text{ occurs in} \{x_1, ..., x_k\} \text{ and does not occur in } l_1, ..., l_{j-1}\} \\ \{x_1/m_1, ... , x_k/m_k\} \subseteq \Phi \end{array} \right\}$$

Informally, the function *calls*(*Pr*, T_i) means derive new abstract calls, i.e. $call(p_j(m_1, ..., m_k))$, by running from program *Pr* the clause "$h \leftarrow l_1,\ldots$" for the goal "$call(h)$" and by using the program description of table T_i. The running of the program in the abstract domain is same as in the concrete domain apart the use of the abstract operations, i.e. unification (*match/2*), substitution composition ($\circ_\alpha$) and substitution application ($\{x_1/m_1, ... , x_k/m_k\} \subseteq \Phi$).

$$answers(Pr, T_i) =$$

$$\left\{ ans(p(m_1, ..., m_k)) \;\middle|\; \begin{array}{l} p(x_1,\ldots, x_k) \leftarrow l_1,\ldots,l_n \in Pr \\ \{call(p(m_1',..., m_k')), ans(l_1), ..., ans(l_n)\} \subseteq T_i, \\ \Phi = match(p(x_1,\ldots, x_k), call(p(m_1', ..., m_k'))) \circ_\alpha \\ match(l_1, ans(l_1)) \circ_\alpha ... \circ_\alpha match(l_n, ans(l_n)), \\ \{x_1/m_1, ..., x_k/m_k\} \subseteq \Phi \end{array} \right\}$$

Informally, the function *answers*(*Pr*, T_i) means derive new abstract answers, i.e. *ans*(*p*(m_1, ..., m_k)), by running from program *Pr* the clause "*p*(x_1,…, x_k) ← l_1,…,l_n" for the goal "*call*(*p*(m_1',..., m_k'))" and by using the program description of table T_i. The running of the program in the abstract domain is same as in the concrete domain apart the use of the abstract operations, i.e. unification (*match/2*), substitution composition ($\circ_\alpha$) and substitution application ($\{x_1/m_1, ... , x_k/m_k\} \subseteq \Phi$).

Note that negative literals are ignored in both *calls*(*Pr*, T_i) and *answers*(*Pr*, T_i). Negative literal are ignored because they don't affect the instantiation of arguments. The upper bound on program descriptions is as follows.

$$lub(T_i, calls(Pr, T_i), answers(Pr, T_i)) =$$
$$\{call(I^1 \sqcup_{aAtom} I^2) \mid call(I^1) \in T_i \text{ and } call(I^2) \in calls(Pr, T_i)\} \cup$$
$$\{ans(I^1 \sqcup_{aAtom} I^2) \mid ans(I^1) \in T_i \text{ and } ans(I^2) \in answers(Pr, T_i)\}$$

In this definition, it is assumed that *call*(*I*) = *call*($\perp_{aAtom}$) for a predicate *p* if there is no expression in T_i of form *call*(*p*(m_1, ..., m_n)). A similar comment applies for *ans*(*I*). In the implementation, each iteration uses only the information added on the previous iteration. This makes the computation more efficient. Termination occurs when nothing is added on some iteration. There is one call atom and success atom in the table per predicate. The call and success atoms for each predicate are the *lub* of all call and success atoms that have occurred for that predicate during the computation. In addition, the analysis of partial programs is performed by assuming that all undefined predicates have success modes consisting only of *u*'s for all arguments.

5.2. *Program Meaning with an Example*

The relation of minimal function graph concept to program analysis is defined as follows [5]. "The meaning of a program is represented as a function from input to output. Its mfg with respect to some given input represents that part of the total meaning of the program which is needed for computing a result for the given input.". The basic notions for mfg are presented in the appendix.

The semantics in this static analysis framework are based on the view of predicates as partial functions from sets of activation instances, i.e. predicate calls, to sets of result instances, i.e. predicate answers. We would like to define the meaning of program *Pr* in the abstract domain, i.e. D_{aAtom}. We see the relations of a program *Pr* as functions. The domain of each function consists of all possible call modes and its range consists from all possible success modes. The discussion will be illustrated with the simple example of the

program for predicate p/4 of Subsection 4.2. The functions of the predicates *concat/2*, *p/4*, $p_0/5$, $p_1/5$ are f_{concat}, f_p, f_{p0}, and f_{p1} respectively. They are defined as follows: f_{concat}: $D_{\text{concat}} \rightarrow D_{\text{concat}}$, f_p: $D_p \rightarrow D_p$, f_{p0}: $D_{p0} \rightarrow D_{p0}$ and f_{p1}: $D_{p1} \rightarrow D_{p1}$. These functions are partial functions in the abstract domain D_{aAtom} because D_{concat}, D_p, D_{p0} and D_{p1} are subsets of D_{aAtom}. The definitions of these functions in the abstract domain are as follows, f_{concat}, f_p, f_{p0}, f_{p1}: $D_{\text{aAtom}} \rightarrow D_{\text{aAtom}}$. The tuple of functions f_{concat}, f_p, f_{p0}, f_{p1} is the functional F which makes the meaning of program *Pr*. A fixed point of this functional is evaluated which is a minimal function graph of the total meaning of program restricted to an input query. The input query is a call to the top-level predicate. This functional is represented by a table T which has an entry for each function. For each function f_{concat}, f_p, f_{p0}, f_{p1} there is one element from its domain, i.e. the call of predicate or activation instance, and one element from its range, i.e. the answer of the predicate or result instance. If a new element either from the domain or from the range of a function is created during program evaluation, apart the one in the table, then the *lub* of these entries is taken.

The description of program *Pr* is represented by a table T which consists from the set of calls and the set of answers of predicates, i.e. $T = calls(Pr) \cup ansers(Pr)$. The set of activation instances of program *Pr* is *calls*(*Pr*) which in each iteration starts from the call of the top-level predicate. The meaning of a program in each iteration is represented by a table T_i $(0 \leq i \leq n)$ where T_n *is a fixpoint*. The initial table for program *Pr* is $T_0 = calls(Pr) \cup answer(Pr) = \{\text{call}(p(g, g, g, u))\} \cup \{\text{ans}(concat(g, g, g))\}$. Predicates *p0/5*, *p1/5* and *concat/3* which do not have entry in the set *calls(Pr)*, an entry $call(\perp_{aAtom})$ is assumed for each one to be in the set *calls*(*Pr*). Similarly, predicates *p/4*, *p0/5* and *p1/5* do not have entry in the set *answers(Pr)*, an entry $ans(\perp_{aAtom})$ is assumed for each one as well in the set *answers(Pr)*. The fixpoint, i.e. T_n, is $T_n = calls(Pr) \cup answer(Pr) = \{\text{call}(p(g, g, g, u)), call(concat(g, g, u)), \text{call}(p0(g, g, g, u, u)), \text{call}(p1(g, g, g, g, u))\} \cup \{\text{ans}(p(g, g, g, g)), ans(p0(g, g, g, g, u)), \text{ans}(p1(g, g, g, g, g)), \text{ans}(concat(g, g, g))\}$. T_n is essentially a *minimum function graph* semantics of program *Pr*. It can be seen as the *total function graph (tfg)* semantics restricted to the call instances reachable from the specified query set $calls(Pr) = \{call(p(g, g, g, u))\}$.

5.3. *SI-Programs are well-moded*

The abstract analysis framework is a safe approximation of the concrete one. The constructed logic programs are statically executed in the abstract domain which is a safe approximation of the concrete domain and the results from that abstract evaluation are the derived modes. Therefore, the well-modeness is derived by construction. The well-modeness is formally stated by the next definitions and proposition.

Definition 17. Let $G^c = \leftarrow p(t_1, \ldots, t_n)$ be a negative literal where t_i $(1 \leq i \leq n)$ are (concrete) terms. G^c is called a concrete goal for predicate p/n.

Definition 18. Let $p(t_1,..., t_n)$ be an atom where t_i $(1 \leq i \leq n)$ are terms. Let $m_1, \ldots, m_n$ be the (call) mode of p/n. The (call) mode of $p(t_1,..., t_n)$ is subsumed by the mode of p/n if

the following hold. If m_i = g then t_i ($1 \leq i \leq n$) is a ground term. If m_i = u then t_i ($1 \leq i \leq n$) is any term.

Definition 19. Let *Pr* be an SI-Program with top-level predicate *p*/*n*. Let $G^c = \leftarrow p(t_1, \ldots, t_n)$ be a concrete goal where the call mode of $p(t_1, \ldots, t_n)$ is subsumed by the mode of *p*/*n*. The SI-program *Pr* is *well-moded* if the evaluation of goal G^c generates for each predicate of *Pr* calls whose modes are subsumed by the mode of its predicate.

Proposition 1. The mode analysis algorithm produces well-moded SI programs.

PROOF. The modes of the predicates of an SI-program *Pr* are validated by static mode analysis with respect to an abstract goal same as the mode of the top-level predicate *p*/*n*. The run-time calls of each predicate of *Pr* with respect to goal G^c are subsumed by the mode of its predicate because the goal G^c is subsumed by the mode of the top-level predicate *p*/*n*. Hence *Pr* is well-moded. □

6. Illustration of the Approach with an Example

This example illustrates the following features of this approach. First, it shows the mode analysis of a partial SI-program. Second, it shows the argument matching of DT operations with the ones of the predicate they define. The detection of an error after matching inappropriate arguments is also discussed. Finally, it shows that the declared modes as defined by the DT operations are consistent with the inferred runtime modes.

The predicate *max_seq*(*x1*, *x3*) where *Type*(*max_seq*) = *seq*(*int*) × *int* and *Mode*(*max_seq*) = (*g*, *u*) is *true iff x3* is the maximum element from the elements of sequence *x1*.

Let us assume that we have constructed the following partial logic program for the predicate *max_seq*/*2*. Note that in this example predicates p3/4 and p9/3 are unrefined and they are not reachable by the mode inference procedure because of that the system assigns mode "undefined" in their arguments.

Signatures

empty_seq: *seq(a1)*
tail: *seq*(*a1*) × *seq*(*a1*)
head: *seq*(*a1*) × *a1*
max_seq: *seq*(*int*) × *int*
p0: *seq*(*int*)
p1: *seq*(*int*) × *int*
p2: *seq*(*int*) × *a2* × *seq*(*int*)
p3: *seq*(*int*) × *a2* × *int* × *int*
p4: *seq*(*int*) × *seq*(*int*)
p5: *seq*(*int*) × *seq*(*int*)
p6: *seq*(*int*) × *int* × *int*

p7: *seq*(*int*) × *int* × *int*
p8: *seq*(*int*) × *a2* × *seq*(*int*)
p9: *seq*(*int*) × *a2* × *seq*(*int*)

Modes

empty_seq: u
tail: g, u
head: g ,u
max_seq: g, u
p0: g
p1: g, u
p2: g, u, u
p3: undefined, undefined, undefined, undefined
p4: g, u
p5: g, g
p6: g, u, u
p7: g, g, u
p8: g, u, u
p9: undefined, undefined, undefined

Program clauses

max_seq(x1, x3) ← *p0(x1)* ∧ *p1(x1, x3)*
max_seq(x1, x3) ← ¬ *p0(x1)* ∧ *p2(x1, x4, x5)* ∧ *max_seq(x5, x6)* ∧ *p3(x1, x4, x6, x3)*
p0(x1) ← *p4(x1, x2)* ∧ *p5(x1, x2)*
p1(x1, x3) ← *p6(x1, x7, x3)* ∧ *p7(x1, x7, x3)*
p2(x1, x4, x5) ← *p8(x1, x4, x5)* ∧ *p9(x1, x4, x5)*
p4(x1, x2) ← *tail(x1, x2)*
p5(x1, x2) ← *empty_seq(x2)*
p6(x1, x7, x3) ← *head(x1, x7)*
p7(x1, x7, x3) ← *eq(x7, x3)*

This partial program shows the modes of the predicates which have been derived at this stage of development. The DT operations, i.e. *empty_seq*/*1*, *tail*/*2* and *head*/*2*, have user-defined declared modes while the modes of the other predicates have been derived by the mode inference procedure during its last call which occurred after the last refinement. That is, after matching the arguments of equality *eq/2* with the ones of *p7*/*3*. Note that the predicates *p3*/*4* and *p9*/*3* have undefined mode. These predicates are not reachable by the mode inference procedure. The inferred modes of the DT operations are as follows.

Mode(empty_seq) = g
Mode(head) = g, u

Mode(tail) = g, u

Note that, for each DT its inferred mode is subsumed by its declared one. We are at the stage of matching the arguments of DT operations with the ones of the predicates that they define. Predicate *p8*/*3* has to be refined by the DT operation *head*/*2*. The arguments of predicate *p8*(*x1*, *x5*, *x6*) have to be matched with the ones of the DT operation *head*(*y1*, *y2*). At this stage of development the system displays the types, types are omitted from this discussion in order to simplify presentation, and the modes of the predicates *p8*/*3* and *head*/2. That is,

Type p8: seq(int) × a2 × seq(int)
Type head: seq(a3) × a3
Mode p8: g, u, u
Mode head: g, u
p8(x1, x5, x6) ← head(y1, y2)

If we match the argument pairs (*x5*, *y2*) and (*x6*, *y1*) the clause *p8*(*x1*, *x5*, *x6*) ← *head*(*x6*, *x5*) will be added in the partial logic program. The system will report mode error for the DT operation *head/2*. The mode analysis has inferred mode (*u*, *u*) for this DT operation. The declared mode of *head/2* does not subsume the inferred one. The inferred mode is not valid because the development method requires the arguments of the DT operations with declared mode g to be ground. Note that the matching pairs of arguments have type compatibility but not mode compatibility. The correct matching of argument pairs is (*x5*, *y2*) and (*x1*, *y1*). In this case, the mode analysis will infer for predicate *head*/*2* mode (*g*, *u*) which is subsumed by the declared one. The matching of these pairs of arguments has type and mode compatibility. Therefore, this matching is valid. The clause *p8*(*x1*, *x5*, *x6*) ← *head*(*x1*, *x5*) is added in the partial logic program. Table 2 illustrates this matching of arguments.

Table 2. Argument matching in clause *p8*(*x1*, *x5*, *x6*) ← *head*(*y1*, *y2*)

Pred	**head**	**head**	**p8**	**p8**	**p8**
Arg	*y1*	*y2*	*x1*	*x5*	*x6*
Type	*seq(a3)*	*a3*	*seq(int)*	*a3*	*seq(int)*
Mode	*g*	*u*	*g*	*u*	*u*

If we refine the remaining undefined predicates we will get the following complete program for *max_seq/2*.

Signatures

tail: seq(a1) × seq(a1)
head: seq(a1) × a1
empty_seq: seq(a_1)

ge_int: int × int
le_int: int × int
max_seq: seq(int) × int
p0: seq(int)
p1: seq(int) × int
p2: seq(int) × int × seq(int)
p3: seq(int) × int × int × int
p4: seq(int) × seq(int)
p5: seq(int) × seq(int)
p6: seq(int) × int × int
p7: seq(int) × int × int
p8: seq(int) × int × seq(int)
p9: seq(int) × int × seq(int)
p10: seq(int) × int × int × int
p11: seq(int) × int × int × int
p12: seq(int) × int × int × int
p13: seq(int) × int × int × int
p14: seq(int) × int × int × int
p15: seq(int) × int × int × int

Modes

tail: g, u
head: g, u
empty_seq: u
ge_int: g, g
le_int: g, g
max_seq: g, u
p0: g
p1: g, u
p2: g, u, u
p3: g, g, g, u
p4: g, u
p5: g, g
p6: g, u, u
p7: g, g, u
p8: g, u, u
p9: g, g, u
p10: g, g, g, u
p11: g, g, g, u
p12: g, g, g, u
p13: g, g, g, u
p14: g, g, g, u

p15: g, g, g, u

Program clauses

max_seq(x1, x3) ← p0(x1) ∧ p1(x1, x3)
max_seq(x1, x3) ← ¬ p0(x1) ∧ p2(x1, x4, x5) ∧ max_seq(x5, x6) ∧ p3(x1, x4, x6, x3)
p0(x1) ← p4(x1, x2) ∧ p5(x1, x2)
p1(x1, x3) ← p6(x1, x7, x3) ∧ p7(x1, x7, x3)
p2(x1, x4, x5) ← p8(x1, x4, x5) ∧ p9(x1, x4, x5)
p3(x1, x4, x6, x3) ← p10(x1, x4, x6, x3)
p3(x1, x4, x6, x3) ← p11(x1, x4, x6, x3)
p4(x1, x2) ← tail(x1, x2)
p5(x1, x2) ← empty_seq(x2)
p6(x1, x7, x3) ← head(x1, x7)
p7(x1, x7, x3) ← eq(x7, x3)
p8(x1, x4, x5) ← head(x1, x4)
p9(x1, x4, x5) ← tail(x1, x5)
p10(x1, x4, x6, x3) ← p12(x1, x4, x6, x3) ∧ p13(x1, x4, x6, x3)
p11(x1, x4, x6, x3) ← p14(x1, x4, x6, x3) ∧ p15(x1, x4, x6, x3)
p12(x1, x4, x6, x3) ← ge_int(x4, x6)
p13(x1, x4, x6, x3) ← eq(x3, x4)
p14(x1, x4, x6, x3) ← le_int(x4, x6)
p15(x1, x4, x6, x3) ← eq(x3, x6)

The definitions of the DT operations and equality predicates are as follows.

empty_seq([])
head([h|t], h)
tail([h|t], t)
ge_int(x, y) ← x ≥ y
le_int(x, y) ← x ≤ y
eq(x, x)

The mode analysis of the complete program with respect to goal G_α = *max_seq*(*g*, *u*) has generated the modes in the above program. The inferred mode of each DT operation is subsumed by its declared one. These inferred modes are as follows.

Mode(empty_seq) = g
Mode(head) = g, u
Mode(tail) = g, u
Mode(ge_int) = g, g
Mode(le_int) = g, g

7. Implementation & Evaluation

The algorithms described in this paper have been implemented using SICStus Prolog. Reasoning based on the types and modes of each undefined predicate and the ones of the DT operation which refines it has to be performed by the programmer in order to match the appropriate arguments. Then, the programmer interactively specifies the matching pairs of arguments. For example, the pair (*2*, *1*) stands for matching the second argument from the undefined predicate, head of the new clause, with the first one from the DT operations which refines it. An example run from the construction of program for predicate *sum/2* is shown in Fig. 6.

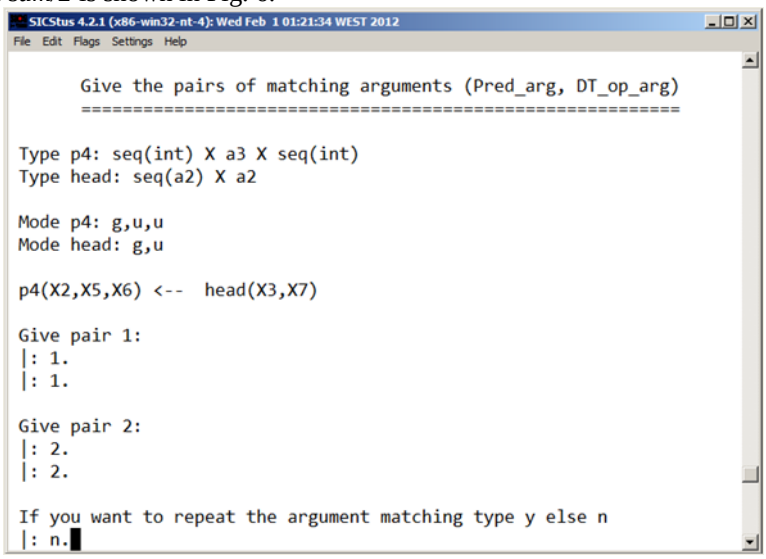

Fig. 6. Snapshot from the construction of program for predicate *sum/2*.

At this stage of program construction, the programmer has to match the arguments of DT operation *head/2* with the ones of predicate *p4/4*. The appropriate decision is taken by reasoning on the types and modes of these predicates. The first argument of *head/2*, i.e. "*x3*", has type "*seq*(*a2*)" and call mode "*g*". Its type matches with the types of first and third arguments of *p4/3*, i.e. "*x2*" and "*x6*", but its call mode "*g*" subsumes only the derived mode of the first argument of *p4/3*, i.e. "*x2*", because "*x2*" has derived mode "*g*". Therefore, "*x3*" has to be matched with "*x2*". The matching pair of arguments is (*1*, *1*). The second argument of *head/2*, i.e. "*x7*", has type "*a2*" and call mode "*u*". Its type matches only with the type of the second argument of *p4/3*, i.e. "*x5*", but its call mode "u" subsumes the derived modes of all arguments of *p4/3*. Therefore, "*x7*" has to be matched with "*x5*". The matching pair of arguments is (*2*, *2*). Note that "*a2*" and "*a3*" are type variables and they match with any type.

Suppose that we are at the stage of matching the arguments of predicate *p5/3* with the ones of DT operation *tail/2*, Fig. 7. We match the 3rd argument of *p5/3*, i.e. *x6*, with the 1st argument of tail, i.e. *x3*. They have type compatibility but not mode compatibility. *x3* has call mode g which does not subsume the mode *u* of *x6*. We also match the 1st

argument of *p5/3*, i.e. *x2*, with the 2nd argument of tail, i.e. *x7*. These arguments have both type and mode compatibility. The mode u of *x3* subsumes the mode g of *x2*. The system detects mode incompatibility and requires repeating of the argument matching process. The correct matching process is shown in figure, Fig. 8. The 1st and 3rd argument of *p5/3*, i.e. *x2* and *x6*, are matched with the 1st and 2nd arguments of *tail/2*, i.e. *x3* and *x7*, respectively. These pairs have both type and mode compatibility.

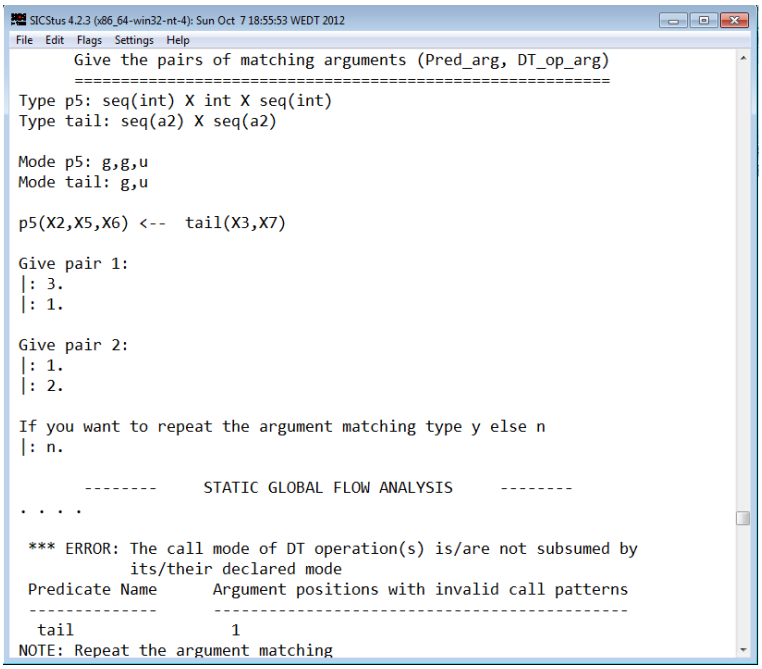

Fig. 7. Incorrect argument matching of predicate $p_5/3$ with the ones of *head*/2.

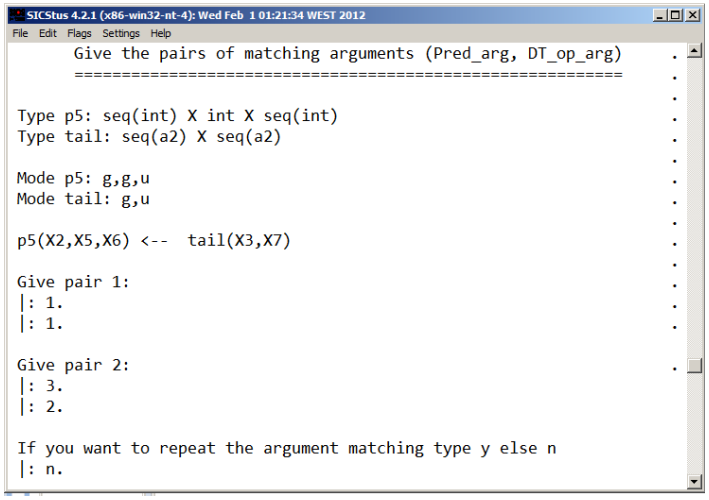

Fig. 8. Correct argument matching of predicate $p_5/3$ with the ones of head/2.

We have found the performance of the system very good. This is due to the fact that our mode analysis algorithm in each iteration of the static analysis evaluates only the calls and answers which have been changed during that iteration, not the whole table of calls and answers.

In order to demonstrate the requirements of our solution is space and time we present in Fig. 9 the statistics of static analysis after matching the arguments of predicate *p4/3* with the ones of DT operation *head/2*. In the figure we print various statistics. That is, the run time of global flow analysis, and the usage of memory by the four stacks, i.e. global, local, trail, and control. In the four stacks, the first value is the size of the stack and the second one is the space usage by the stack. The figure show that the total execution time is only 0.390 sec and the used memory is 4823.544 Kbytes

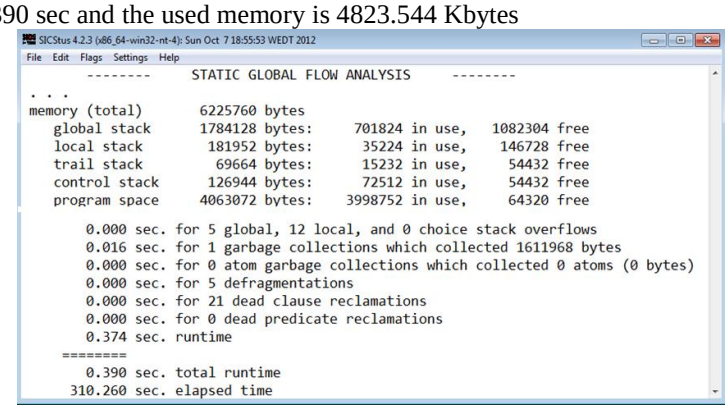

Fig.9. Statistics for memory and time requirements of the static analysis.

In addition, Table 3 shows the memory and time requirements of the mode analysis algorithm after each DT refinement of *sum/2*. All the runs of the mode analysis procedure for this example have similar space and time requirements with maximum execution time 0,452 sec which demonstrates the feasibility and the performance of our solution.

Table 3. Memory and space requirement during static analysis of sum/2.

Refined Clause	**Global Stack**	**Local stack**	**Trail stack**	**Control stack**	**Program space**	**Runtime**
p0(X2) ← empty_seq(X3)	1235464 bytes	35080 bytes	20872 bytes	71848 bytes	3997856 bytes	0.374 sec
p1(X2,X4) ← neutral_add_subtr_int(X3)	1753488 bytes	35080 bytes	35944 bytes	71944 bytes	3998224 bytes	0.374 sec.
p4(X2,X5,X6) ← head(X3,X7)	712088 bytes	35080 bytes	21048 bytes	72040 bytes	3998736 bytes	0.421 sec
p5(X2,X5,X6) ← tail(X3,X7)	1304128 bytes	35080 bytes	17784 bytes	72136 bytes	3999056 bytes	0.436 sec
p3(X2,X5,X8,X4) ← plus_int(X3,X7,X9)	371720 bytes	35080 bytes	12240 bytes	12240 bytes	3999488 bytes	0.452 sec

Comparative evaluation with other similar systems has been left for future research. This is due to the fact that such a comparison in order to be valid should be based on same set of programs. However, it is hard to construct such a set because similar systems have to support static analysis to partially constructed programs and in addition the logic programs should have types and modes.

8. Discussion, Conclusions and Further Research

Refinement of undefined predicates is performed in left-right order, same as the evaluation order. Suppose that the literal l_i from clause $h \leftarrow l_1, \ldots, l_i, \ldots, l_n$ is refined before the refinement of the literals $l_1, \ldots, l_{i-1}$. The actual mode of l_i will be undefined because l_i is not reachable due to the undefined predicates of literals $l_1, \ldots, l_{i-1}$. Using a refinement order which is same as the assumed computation rule will prevent such problems. In principle the definition of predicates does not assume any order. However, in order for the analysis to be practical the left-right order is required.

The mode inference procedure of this paper takes a partially constructed program and the expected call mode of the top-level predicate and computes a call and success mode for each program predicate. The call modes derived by the procedure are used to check that the DT operations are used consistently with their declared call modes. That is, the arguments of each DT operation have to be unified with the ones of the predicate it refines. Each unified argument of an unrefined predicate must have call mode which is subsumed by the mode of the corresponding argument of the DT operation. The novel features of this mode analysis from any previous work we are aware of are the following. This mode analysis is used during program development. That is, it guides the application of DT refinements to undefined predicates. In addition, this mode analysis is performed on partial logic programs.

A direction for further research is static analysis of the constructed programs in other more complex abstract domains which have abstract terms like {*ground*, *free*, *partially ground*}. The programmer can be given the possibility to select an abstract domain from a set of available ones for the construction of his program. Another research direction is the study of our program development method and consequently of the static analysis in more powerful computation rules such as dynamic selection rules defined by delay declarations [9] and dynamic selection of calls based on the data flow through their variables [31]. This static analysis algorithm could be used for type inference as well. In this case, the static analysis algorithm has to be adjusted properly and new abstract domain and abstract operations have to be defined as well. Finally, another interesting research direction is the implementation of our approach in the Answer Set Programming paradigm [32].

9. References

1. E. Marakakis, H. Kondylakis, N. Papadakis, *A Knowledge-Based Interactive Verifier for Logic Programs*, International Journal of Knowledge-based and Intelligent Engineering Systems, 2014, 18, pp.143-156, IOS Press.
2. B. Delbos, *Development of an Interface for a Logic Program Construction Methodology*, Diploma thesis, TEI of Crete and IUP Génie Informatique – Université de La Rochelle, 2004.
3. N. Jones, A. Mycroft, *Dataflow Analysis of Applicative Programs Using Minimal Function Graphs*, in: Proceedings of 13th Annual Symposium on Principles of Programming Languages, (ACM Press, New York, 1986), 296-306.
4. W. Winsborough, *Multiple Specialization Using Minimal-Function Graph Semantics*, The Journal of Logic Programming 13 (1992) 259-290.
5. J. Gallagher, M. Bruynooghe, *The Derivation of an Algorithm for Program Specialization*, New Generation Computing 9 (1991), pp. 305-333.
6. E. Marakakis, *Logic Program Development Based on Typed, Moded Schemata and Data Types*, PhD thesis, University of Bristol, February 1997.
7. E. Marakakis, J. Gallagher, *Schema-Based Top-Down Design of Logic Programs Using Abstract Data Types*, in: Fribourg, L., and Turini, F. ed., Proceedings of Fourth International Workshops on Logic Program Synthesis and Transformation - Meta-Programming in Logic, LNCS 883, June 1994, Pisa, Italy, (Springer-Verlag, Berlin, 1994), pp. 138-153.
8. E. Marakakis, *Use of Static Analysis During Program Development*, Proceedings of the Eighth IASTED International Conference on Artificial Intelligence and Soft Computing, ASC 2004, edited by A.P. del Pobil, September 1-3, 2004, Marbella, Spain, pp. 66-71, ACTA Press.
9. P. Hill, J. Lloyd, *The Gödel Programming Language*, The MIT Press, Cambridge Massachusetts, 1994.
10. P. Cousot, *Abstract Interpretation*, Web Site: http://www.di.ens.fr/~cousot/AI/.
11. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, R. Xavier, *Static Analysis by Abstract Interpretation of Embedded Critical Software*, SIGSOFT Softw. Eng. Notes (2011), 36(1), pp. 1-8.
12. L. Naish, *Negation and Control in Prolog*, Lecture Notes in Computer Science, No. 238, Springer-Verlag, Berlin, 1986.
13. J.G. Smaus, P.M Hill, A. King, *Mode Analysis Domains for Typed Logic Programs*. LOPSTR 1999, pp. 82-101.
14. M.V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J.F. Morales, G. Puebla, *An Overview of the Ciao Multiparadigm Language and Program Development Environment and Its Design Philosophy*, Concurrency, Graphs and Models (Springer-Verlag 2008), pp. 209-237.
15. M. Bruynooghe, *A Practical Framework for the Abstract Interpretation of Logic Programs*, The Journal of Logic Programming 10 (1991), pp. 91-124.
16. H. Mannila, E. Ukkonen, *Flow Analysis of Prolog Programs*, in: Proceedings of 1987 Symposium on Logic Programming, Sept. 1987, San Francisco, California, (IEEE Computer Society Press, 1987), pp. 205-214.
17. S. Debray, D. Warren, *Automatic Mode Inference for Logic Programs*, The Journal of Logic Programming 5 (1988), pp. 207-229.
18. K. Marriott, H. Søndergaard, *Bottom-Up Abstract Interpretation of Logic Programs*. in: Kowalski, R., Bowen., K. ed., Proceedings of 5th International Conference and Symposium on Logic Programming, Volume 1, Seattle, WA, US, 1998, pp. 733-748.
19. C. Mellish, *Abstract Interpretation of Prolog Programs*, in: Shapiro, E. ed., 3rd ICLP (Springer, 14-18 Jul 1986), London, GB, LNCS 225, pp. 463-474.
20. H. Søndergaard, *An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction*. In: Robinet, B., Wilhelm, R., ed., Proc. ESOP '86, (Springer, 1986), Saarbrücken, pp. 327-338.

21. K. Muthukumar, M. Hermenegildo, *Determination of Variable Dependence Information through Abstract Interpretation*, in: Lusk, E., Overbeek, R. ed., NACLP 1989, Volume 1, (MIT Press, 1989), 166-185.
22. R. Barbuti, R. Giacobazzi, G. Levi, *A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs*. TOPLAS, Vol. 15(86); 1, Jan. 1993, pp. 133-181.
23. A. Cortesi, G. Filé, *Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis*, in: Hudak, P., Jones, N. ed., Proc. PEPM '91. (ACM Press, Sep. 1991), Yale U., New Haven, CT, US, ACM SIGPLAN Not. 26(9), 52-61.
24. C. Mellish, *Some Global Optimizations for a Prolog Compiler*, The Journal of Logic Programming 1 (1985), pp. 43-66.
25. J. Gallagher, *A System for Specializing Logic Programs*, Technical Report TR-91-32, Department of Computer Science 1991, University of Bristol.
26. D. Sahlin, *An Automatic Partial Evaluator for Full Prolog*, PhD Thesis, March 1991, The Royal Institute of Technology (KTH), Department of Telecommunication and Computer Systems, Stockholm.
27. P. Hill. R. Topor, *A Semantics for Typed Logic Programs*, in: Pfenning, F., ed., Types in Logic Programming, (The MIT Press, Cambridge Massachusetts, 1992), pp. 1-62.
28. P. M. Hill, J. Gallagher, *Meta-programming in Logic Programming*, Handbook of Logic in Artificial Intelligence and Logic Programming 1988, vol. 5, Logic Programming edited by D. Gabbay, C. Hogger, J. Robinson, pp.421-497, Clarendon Press, Oxford.
29. J. Komorowski, *Sythesis of Programs in the Framework of Partial Deduction*, Ådo Academi, Department of Computer Science and Maths, TR Ser.A, No. 81, July 1989.
30. P. Cousot, R. Cousot, *Abstract Interpretation and Application to Logic Programs*, The Journal of Logic Programming 13 (1992), pp. 103-179.
31. L. Naish, *Automating Control for Logic Programs*, The Journal of Logic Programming 3 (1985), pp. 167-183.
32. G. Brewka, T. Eiter and M. Truszczynski, *Answer Set Programming at a Glance*, Communications of the ACM (2011), 54(12), pp. 92-103.

10. Appendix - Background.

10.1. *The Design Schema Language and Schema Instantiation*

10.1.1. *The Design Schema Language*

The schemata are formally defined as terms in a meta-language for (expressing) polymorphic many-sorted first-order logic. The symbols of the meta-language have the following meanings. The lower case letters u and v, possibly subscripted, are *schema argument variables*. Schema argument variables range over object-language terms. Identifiers beginning with a capital letter are *predicate variables*. They range over object language predicate symbols. The meta-language logical symbols $\wedge$, $\neg$ and $\leftarrow$, when they occur within schemata, stand for the same object level symbols with their usual meaning. The lower case Greek letters α and β, possibly subscripted and/or superscripted are *parameter variables* and they stand for parameters. The lower case Greek letters τ and ρ possibly subscripted and/or superscripted are *type variables* and they stand for arbitrary types.

Definition 20. An *atom schema* is a formula of the form $P\ (u_1, \ldots, u_n)$ where P is a predicate variable of arity n and $u_1, \ldots, u_n$ are distinct schema argument variables.

Definition 21. A *literal schema* has the form A or $\neg A$, where A is an atom schema.

Definition 22. A *clause schema* has the form $A \leftarrow L_1 \wedge \ldots \wedge L_n$, where A is an atom schema and L_i $(1 \leq i \leq n)$ are literal schemata.

Definition 23. Let P be a predicate variable of arity n. A type schema for P, denoted by *Type(P)*, is of the form $Type(P) = \alpha_1 \times \ldots \times \alpha_n$ where α_i $(1 \leq i \leq n)$ are parameter variables.

Definition 24. Let $P(u_1, \ldots, u_n)$ be an atom schema and let $\alpha_1 \times \ldots \times \alpha_n$ be a type schema for P. Then the schema type of u_i is α_i $(1 \leq i \leq n)$.

Definition 25. A *typed clause schema* is a clause schema C together with a type schema for each predicate variable occurring in C, such that each schema argument variable in C has the same schema type wherever it occurs.

Definition 26. A *typed procedure schema* is a set of typed clause schemata with the same predicate variable in the head of each clause.

Definition 27. A *typed program schema* is a set of typed procedure schemata, with one distinguished predicate variable called the top predicate variable. The top predicate variable appears in the head of one procedure schema which is called the top procedure schema.

It is important to make a note about the scope of variables in schemata. An argument variable has as scope the clause schema in which it occurs. However, predicate variables and parameter variables have as scope the program schema in which they occur. This implies that the argument variables in different schema clauses can be renamed but predicate and parameter variables cannot.

Example 1: An example of a schema is as follows.

Type Schemata

Type(P): $\alpha_1 \times \alpha_2$
Type(Q): α_1
Type(R): α_2
Type(S): $\alpha_1 \times \alpha_2$

Clause Schemata

$$P(u_1, u_2) \leftarrow Q(u_1) \wedge R(u_2)$$
$$P(u_1, u_2) \leftarrow S(u_1, u_2)$$

Note that u_1 and u_2 can be renamed to v_1 and v_2 respectively in the schema clause $P(u_1, u_2) \leftarrow S(u_1, u_2)$ to have instead the schema clause $P(v_1, v_2) \leftarrow S(v_1, v_2)$. The predicate variables P, Q, R, S and the parameter variables α_1, α_2 cannot be renamed.

10.1.2. *Schema Instantiation*

Definition 28. Let Σ be a typed program schema. A *predicate substitution* for Σ is defined to be $\{P_1/p_1, \ldots, P_m/p_m\}$ where each p_i $(1 \leq i \leq m)$ is a predicate symbol and $P_1, \ldots, P_m$ are all the predicate variables which occur in the clauses of schema Σ.

If a predicate variable P occurs in two clause schemata in Σ then it is the same P, and a predicate substitution for P assigns the same predicate symbol in each occurrence of P.

Definition 29. Let Σ be a typed program schema with clause schemata $C_1, \ldots, C_r$ and let $C \in \{C_1, \ldots, C_r\}$. An *argument substitution* for clause schema C is defined to be $\{u_1/t_1, \ldots, u_d/t_d\}$ where $u_1, \ldots, u_d$ are the schema argument variables which occur in C and t_i $(1 \leq i \leq d)$ are object language terms. Given argument substitutions $\theta_1, \ldots, \theta_r$ for $C_1, \ldots, C_r$ respectively an argument substitution for schema Σ is defined to be $\theta_1 \cup \ldots \cup \theta_r$.

Definition 30. Let $\alpha_1, \ldots, \alpha_n$ be the parameter variables that occur in the type schemata for the predicate variables of schema Σ. Given types $\tau_1, \ldots, \tau_n$, $\{\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n\}$ is defined to be a *type substitution* for Σ.

If a parameter variable α occurs in two different predicate variables then it is the same α, and a substitution for α assigns the same type in each occurrence of α.

Definition 31. Let Σ be a typed program schema, and let Θ_P and Θ_T be a predicate substitution and a type substitution respectively for Σ. Let P be a predicate symbol in Σ and let $Type(P) = \alpha_1 \times \ldots \times \alpha_n$ be the type schema of P. Let p be a predicate and $P/p \in \Theta_P$. The type for p, denoted by $Type(p)$, is defined to be $Type(P)\Theta_T = \alpha_1\Theta_T \times \ldots \times \alpha_n\Theta_T$.

Definition 32. Let Σ be a typed program schema. Let Θ_P, Θ_A and Θ_T be predicate, argument and type substitutions of schema Σ respectively. A schema substitution Θ consists of the three components Θ_P, Θ_A and Θ_T. An instance of schema Σ is defined to be $\Sigma\Theta_P\Theta_A\Theta_T$.

The definitions of polymorphic many-sorted term, atom and formula are assumed from [9]. The components Θ_P, Θ_A and Θ_T are constructed by this method in such a way as

to produce polymorphic many-sorted formulas. That is, the instances of schemata are polymorphic many- sorted formulas.

10.2. *Partially Ordered Structures, Least Fixed Points and Abstract Interpretation.*

In this section, initially we present some basic definitions on partially ordered structures. Next, we present some theoretical definitions about least fixed points and abstract interpretation. Next, we will proceed to the presentation of abstract operations and the mode analysis algorithm. Then, we will informally define the notion of minimal function graph. After that, we will illustrate the discussion with an example. Finally, it is shown that SI-programs are well-moded.

10.2.1. *Partially Ordered Structures*

Some fundamental definitions for ordered structures are given in this subsection.

Definition 33. A *partial ordering* $\sqsubseteq_D$ on a set D is a binary relation that is reflexive, antisymmetric and transitive. A set with a partial ordering $(D, \sqsubseteq_D)$ is called a *partially ordered set* or *poset*.

Definition 34. Let $(D, \sqsubseteq_D)$ be a poset and $Y \sqsubseteq D$. An element $x \in D$ is called an *upper bound* for Y ***iff*** $\forall$ y $\in Y$, $y \sqsubseteq_D x$. An element $z \in D$ is called a *least upper bound (lub)* for Y, denoted by $\sqcup_D Y$, ***iff*** z is an upper bound and for every upper bound x of Y, $z \sqsubseteq_D x$. An element $x \in D$ is called a *lower bound* for Y ***iff*** $\forall$ y $\in$ Y $x \sqsubseteq_D y$. An element z $\in D$ is called a *greatest lower bound (glb)* for Y, denoted by $\sqsubseteq_D Y$, ***iff*** z is a lower bound and for every lower bound x of Y, $x \sqsubseteq_D z$.

Definition 35. Let $(D, \sqsubseteq_D)$ be a poset and $Y \sqsubseteq D$.Y is called a *chain* in D if Y is nonempty and $\forall\, y_1, y_2 \in$ Y either $y_1 \sqsubseteq_D y_2$ or $y_2 \sqsubseteq_D y_1$.

Definition 36. A poset $(D, \sqsubseteq_D)$ is a *complete partially ordered set (cpo)* ***iff*** every chain Y in D has a least upper bound ($\sqcup_D Y$) in D. A poset $(D, \sqsubseteq_D)$ is a *pointed cpo* ***iff*** it is a *complete partially ordered set* and it has a least element.

The requirement for domains to be either *cpo's* or *pointed cpo's* are sufficient for static analysis based on least fix point semantics.

10.2.2. *Least Fixed Points and Abstract Interpretation*

We describe functions by their graphs. The graph representation of a function is a set of pairs from input - output values.

Definition 37. Let $f: S \rightarrow R$ be a function. $graph(f)=\{(x,f(x)): x \in S\}$. That is, the *graph* of a function f is the set of all *ordered pairs* $(x, f(x))$.

The partial ordering on functions is defined with respect to their graphs.

Definition 38. Let f, g: $D_1 \rightarrow D_2$ and $(D_1, \sqsubseteq_{D1})$ and $(D_2, \sqsubseteq_{D2})$ be two *cpo*'s. The partial order on functions f and g is defined as follows: $f \sqsubseteq g$ ***iff*** $\forall d \in D_1$. $f(d) \sqsubseteq_{D2} g(d)$. That is, f's graph is a subset of g's graph.

Definition 39. Let $(D_1, \sqsubseteq_{D1})$ and $(D_2, \sqsubseteq_{D2})$ be two *cpo*'s. A function f: $D_1 \rightarrow D_2$ is *monotonic* ***iff*** $\forall x, y \in D_1$, $x \sqsubseteq_{D1} y$ implies $f(x) \sqsubseteq_{D2} f(y)$.

Informally, f maps the order of D_1 into the order of D_2. This requirement guarantees that the sequence of elements $\perp$, $F(\perp)$, $F(F(\perp))$,..., $F^i(\perp)$ generated by a monotonic functional F is a chain. *Functional* is a function F: $D \rightarrow D$ where D is a set of functions.

Definition 40. Let $(D_1, \sqsubseteq_{D1})$ and $(D_2, \sqsubseteq_{D2})$ be two *cpo*'s. A function f: $D_1 \rightarrow D_2$ is *continuous* ***if*** it is monotone and for every chain Y in D_1 holds

$$f(\bigcup_{D1} Y) = f(\bigsqcup_{D1} Y) = \bigsqcup_{D2} \{ f(y): y \in Y \}$$

Informally, f maps chains in D_1 into chains in D_2 and their least upper bounds from D_1 into D_2. f $(\sqcup_{D1} Y)$ contains exactly the same information as that obtained by mapping all y's to $f(y)$'s and joining the results.

Definition 41. Let $F: D \rightarrow D$ be a functional and d is an element such that $d \in D$. d is a fixed point of F ***iff*** $F(d)=d$. d is the *least fixed point (lfp)* of F ***if***, $\forall e \in D$, $F(e) = e$ ***implies*** $d \sqsubseteq e$.

If a domain D is a pointed *cpo*, then the least fixed point of a continuous functional F: $D \rightarrow D$ exists and is defined to be the $fixF = \bigsqcup \{F_i(\perp): i \geq 0\}$ where $F^i = F(F(\ldots F(\perp)\ldots))$.

Let $(D^c, \sqsubseteq_{D}c)$ be a *cpo* and D^c stands for the *concrete domain*. *Prog* stands for the set of all logic programs. Let *F(Pr)* be a continuous function where $Pr \in Prog$. F is the concrete semantic function which defines the concrete denotation of logic programs.

$$F^c: Prog \rightarrow (D^c \rightarrow D^c)$$

The *concrete semantics* of a program $Pr \in Prog$ is defined to be the $lfp(F^c(Pr))$. F^c is defined in terms of some other functions. Some of them are *domain-independent* and the others are *domain-dependent*. The latter are referred to here as *operations*. An abstract interpretation is defined by:

1. A cpo $(D^\alpha, \sqsubseteq_{D\alpha})$ where D^α stands for the abstract domain. The abstract domain is a description of the concrete domain.

2. An abstract semantic function F^{α} which defined the abstract denotation of logic programs. $F^{\alpha}(Pr)$ is continuous with respect to the ordering $\sqsubseteq_{D\alpha}$.
   $F^{\alpha}: Prog \rightarrow (D^{\alpha} \rightarrow D^{\alpha})$
   The *abstract semantics* of *Pr* is defined to be $lfp(F^{\alpha}(Pr))$.
3. The abstraction function α which is defined to be $\alpha: D^{c} \rightarrow D^{\alpha}$. The concretization function γ which is defined to be $\gamma: D^{\alpha} \rightarrow D^{c}$. In addition,
   a. Functions α and γ are *monotonic*.
   b. Functions α and γ are *adjoint*. That is,
      - $\forall d \in D^{c}: d^{c} \sqsubseteq_{Dc} \gamma(\alpha(d^{c}))$
      - $\forall d^{\alpha} \in D^{\alpha}: d^{\alpha} = \alpha(\gamma(d^{\alpha}))$

   Note that the function α is usually defined to be of type $\alpha: 2^{Dc} \rightarrow D^{\alpha}$ where 2^{Dc} stands for the power set of D^{c}. That is, the abstraction function α maps subsets of elements of 2^{Dc} into elements of D^{α}. Similarly, the function γ is defined to be $\gamma: D^{\alpha} \rightarrow 2^{Dc}$.
4. Each concrete operation $f^{c}: D^{c} \times \ldots \times D^{c} \rightarrow D^{c}$ must be related to its corresponding abstract operation $f^{\alpha}: D^{\alpha} \times \ldots \times D^{\alpha} \rightarrow D^{\alpha}$ as follows: $\forall \bar{y}\ D^{c} \times \ldots \times D^{c} \rightarrow D^{c}$, $f(\bar{y}) \sqsubseteq_{Dc} \gamma(f^{\alpha}(\alpha(\bar{y})))$.

Definition 42. Given the two semantic functions F^{c} and F^{α} and the monotonic concretization function γ, F^{α} is a safe approximation of F^{c} ***iff*** $\forall Pr \in Prog\ \ lfp(F^{c}(Pr)) \sqsubseteq_{Dc} \gamma(lfp(F^{\alpha}(Pr)))$.

If each abstract operation f^{α} safely approximates the corresponding concrete f then F^{α} is a safe approximation of F.

10.3. *Basic Notions for Minimal Function Graphs*

Definition 43. A *partial function* from S to R is a function $f: S' \rightarrow R$, where S' is a subset of S. If $S' = S$, then f is called a *total function.*

The idea of minimal function graph will be illustrated informally by the example of factorial. Consider the next definition of factorial, $fact: \aleph \rightarrow \aleph$.

$$fact(x) = \begin{cases} 1 & \text{if } x = 0 \\ fact(x\text{-}1) \cdot n & \text{if } x > 0 \end{cases}$$

The graph of this function is defined by the set $S = \{(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), \ldots\}$. Suppose that a particular value for x is given for the evaluation of factorial, e.g. $x=3$. Some subset of function graph is only used for the evaluation of $fact(3)$, i.e. $S_3 = \{(0, 1), (1, 1), (2, 2), (3, 6)\}$. The set S_3 is the subset of S which is activated or called for the evaluation of $fact(3)$. The graph S_3 is the *minimal function graph* for $fact(3)$. For any value of $x=k$, $fact(k)$ is the restriction of function $fact(x)$ to some subset S_k, , i.e. $S_k \subseteq S$.