Logic Program Development Based on Typed, Moded Schemata and Data Types

Emmanouil I. Marakakis

A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

February 1997
Abstract

The main contribution of the thesis is to develop a comprehensive framework for developing logic programs using five program schemata and some basic data types, and to evaluate its suitability for developing sizable programs. The schemata are intended to capture design decisions made by a programmer during top-down refinement of a predicate. The schemata are completely independent from the data types, and have associated types and modes. The method of construction is to start from an initial typed moded call, and proceed using precisely defined refinement operations until primitive data type operations are introduced. It is shown that the construction method produces programs that are polymorphic many-sorted formulas, and that computations using the programs cannot result in type or mode errors. The framework is compared with previous schema-based program development methods from the literature, and we argue that this method is better suited to the development of non-trivial programs, and is more amenable to computerised support. The introduction of types and modes into schemata is a significant addition, not previously reported. An approach to program verification is also introduced, in which the proof of correctness of a program is built from proof schemata corresponding to the program schemata used in the design of the program. Finally an implementation of the system is described and its use is illustrated on some sizable programs. The implementation incorporates a static analysis algorithm for checking the validity of the modes during the development of the program.
Dedication

To my mother Frosini and in memory of my father Yannis.
To my wife Alina and to my son Yannis.
Acknowledgements

I would like to express my sincere thanks to my advisor Dr. John Gallagher. His advice and encouragement throughout this thesis were invaluable.
Declaration

The work in this thesis is the independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of education.

E. I. Marakakis
# Contents

List of Figures 

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>xiii</th>
</tr>
</thead>
</table>

## 1 Introduction 

1.1 Paradigms in Logic Programming 

1.2 Motivation for a New Method for Developing Logic Programs 

1.2.1 Requirements for a Software Development Method 

1.2.2 Requirements for a Schema-Based Approach to Logic Program Development 

1.3 Survey of Schema-Based Methods in Logic Programming 

1.3.1 Program Skeletons 

1.3.2 Most Specific Generalization 

1.3.3 Construction Strategies 

1.3.4 Higher-Order Approaches 

1.3.5 Clichés 

1.3.6 Schemata Based on Inductive Data Types 

1.4 Discussion of Related Work wrt to the Requirements for a Schema-Based Approach 

<table>
<thead>
<tr>
<th>1 Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2</td>
<td>11</td>
</tr>
<tr>
<td>1.3.3</td>
<td>13</td>
</tr>
<tr>
<td>1.3.4</td>
<td>16</td>
</tr>
<tr>
<td>1.3.5</td>
<td>19</td>
</tr>
<tr>
<td>1.3.6</td>
<td>21</td>
</tr>
<tr>
<td>1.4</td>
<td>23</td>
</tr>
</tbody>
</table>
3.5.1 Refinement by a Schema

3.5.1.1 Tupling a Typed cga

3.5.1.2 Schema Refinement of a Typed Atom

3.5.1.3 Flattening

3.5.1.4 Examples

3.5.2 Refinement by Data Type Operations or Equality

3.5.2.1 Derivation of Basic Clauses

3.5.2.2 Type Substitution in Basic Clauses

3.5.2.3 Examples of Basic Clause Construction

3.6 Mode Derivation and Validation

3.6.1 Derivation of Expected Modes in Flattened Clauses

3.6.2 Mode Validation in Basic Clauses

3.7 The Construction Process

3.7.1 Refinement Trees

3.7.2 Signatures and Modes of the Predicates of an SI-Program

3.7.3 An Example

3.8 Properties of SI-Programs

3.8.1 SI-Programs are Polymorphic Many-Sorted Programs

3.8.2 SI-Programs Require No Run-Time Type Check

3.9 Data Types

3.9.1 Built-in Data Types

3.9.2 User-defined Data Types
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9.3</td>
<td>Data Types Supporting Meta-programming</td>
<td>84</td>
</tr>
<tr>
<td>3.10</td>
<td>Examples</td>
<td>86</td>
</tr>
<tr>
<td>4</td>
<td>Mode Analysis</td>
<td>93</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>93</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Abstract Interpretation</td>
<td>94</td>
</tr>
<tr>
<td>4.2</td>
<td>Abstract Analysis Framework</td>
<td>96</td>
</tr>
<tr>
<td>4.3</td>
<td>Mode Analysis Algorithm</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>Examples</td>
<td>100</td>
</tr>
<tr>
<td>4.5</td>
<td>SI-Programs are Well-Modeled</td>
<td>105</td>
</tr>
<tr>
<td>4.6</td>
<td>Discussion</td>
<td>105</td>
</tr>
<tr>
<td>5</td>
<td>Correctness of SI-Programs</td>
<td>107</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>107</td>
</tr>
<tr>
<td>5.2</td>
<td>Logic Specifications</td>
<td>109</td>
</tr>
<tr>
<td>5.3</td>
<td>Correctness Method</td>
<td>110</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Examples</td>
<td>111</td>
</tr>
<tr>
<td>5.4</td>
<td>General Scheme to Prove Correctness</td>
<td>113</td>
</tr>
<tr>
<td>5.5</td>
<td>Schemata-Based Proof Schemes</td>
<td>114</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Proof Scheme for Subgoal Schema</td>
<td>114</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Proof Scheme for Case Schema</td>
<td>114</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Proof Scheme for Incremental and Divide-and-conquer Schemata</td>
<td>115</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Proof Scheme for Search Schema</td>
<td>116</td>
</tr>
</tbody>
</table>
5.6 Application of the Proof Schemes .................................................. 120
  5.6.1 Structured Form of Specifications ............................................. 120
  5.6.2 Equivalence Preserving Transformations ...................................... 121
5.7 Examples of Correctness Proofs .................................................... 123

6 System Implementation ........................................................................ 128
  6.1 Introduction .................................................................................. 128
  6.2 System Components ....................................................................... 129
  6.3 Representation of Design Knowledge ............................................. 131
    6.3.1 Representation of Program Schemata ....................................... 131
    6.3.2 Representation of Data Types .................................................. 133
    6.3.3 Representation of Equality Predicate ....................................... 134
  6.4 Representation and Interpretation of Refinement Trees ..................... 134
    6.4.1 Representation of Nodes of Refinement Trees ............................... 135
    6.4.2 Interpretation of Nodes of Refinement Trees ................................ 136
    6.4.3 Examples ............................................................................... 139
  6.5 Representation of SI-Programs ...................................................... 143
  6.6 Partial Evaluation ......................................................................... 145
    6.6.1 Introduction ........................................................................... 145
    6.6.2 Partial Evaluation of Constructed Programs ................................. 146

7 Conclusions ......................................................................................... 152
  7.1 Comparison of this Program Development Method with Related Work .... 152
7.1.1 Comparison of Schemata ................................................. 153
7.1.2 Comparison of Methods ................................................ 155
7.1.3 Comparison of Correctness Methods ................................. 158
7.1.4 Reasoning about Programs in the Meta-Level ...................... 158
7.1.5 Conclusions from the Comparison .................................... 160
7.2 Limitations of this Method ............................................... 161
7.3 The Practical Use of the Method ........................................ 161
7.4 Contributions .............................................................. 163
   7.4.1 Method of Programming ............................................ 163
   7.4.2 Schema-based Program Development ............................ 164
   7.4.3 Correctness of Logic Programs .................................. 164
7.5 Further Research ......................................................... 165
Bibliography ...................................................................... 167

A Partial SI-Programs and Implementations of Data Type Operations 175
A.1 Partial SI-Program for Insertion Sort ................................. 175
A.2 Implementation of a Data Type Operation for a User-defined Data Type 177

B Non-trivial Programs .......................................................... 183
B.1 Depth-First Search Spanning Tree ..................................... 183
   B.1.1 Refinement tree ...................................................... 184
   B.1.2 SI-Program Without Definitions of DT Operations .......... 187
B.2 Unification ................................................................. 197
B.2.1 Unification Algorithm ................................................. 197
B.2.2 Refinement tree ............................................................ 199
B.2.3 SI-Program Without Definitions of DT Operations ......... 203

C Examples of Correctness Proofs ........................................... 225
C.1 Sequence Length Even ..................................................... 225
C.2 Ordered Sequence ........................................................... 231

D BNF Syntax of the Refinement Meta-Language .................. 247

E Sample Sessions of SIDS ................................................... 249
E.1 Summation of Elements of a Sequence ......................... 250
E.2 Insertion Sort. ................................................................. 256

F Refinement Trees ............................................................... 266
List of Figures

3.1  A sample of a refinement tree. .............................................. 71
3.2  Refinement tree for predicate insSort/2. .............................. 90
5.1  Search space for constructing the Fibonacci number for \( n = 5 \). .... 119
6.1  The system components and schematic view of the construction process. . . . . 130
6.2  The linguistic levels. ............................................................. 131
6.3  Refinement tree for predicate fac/2. ..................................... 138
6.4  Refinement tree for partial program of the predicate insSort/2. ........ 140
6.5  Refinement tree for predicate incrOrd/1. ................................ 141
B.1  The top part of the refinement tree for unify/3. ...................... 204
B.2  Parts 1 and 2 of the refinement tree for unify/3. .................... 205
B.3  Parts 3 and 4 of the refinement tree for unify/3. .................... 206
B.4  Part 5 of the refinement tree for unify/3. ........................... 207
Chapter 1

Introduction

From very early on, software development methods aimed to support the construction of software with the properties of correctness, maintainability and efficiency. They aimed to achieve these properties with minimum effort and cost and without any restriction in the size of the constructed programs. Correctness ensures that software meets its specification. Software maintenance involves the modification of software product after delivery in order to adapt it to a changed environment or to improve other properties. Efficiency is a metric that is used to measure the extent to which computing resources are used by software. That is, software is efficient if it performs its intended functions with a minimum use of computing resources.

A number of different software development methods or paradigms have been evolved in order to satisfy these goals. Some of the major paradigms which have been followed are waterfall model, prototyping, formal transformations, executable specifications, deductive synthesis and the schema-based paradigm [2], [62], [75], [78], [82], [84].

The waterfall model divides the software development process into the following sequence of development stages, requirements specification, design, implementation, testing and maintenance. After the definition of each stage, development proceeds to the one following it [2], [78], [84]. The disadvantages of waterfall model are the high development cost and the unsatisfactory reliability, performance, and functionality of the resulting software. Correctness is mainly shown by testing.

The prototyping paradigm involves the development of a partial implementation of the system in order to demonstrate the most important aspects of the system. The prototype
is then used by the intended users which supply feedback to the developers. Its objective is to establish the system requirements. This stage is followed either by re-implementation of the software or by incremental addition of functionality to the initial system until its full development [2], [78], [84]. Systems designed through prototyping have better user interfaces. This approach helps the development process when the user’s requirements for the system are not well understood.

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 [75], [82]. Formal transformations have been applied in restricted application domains and at the level of individual programs. It is not yet an alternative for the development of large software systems. 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 executable specifications paradigm involves the development of an executable specification of the system. Its focus is on the functions that the system should perform. The resulting executable specification is a kind of prototype which produces all of the functional behavior of the system. Its aims are to clarify the user requirements and to perform some early validation. The resulting specification is ultimately mapped into implementation by changing the mechanisms which produce the behaviour into structures of the target environment [24], [98]. Verification is possible if executable specifications are mapped into implementations by transformations which have been proved correct. Real applications are not simple so large specifications may become complex and unmaintainable. Transformational implementation of specifications do not allow human intervention because specifications become less comprehensible after a few transformations.

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 [59]. 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.

In the schema-based paradigm the user makes the design decisions during development and the system carries them out. This approach has been developed in order to deal with the search problem inherent in deductive synthesis and transformational approaches. Methods based on some kind of abstraction such as program schemata follow this model [79]. In this method large-scale transformations are involved e.g. schemata, leading to short derivations
of programs. If the schemata correspond to structures natural to programmers then the
development process is similar to what programmers do.

1.1 Paradigms in Logic Programming

From the aforementioned major paradigms, the following four have been mainly used in
logic programming.

1. Formal transformations: That is, formal specifications in first-order logic are trans­
formed into logic programs by correctness-preserving transformations [16], [33], [38],
[54].

2. Executable specifications: First-order logic itself is appropriate for expressing spec­
cifications. Such specifications need only control to become executable [51]. Logic
languages which have been extended with some syntactic notation in order to become
more readable are used for writing specifications [46], [47]. Such specifications are
executable thus allowing validation of the user requirements.

3. Prototyping: Logic languages are appropriate for rapid prototyping [50], [65].

4. Schema-Based: The program development in such systems is based on programming
abstractions like program schemata, skeletons, logic program forms, clichés [5], [20],
[21], [25], [26], [30], [31], [40], [52], [53], [73], [74], [80], [86], [90], [97].

1.2 Motivation for a New Method for Developing Logic
Programs

1.2.1 Requirements for a Software Development Method

Each software development method supports some of the aforementioned software prop­
erties. It shows strength in some aspects of the development process and weakness in others.
Here we discuss the requirements and difficulties of each software property. In addition, we
discuss which properties each method supports.

1. Correctness: It requires a formal specification and formal justification of the devel­
opment of code. This is explicitly supported by the methods of formal transforma­
tions, executable specifications, deductive synthesis, but not by waterfall model and schema-based paradigms. The correctness proofs of large and complex software systems becomes difficult. One solution to this problem is to allow specifications to be decomposed into smaller units [82].

2. **Efficiency**: It requires control over low-level operations. This property is taken into consideration in waterfall model. The other approaches rely on refinements, or efficient primitives operations. Its main difficulty is to relate a high-level specification to a low-level computational model.

3. **Maintainability**: It requires clearly structured code. Abstraction and modularization are essential features to enhance maintainability. It is supported explicitly by schema-based paradigm more than the other ones. Generally, it is encouraged by the more formal approaches.

The above discussion suggests the following requirements for a development method for software intended to support the construction of large-scale, effort-effective and cost-effective software with the properties of correctness, maintainability and efficiency.

1. Full automation has not shown strength in developing large software. Semi-automation with integrated tools to support various stages during development seems to be more appropriate for constructing large software.

2. It is sometimes difficult to understand efficient code. It is better to develop an understandable system and then proceed to its optimization. The unoptimized version of the system can be used for verification and maintenance and the optimized one will be used for running.

3. Transformation rules which correspond to design decisions that programmers follow reduce the search problem. Such transformations lead to short and more understandable derivations. The development process corresponds to design and programming activities that programmers follow. Such features reduce cost and effort of constructing software.

4. Abstraction facilitates reusability which in turn results in time and cost gains.

5. **Formal specifications facilitate formal correctness proofs.**

6. **High-level abstraction and modularization mechanisms enhance structure and maintainability.**

These requirements suggest an approach combining the best features of formal transformations and schema-based paradigms.
1.2.2 Requirements for a Schema-Based Approach to Logic Program Development

Schemata capture commonly occurring patterns of programs. Algorithms can be expressed as schemata by abstracting problem representation details. Each schema represents a set of programs. A program can be derived from a schema by the instantiation operation. Using a library of schemata programming becomes selection of a particular schema and instantiation of its abstract components. A schema is constructed once but reused in several programs. In addition, composing instances of schemata non-trivial programs can be constructed. Logic programs are amenable to schema-based development methods due to the simplicity of their syntax. In addition, logic programmers tend to use implicitly certain program patterns. The requirements which should be satisfied by a schema-based method for constructing logic programs are as follows.

1. **Well-engineered schemata.** The design of schemata is very important to a schema-based method. The schemata should be abstract enough in order to be used for the construction of large classes of programs. A desirable feature is independence from particular data representations. The schemata should have enough useful information to support the construction process. Over-generalization should be avoided.

2. **Small set of schemata.** A logic programmer would like to have a small set of schemata which are manageable and computationally powerful.

3. **Schema selection should be based on program design criteria.** A logic programmer should be able to decide easily on the selection of a schema. Schemata may have to be applied to problem domains which are new to logic programmers. Schemata which represent program design decisions that programmers follow in constructing logic programs are desirable. Their selection is based on program design criteria.

4. **Schemata which capture the expected mode and type of predicates.** Logic programmers implicitly use some arguments to represent the input data and some other arguments to represent the constructed solution, i.e. the output data. In addition, logic programs make implicit assumptions about types. For example, the well-known append/3 predicate may succeed when its second and third arguments are not lists unless a type test is added into the unit clause. Untyped logic programs need type information in order to be correct with respect to specifications [71].

5. **Simple schema instantiation operation with instances without uninstantiated components.** The schema instantiation operations should be as simple as possible to use. We believe that such an operation should involve just two objects, i.e. the predicate
that we wish to define and a schema. In addition, the instantiation operation should produce schema instances which are formulas in the target language. It is desirable the interaction with the user to be as little as possible.

6. **Combination of instances of schemata.** Program construction by combining instances of schemata makes a schema-based method more powerful and the construction process an engineering activity. A set of schemata whose instances can be composed can be used in several different ways constructing programs which otherwise would require additional schemata to construct such programs. The construction of logic programs becomes an engineering activity where selection and application of the appropriate sequence of schemata results in the target program.

7. **Construction of non-trivial logic programs.** A schema-based method should be powerful enough in order to support the construction of non-trivial logic programs. The phrase ‘non-trivial programs’ means the following: 1) The method supports the construction of normal logic programs. 2) It should be shown that the method supports the construction of logic programs which are defined in terms of more than one logic procedure. 3) There is adequate evidence that the method is extensible to complex problems which consist of several logic procedures. 4) The method supports the complete construction of the program. That is, it does not assume that some logic procedures which are called by the program being constructed have already been constructed by other applications of other methods.

8. **Construction of maintainable logic programs.** A schema-based method should provide features which allow the maintenance of non-trivial logic programs. The separation of domain knowledge from the programming knowledge results in programs with clear structure which are maintained easily.

9. **Correctness.** The construction method should be complemented by an approach to proving correctness with respect to specifications. It is advantageous specifications and programs to be expressed in the same language.

Efficiency is an issue that does not have to be taken into account during the construction phase. Once a correct program has been constructed then it can be transformed in an equivalent efficient one. Runnable logic programs should be efficient.
1.3 Survey of Schema-Based Methods in Logic Programming

This section contains a survey of logic program development methods based on abstraction of control. Names given to these methods in the literature include schemata, skeletons, cliches and logic program forms. The following categories are distinguished, classified according to their approach to defining schemata.

1. **Skeletons**: These schemata are defined as (first-order) logic programs which consist of the most fundamental components.

2. **Construction strategies**: These schemata are defined to represent a strategy for constructing logic programs.

3. **Most Specific Generalizations**: These schemata are defined as the most specific generalizations of a set of programs.

4. **Higher-order approaches**: These schemata are defined as higher-order expressions. The literals of the schemata have predicate variables as arguments.

5. **Clichés**: These schemata are defined as commonly occurring program forms where a program form is a particular arrangement of the parts of a program.

6. **Schemata Based on Inductive Data Types**: These schemata are defined in terms of the inductive definition of an underlying data type.

1.3.1 Program Skeletons

The construction of logic programs from other basic logic programs by adding computation around them in a stepwise manner is discussed in [52], [53], [80], [86]. This method is based on the idea that a logic program may be conceived of as an enhancement of a simpler program, i.e. a skeleton.

**Stepwise Enhancement**: Schemata are basic programs with a well-understood control flow. Schemata are classified in [52] into data structure traversers, syntax analyzers, meta-interpreters, clichés of control flow and algorithms. These classes are not mutually exclusive. Enhancements are a restricted class of modifications commonly applied to logic programs. Enhancements are classified into modulation, extension and mutation [52]. Creating a new
procedure from existing code or unfolding a procedure call is called modulation. Extension introduces extra arguments and goals around the control flow of an existing program without altering its flow of control. Changes which alter the control flow of the original program, such as introduction of new clauses and recursive goals, are called mutation.

Programming techniques are standard programming practices. They are classified as enhancements whose changes do not affect the problem domain [52]. Programming techniques can be applied into different types of program skeletons. For example, the propagate context up technique is used to propagate information computed at deeper levels in an execution tree outside.

The first step of this method is to identify a skeleton for the problem. This skeleton provides the control of the target program. Computation is added into skeleton by applying successively enhancements on it. Programs which are extensions of the same skeleton can be composed into a single program thus combining the functionalites of the individual extensions [52].

For example, the skeleton traverse_list performs the complete traversal of a list. The predicate length(L, N) is true if N is the number of elements of list L. sum(L, S) is true if S is the sum of elements of list L. length/2 and sum/2 are extensions of the traverse_list skeleton obtained by applying the propagate context up technique. length/2 and sum/2 can be combined into program length_sum/3 because they are extensions of the same skeleton.

Skeleton traverse_list:

\[
\text{traverse_list}([X|Xs]) \leftarrow \text{traverse_list}(Xs) \\
\text{traverse_list}([])
\]

Program constructed by extension enhancement:

\[
\text{length}([], 0) \\
\text{length}([X|Xs], L) \leftarrow \text{length}(Xs, L1), L \text{ is } L1 + 1
\]

Program constructed by extension enhancement:

\[
\text{sum}([], 0) \\
\text{sum}([X|Xs], S) \leftarrow \text{sum}(Xs, S1), S \text{ is } S1 + X
\]

Program constructed by composition:

\[
\text{length_sum}([], 0, 0) \\
\text{length_sum}([X|Xs], L, S) \leftarrow \text{length_sum}(Xs, L1, S1) \\
L \text{ is } L1 + 1, S \text{ is } S1 + X
\]

An implementation method for incorporating programming techniques into programs based on meta-interpreters is also presented [52], [53].
This approach supports the construction of non-trivial Prolog programs and meta-programs including programs with negation. This is illustrated by the development of a tracer for Prolog [52].

**Applying Skeletons to Techniques in MIS:** The work in [86] is based on ideas of its predecessor work in [52]. That is, a class of enhancements called extensions is discussed. In particular, programs are constructed by applying techniques to skeletons which do not alter the control flow of skeletons. The composition of extensions of the same skeleton is considered as well. Their study of constructing programs by applying only techniques to skeletons is wider than in [52] but in the same spirit. Their list of techniques is larger than the ones in [52]. In addition, the application of skeletons and techniques method within the framework of Model Inference System (MIS) [83] is discussed. This is an extension of the work in [52]. MIS is a system which synthesizes Prolog programs from examples of their behaviour [83]. A refinement operator for MIS based on skeletons and techniques is proposed in [86].

Initially, the system is given a positive example for the target Prolog program. It creates a number of possible skeletons. Meta-knowledge added into the system is used in order to reduce the number of generated skeletons. This meta-knowledge includes the following.
1) Mode and type of variables is used to determine how the variables are instantiated. 2) Information about ‘allowable’ predicates guides clause creation. For example, the predicates < and \( \geq \) cannot be used in the same clause if they have the same arguments in the same order. 3) Every recursive skeletal clause has the recursive predicate as the last goal. Next, these skeletons are extended with all possible enhancements by the enhancement module. Finally, redundant clauses are removed by applying negative examples to the generated clauses, producing the final program.

For example, the following program for \( \text{prefix} /2 \) is generated by MIS using the skeletons and techniques refinement operator. It also shows that this extended MIS system cannot remove redundant clauses due to the underlying structure of MIS. \( \text{prefix}(L1, L2) \) is true if list \( L1 \) is a prefix of list \( L2 \).

\[
\begin{align*}
\text{prefix}(X, X) \\
\text{prefix}([], []) \\
\text{prefix}([], X) \\
\text{prefix}([Y|Xs], [X|Ys]) & \leftarrow \text{prefix}(Xs, Ys)
\end{align*}
\]

Programs which contain only one predicate are learned easily while programs which are defined in terms of other predicates are not. The reason is that the refinement operator gives high priority to expanding and unifying elements on a list in contrast to adding goals
to a clause. The main weakness of the extended MIS system is the large number of clauses which are produced. The logic programs that are constructed by this methodology are trivial. It is difficult for the extended MIS to learn programs with calls to other procedures unless these logic procedures are synthesized before their use. The extended MIS may have infinite loops for such programs.

An Editor for Techniques: A prototype system for constructing simple Prolog programs based on the notions of skeletons and techniques [52], [86] is presented in [80]. This approach is a variant of the ones in [52], [86]. Its motivation is to develop a practical tool intended to teach techniques to novice programmers.

A new component introduced by this system are elaborations. Elaborations are performed to individual clauses after the application of techniques. They involve the instantiation of certain flagged subgoals. Note that techniques are applied all over the clauses of a program.

An initial skeleton is selected. Next, techniques are applied successively to it producing a partial program. Finally, elaborations are applied on individual clauses with flagged subgoals. This approach retains a single partial program which is successively enhanced by techniques and elaborations. It does not support construction of programs by joining extensions of the same skeleton as in [52], [86].

For example, suppose that a user selects the skeleton traverse_list in order to construct a procedure for predicate append/3. The required sequence of techniques and elaborations for the construction of the procedure are as follows.

Partial program obtained by selecting the skeleton traverse_list:

\[
\begin{align*}
\text{append}([A|B]) & :- \text{append}(B) \\
\text{append}([], F)
\end{align*}
\]

Partial program obtained by applying the technique add_carrier:

\[
\begin{align*}
\text{append}([A|B], C) & :- \text{append}(B, C) \\
\text{append}([], F, G)
\end{align*}
\]

Partial program obtained by applying the technique back_accumulate:

\[
\begin{align*}
\text{append}([A|B], C, D) & :- \text{append}(B, C, E), $\text{flagged_subgoal1}(A, B, C, D, E) \\
\text{append}([], F, G) & :- $\text{flagged_subgoal2}([], F, G)
\end{align*}
\]

Partial program obtained by elaborating $\text{flagged_subgoal1}(A, B, C, D, E):

\[
\begin{align*}
\text{append}([A|B], C, D) & :- \text{append}(B, C, E), D = [A|E] \\
\text{append}([], F, G) & :- $\text{flagged_subgoal2}([], F, G)
\end{align*}
\]

Program obtained by elaborating $\text{flagged_subgoal2}([], F, G):

\[
\begin{align*}
\text{append}([A|B], C, D) & :- \text{append}(B, C, E), D = [A|E] \\
\text{append}([], F, G) & :- F = G
\end{align*}
\]
The representation of both techniques and elaborations is based on Definite Clause Grammars. The system provides a library of descriptions of predicates which a user might want to define. Techniques and elaborations are also selected by the user from a list of applicable ones. The current form of the system allows the construction of only trivial logic programs. The programs that are illustrated in [80] are definite programs without calls to other logic procedures.

1.3.2 Most Specific Generalization

The notion of schema in [30] and in [90] is defined as the most specific generalization (msg) (or least general generalization) [77] of a set of programs. This notion of generalization captures syntactic similarities between logic programs.

**Basic Prolog Schemata:** Fourteen schemata called basic level schemata have been produced in [30] for representation of recursive list processing programs. Each schema is assumed to represent a general programming technique. For example, *schema_A* classifies the programs that recursively process all the elements of a list. These schemata have been derived as the msg [77] of a set of programs. A hierarchy which consists of a set of less general schemata and programs is built around each basic level schema. Each hierarchy is intended to distinguish specific cases within each general programming technique. These schemata provide the core knowledge of a Prolog tutoring system.

The aim of this approach is to construct a set of schemata which capture most of the simple list processing programs without producing over-generalized schemata. Its motivation is that the definition of the schemata can be based on an extended msg algorithm and on general programming techniques in order to avoid over-generalization. The sets of programs for generalization have been carefully selected in order to produce schemata which represent the desired general programming techniques. Thus each schema captures the essence of its set of programs.

The fourteen basic level schemata and the specialized schemata in their hierarchies are the only components of this approach. Programs can be derived from a schema by applying a series of substitutions and instantiations on the schema as follows. 1) Substitutions are applied on the first-order and second-order schema variables. 2) Instantiations are applied to optional and arbitrary arguments or goals. That is, an optional argument or goal should be either deleted or included. An arbitrary argument or goal should be included any number of times. Instantiations are also applied to permutable arguments.

**Example:** Optional arguments and goals in *Schema_A* are enclosed in < >. Arbitrary
arguments and goals are enclosed in $\langle \rangle$.

Schema $\text{schema}_A$

\[ \text{schema}_A([], \langle &1 \rangle). \]
\[ \text{schema}_A([H \mid T], \langle &2 \rangle) \leftarrow \langle \text{pre.process}(\langle &3 \rangle, H, \langle &4 \rangle) \rangle, > \]
\[ \text{schema}_A(T, \langle &5 \rangle) \leftarrow \langle \text{post.process}(\langle &6 \rangle, H, \langle &7 \rangle) \rangle. \]

Instance of $\text{schema}_A$.

\[ \text{length}([], 0) \]
\[ \text{length}([X \mid Xs], Len) :- \text{length}(Xs, Len1), Len \text{ is } Len1 + 1 \]

The implementation of an extended msg algorithm is discussed in [30]. That is, the msg algorithm for first-order formulas [77] has been extended in order to produce second-order schemata. This algorithm produces the msg schema $S$ of two formulas $F_1$ and $F_2$ where a formula can be either a logic program or a schema. It also finds the appropriate substitutions $\sigma_1$ and $\sigma_2$ necessary to transform the msg schema $S$ to $F_1$ and $F_2$ respectively. The programs that have been constructed by this approach are simple list processing Prolog programs. Construction of logic programs by applying sequences of schemata is not discussed.

**Induction of Schemata for Program Synthesis:** The construction of logic programs by this method is a search through a large space of schemata and programs [90]. An ordering and an equivalence relation is defined on schemata and programs in order to search their space with some efficiency. In addition, two refinement operators, one for generalization and one for specialization, are defined in order to provide structure to the space.

The motivation for this approach is that efficient algorithms can be designed which narrow the search. The algorithm which finds the msg of a set of programs uses properties of schemata and properties of the refinement operators in order to prevent the search of the entire space. It is also proposed that synthesis start from an initial schema in order to prevent the search of the space generated by the refinement operators.

The components of this approach is a schema and sets of positive and negative examples.

Two algorithms for finding a program are presented. Their search is guided by the specialization refinement operator and by examples. One algorithm is given as input a schema and a set of positive and negative examples. It generates specializations of schema until one is found which covers all the positive examples and none of the negatives. The other algorithm is given as input a hierarchy of schemata and a set of positive and negative examples. It searches the schemata in the hierarchy up to a certain depth for a program which covers all the positive examples and none of the negatives.
For example, the next schema and the sets of positive and negative examples are used for the synthesis of a program for *prefix/2*.

**Schema:**

\[
P(\rightarrow, \cdot)
P([H|T], [H|]) \leftarrow P(T, \cdot)
\]

**Positive examples:**

*prefix([a], [a, b, c]), prefix([], [a, b, c]), prefix([a, b, c], [a, b, c]), prefix([c], [c]), prefix([b], [b, a])*

**Negative examples:**

*prefix([b, c], [a, b, c]), prefix([c], [a, b, c]), prefix([a, b, c], [a, b])*

**Constructed program:**

*prefix([], \cdot)
prefix([H|T1], [H|T2]) \leftarrow prefix(T1, T2)*

Algorithms which derive schemata as the msg of a set of programs are also presented. This method illustrates the synthesis of trivial recursive logic programs. The illustrated logic programs have calls to basic logic procedures like *member/2, and append/3*. These basic logic procedures are assumed to be constructed before their use.

### 1.3.3 Construction Strategies

**Construction Based on Structural Induction and Generalization:** A method for constructing logic programs based on structural induction and on generalization is proposed in [20], [21]. This method first constructs a *logic description (LD)*, i.e. a formula in first-order logic, which is then transformed into an efficient logic program. The construction of a LD requires the choice of an induction parameter, the choice of a well-founded relation, the construction of structural forms of the induction parameter and the construction of structural cases of the problem. If a problem can not be reduced into a smaller one that can be solved by recursive use of the description then its generalization is proposed. Generalization also results in more efficient solutions. Three generalization strategies are proposed, i.e. *tupling, descending and ascending generalization*. Tupling strategy generalizes the structure of a parameter. Descending and ascending strategies characterize a state of computation in terms of 'what has been done' and 'what remains to be done'. This method proposes 4 schemata, i.e. one schema for constructing LDs without generalization and one schema for each generalization strategy.
The motivation for this approach is that the method of constructing LDs with or without generalization can be expressed by schemata. Such schemata can be specialized to partial LDs which can be further refined to get the desired LDs.

The basic technical components of this approach are the 4 schemata, information about types, structural forms of types, well-founded relations and directionalities (i.e. call and success modes).

The constructed LDs are closed well-formed formulas of the following form.

$$\forall X_1,\ldots,X_n(P(X_1,\ldots,X_n) \iff C_1 \land F_1 \lor C_2 \land F_2 \ldots \lor C_n \land F_n)$$

where each $C_i (1 \leq i \leq n)$ is a case of the induction parameter which is often a literal. Each $F_i (1 \leq i \leq n)$ verifies that a relation holds in this particular case. $F_i$ is a conjunction of literals. The construction of a LD is performed in 3 phases. The design phase which produces a partially instantiated LD. The undefined literals have to be defined by the user. The verification phase which tries to detect syntactic error as well as inconsistent, incomplete or undefined concepts. Finally, the transformation phase helps the user to produce a version of the LD which will give a more efficient and simplified logic program.

For example, the construction of the LD for length/2 using descending generalization is illustrated.

**Schema for descending generalization:**

\[ p(X, Y) \iff p\text{-}desc(X, Y, 0\ldots Y) \]
\[ p\text{-}desc(XX, Y, Int\ldots Y) \iff \text{minimal}(XX) \land Y = \text{Int}\ldots Y \]
\[ \lor \neg\text{minimal}(XX) \land \text{decompose}(XX, \text{FirstPart}\ldots XX, \text{Rem}\ldots XX) \land \text{extend}\ldots \text{Int}(\text{FirstPart}\ldots XX, \text{Int}\ldots Y, \text{New}\ldots \text{Int}\ldots Y) \land \]
\[ p\text{-}desc(\text{Rem}\ldots XX, Y, \text{New}\ldots \text{Int}\ldots Y) \]

**Constructed logic description:**

\[ \text{length}(L, Lg) \iff \text{length}\_\text{desc}(L, Lg, 0) \]
\[ \text{length}\_\text{desc}(LL, Lg, \text{Int}\ldots Lg) \iff LL = [\] \land Lg = \text{Int}\ldots Lg \land \text{pos}\ldots \text{int}(Lg) \]
\[ \lor LL = [H T] \land \text{add}(\text{Int}\ldots Lg, t, \text{New}\ldots \text{Int}\ldots Lg) \land \]
\[ \text{length}\_\text{desc}(T, Lg, \text{New}\ldots \text{Int}\ldots Lg) \]

An interactive assistant for incremental construction of correct and efficient LDs has been developed. The schemata form part of the knowledge of the system [21].

LDs with negation are constructed by this method. The examples illustrated by the method
in [20], [21] are trivial LDs. These LDs have been derived by specializing once one schema. Construction of LDs by applying sequences of schemata is not shown.

**Construction Based on Divide-and-Conquer Strategy:** A framework for stepwise synthesis of logic algorithms (LAs) from sets of examples and properties is presented in [25], [26]. A logic algorithm of a procedure \( r \), denoted by \( LA(r) \), consists of a formula of the form \( r(X, Y) \rightarrow \text{Def}[X, Y] \), where \( \text{Def} \) is a first-order logic formula. The set of examples \( E(r) \) expresses the behaviour of procedure \( r \). The set of properties \( P(r) \) consists of first-order logic statements. Initially, the logic algorithm \( LA_1(r) \) is constructed which is a first approximation of the relation \( r \). Next refining \( LA_i(r) \) results in \( LA_2(r) \) and so on. The successive refinements which lead to the series \( LA_1(r), LA_2(r), \ldots, LA_f(r) \) are guided by a Divide-and-Conquer schema. Each \( LA_i \) (2 \( \leq i \leq f \)) is a better approximation of the intended relation \( R \) than \( LA_{i-1} \). The “intended” relation \( R \) is the relation that one has in mind when he/she elaborates a specification consisting of examples and properties. The motivation of this method is to express specifications by examples and properties which have complementary strengths and weaknesses. The synthesis method guided by the Divide-and-Conquer schema and by using this specification aims to the construction of a correct LA with respect to the intended relation \( R \).

The basic technical components of this approach are a set of positive examples and a set of properties. It should be noted that the schema is not input into the synthesis mechanism. It is hardwired into it.

The synthesis process consists of the next sequence of steps. 1) Creation of the first approximation \( LA_1 \) by re-expressing the example set \( E(r) \) as a LA. 2) Synthesis of Minimal, NonMinimal and Decompose subgoals of the Divide-and-Conquer schema. 3) Synthesis of the recursive atoms via deduction and/or analogical reasoning. 4) Synthesis of Solve, Process, Compose and Discriminate subgoals of the Divide-and-Conquer schema. 5) Syntactic generalization.

**Example:** The predicate \( \text{efface}(X, L1, L2) \) is true if list \( L2 \) is list \( L1 \) without the first occurrence of \( X \). A bold term and atom in the Divide-and-Conquer schema denotes a vector of terms and a conjunction of atoms respectively.

**Divide-and-Conquer Schema:**
\[
R(X, Y) \rightarrow \text{Minimal}(X) \land \text{Solve}(X, Y) \\
\lor \lor_{1 \leq k \leq c} \text{NonMinimal}(X) \land \text{Decompose}(X, HX, TX) \land \\
\text{Discriminate}_k(HX, TX, Y) \land R(TX, TY) \land \\
\text{Process}_k(HX, HY) \land \text{Compose}_k(HY, TY, Y)
\]
Set of examples $E(\text{efface})$:
- $\text{efface}(a, [a], [])$
- $\text{efface}(b, [b, c], [c])$
- $\text{efface}(e, [d, e], [d])$
- $\text{efface}(f, [f, g, h], [g, h])$
- $\text{efface}(j, [i, j, k], [i, k])$
- $\text{efface}(p, [m, n, p], [m, n])$

Set of properties $P(\text{efface})$:
- $\text{efface}(X, [X|T], T), \text{efface}(X, [Y, X|T], [Y|T]) \leftarrow X \neq Y$

 Constructed Logic Algorithm:

$\text{efface}(X, L, R) \leftarrow L = [E] \land R = []$

\[
\lor L = [HL|TL] \land E = HL \land R = TL \\
\lor L = [HL|TL] \land E \neq HL \land \text{efface}(E, TL, TR) \land R = [HL|TR]
\]

The synthesized LAs which are illustrated are trivial. There is one example in [26] which illustrates the synthesis of the LA of a procedure with a call to another procedure. This synthesis derives LAs for both procedures. The extension of this kind of synthesis to non-trivial LAs is not straightforward.

### 1.3.4 Higher-Order Approaches

**Logic Program Forms:** A logic program form is a program abstraction where the basic syntax of the program called the logic program form is separated from its low level complex details called interpretation [97]. From a logic program form $F$ a class $C(F)$ of logic programs are derived by defining interpretations. The synthesis of a program requires to supply a program form with the definition of an interpretation. The motivation of this approach is that by separating logic programs into their program form and the low level structural details results in clarity and modularity of their representation. The construction of new programs becomes more flexible by adding new interpretations into a program form.

The components of this approach are logic program forms and interpretations. A logic program form $F$ is a finite set of extended Horn clauses. An extended Horn clause is a Horn clause with extended atomic formulas. An extended atomic formula is an atomic formula with possibly un-instantiated predicate names, i.e. predicate variables. That is, $F = \{C_1, \ldots, C_n\}$ where each $C_i$ $(1 \leq i \leq n)$ is of the form either $P_0$ or $P_0 \leftarrow P_1, \ldots, P_k$ $k \geq 1$ and $P_j$ $(0 \leq i \leq n)$ are extended atomic formulas. An interpretation of a logic program form $F$ is an ordered pair $I = (f, G)$. $f$ is a mapping from a set of predicate variables appearing in $F$ to a set of predicate symbols. $G = \{g_1, \ldots, g_t\}$ where each $g_i$ $(1 \leq i \leq t)$ is a mapping from a set of variables which appear in logic form $F$ to a set of terms with the property for any predicate variable $P g_i(P) = f(P)$. $g_i$ is applicable to a clause $C$ in a given logic form $F$ if $g_i$ is defined for all variables appearing in $C$. The program clauses which are derived from the
extended Horn clause $C : P_0 \leftarrow P_1, \ldots, P_k$ of the logic form $F$ by applying interpretation $I$ are \{\(g_i(C), \ldots, g_i(C)\)\} where $g_i(C) = g_i(P_0) \leftarrow g_i(P_1), \ldots, g_i(P_k) (1 \leq i \leq t)$.

Example:

Logic program form $F_0 = \{C_1, C_2\}$:

\[
\begin{align*}
C_1 & : \text{recursive}(Q, R, X_0) \leftarrow Q(X_0) \\
C_2 & : \text{recursive}(Q, R, X_0) \leftarrow R(Y_0), \text{recursive}(Q, R, Z_0)
\end{align*}
\]

Let the interpretation $I = (f, \{g\})$ be defined as follows: $f(Q) = \text{unif}I$, $f(R) = \text{unif}$, $g(X_0) = (X, Y, Z)$, $g(Y_0) = (\text{car}(X), \text{car}(Z))$ and $g(Z_0) = (\text{cdr}(X), Y, \text{cdr}(Z))$. The functions $\text{car}(X)$ and $\text{cdr}(X)$ denote the head and the tail of list $X$ respectively. The constructed program is $P = \{g(C_1), g(C_2)\}$. The logic program form $F_0$ and the interpretation $I$ define a logic program for $\text{append}/3$. That is, the logic program $P$ works as the standard definition of $\text{append}/3$.

Constructed logic program $P = \{g(C_1), g(C_2)\}$

\[
\begin{align*}
g(C_1) & : \text{recursive}(\text{unif}I, \text{unif}, (X, Y, Z)) \leftarrow \text{unif}(X, Y, Z) \\
g(C_2) & : \text{recursive}(\text{unif}I, \text{unif}, (X, Y, Z)) \leftarrow \text{unif}(\text{car}(X), \text{car}(Z)), \\
& \quad \text{recursive}(\text{unif}I, \text{unif}, (\text{cdr}(X), Y, \text{cdr}(Z)))
\end{align*}
\]

where

\[
\begin{align*}
\text{unif}(X, Y, Z) & \leftarrow \text{unif}(X, []), \text{unif}(Y, Z) \\
\text{unif}(X, X) & \leftarrow
\end{align*}
\]

Definition of $\text{append}/3$ in terms of logic form $F_0$:

\[
\begin{align*}
\text{append}(X, Y, Z) & \leftarrow \text{recursive}(\text{unif}I, \text{unif}, (X, Y, Z))
\end{align*}
\]

It should be noted that there is not any implemented system but only a theoretical study of the problem. Definite logic programs are constructed by the logic forms in [97].

**A Method Based on Quasi-Higher-Order Predicates:** A method for constructing logic programs based on a set of higher-order predicates is proposed in [73], [74]. Some of these higher-order predicates are based on inductive data structures. The others are heuristic schemes of commonly occurring recursive program forms. One of these higher-order predicates, i.e. $\text{solve}$, is universal. The other higher-order predicates can be derived as instances of it. Logic programs are constructed from these quasi-higher-order predicates by using a meta-logic technique introduced in [93]. This technique handles predicate variables thus making redundant the use of higher-order unification. The motivation of this approach is to provide a set of quasi-higher-order predicates which are intended to handle all recursions in logic programs. In addition, this method stays in first-order logic by using the well-known meta-logic technique in [93] for handling predicate variables.
The components of this method are the quasi-higher-order predicates. User-defined predicates which are used as arguments of the quasi-higher-order predicates. The predicate apply which applies the user-defined predicates to their arguments.

A logic program is constructed by this method as follows. A clause is specified whose head is the predicate that the user wants to define and its body has a call to a quasi-higher-order predicate. Arguments of the quasi-higher-order predicate are other user-defined predicates. These user-defined predicates are applied to their arguments by using the predicate apply. Their definitions are also included in the constructed programs. A more efficient program in standard form may be derived by folding and unfolding the constructed logic programs.

For example, the predicate `append/3` is defined in terms of the higher-order predicate `foldr`. Folding and unfolding this definition we get the standard definition of `append/3`.

Higher-Order predicate `foldr`:
\[
\begin{align*}
\text{foldr}(P, Z, [], Z). \\
\end{align*}
\]

Definition of `append/3` using `foldr`:
\[
\begin{align*}
\text{append}(U, V, W) & \leftarrow \text{foldr}(\text{cons}, V, U, W). \\
\text{foldr}(P, Z, [], Z). \\
\end{align*}
\]

A large set of examples such as list processing programs, interpreters for finite automata, parsers and meta-interpreters illustrate the potential of this approach. Negation is not considered by this method.

Representing Program Schemata and Techniques in λProlog: A method for constructing logic programs in λProlog [66] from program schemata and techniques is presented in [31]. This method basically follows the idea of skeletons and techniques for first-order logic programs in [52], [86]. The representation of schemata (skeletons) and techniques has changed to higher-order. The motivation of this approach is that programs, program schemata and techniques can be represented in the same higher-order language. This eliminates the need for the creation of an abstract meta-language for the definition of schemata and techniques. In addition, the construction of programs using program schemata and techniques becomes just the definition of a clause in λProlog.

The components of this approach are program schemata and techniques. A program is
constructed from a program schema by defining a predicate in terms of the schema as a higher-order clause. A program definition contains calls to the program schema defining it. Programming techniques are also defined as higher-order programs. They are used in the same way as the program schemata. Programs defined in terms of techniques include that technique. In addition, techniques can be applied to general schemata to produce more specialized ones.

Example: The λProlog schema `reduceAll/4` captures programs which process all the input list. The predicate `append/3` is defined in terms of this schema.

Program schema in λProlog:

```
reduceAll [] Base Constructor Base.
reduceAll [A | B] Result Constructor Base :-
    reduceAll B C Constructor Base,
    Constructor A C Result.
```

Definition of `append/3` using the schema `reduceAll/4`:

```
append X Y Z :- reduceAll A C (X \ Y \ Z \ (Z = [X | Y])) B
```

The defined λProlog program for `append/3`:

```
append [] L L.
append [H | T] Result :- append T L X, Result = [H | X].
```

The λProlog schema `reduceAllAcc/4` represents the accumulator technique. `length A B` is true if `B` is the length of list `A`. The predicate `length/2` is defined in terms of this technique.

Accumulator technique in λProlog:

```
reduceAllAcc Lst Acc Constructor Result :-
    (List = [], Result = Acc);
    (List = [A|B], Constructor A Acc C,
      reduceAllAcc B C Constructor Result).
```

Definition of `length/2` using the technique `reduceAllAcc/4`:

```
length A B :- reduceAllAcc A 0 (X \ Y \ Z \ (Z is Y + 1)) B
```

No special implementation is required to get a runnable program. A program defined in terms of a schema or a technique is runnable. This is due to the higher-order unification provided by λProlog. Construction of trivial λProlog programs is illustrated in [66].
1.3.5 Clichés

Cliché Programming in Prolog: A system which supports the construction of Prolog programs by instantiating clichés is discussed in [5]. Clichés are defined as program forms which occur often. This approach does not propose a standard set of clichés. It describes a tool which allows programmers to write and use an individual library of clichés. The motivation of this approach is the practical aspects of clichés. It supports the idea of building individual libraries of clichés. In addition, it provides a language for defining clichés and a software tool which allows their use.

Clichés are the only components of this method. A user can define a new cliché in the cliché language. A tool reads new clichés and creates a procedure for each one. The procedure corresponding to each cliché generates instances for that particular cliché. The programmer in order to get an instance of a cliché has to specify in a command the procedure name that will be defined, the cliché name that will be used and the parameters of the cliché. For example, the command \texttt{p/n := c(q/n)} means define a procedure \texttt{p/n} by instantiating cliché \texttt{c} with parameter \texttt{q/n}.

For example, the \texttt{n.ary-apply} cliché expressed in the cliché language is shown below. The parameters of this cliché are \texttt{$S$} and \texttt{$Base$}. Parameters are instantiated when a cliché is used for the construction of a program. \texttt{List, Result, Arg, Acc, Rest} and \texttt{NewAcc} are Prolog variables in this cliché. Prolog variables remain Prolog variables in instances of the cliché. The procedure which generates instances for the \texttt{n.ary-apply} cliché constructs the next Prolog program for predicate \texttt{sum/2}.

\begin{verbatim}
n.ary.apply cliché:
\$A/n := n.ary.apply($S, $Base).
\$A(List, Result) :- \$A1(List, $Base, Result).
\$A1(List, Result, Rest).
\$A1([Arg | Rest], Acc, Result) :- $S(Arg, Acc, NewAcc), \$A1(Rest, NewAcc, Result).
\end{verbatim}

The constructed Prolog program:

\begin{verbatim}
sum(List, Result) :- sum1(List, 0, Result).
sum1(List, Result, Result).
sum([Arg | Rest], Acc, Result) :- add(Arg, Acc, NewAcc), sum1(Rest, NewAcc, Result).
\end{verbatim}

The classes of programs that can be constructed by this approach depend on each particular library of clichés.
1.3.6  Schemata Based on Inductive Data Types

**Synthesis Based on Schema Instantiation:** A method for formal development of logic programs based on recursive program schemata and on a specification theory is presented in [40]. The motivation of this approach is that formal development of logic programs can be based on program schemata. That is, a schema is the starting point for the synthesis. In addition, each schema clause initiates a derivation for the development of a corresponding program clause.

The basic technical components of this approach are the program schemata, and a specification theory. The specification theory consists of the following: 1) A specification of the program relation. 2) A set of specifications of the auxiliary relations which appear in the definition of the program relation. 3) The theories of the relevant data structures. A such theory consists of the definition of the data structure, its distinctness properties and a specification of its membership relation.

This method derives a logic program as follows. Initially, a representation for the arguments of the program relation has to be chosen. Next, a program schema and the inductive definition of an argument of the program relation has to be chosen. Next, the schema predicate variable is instantiated by the name of the program relation. This partial instance is called *instantiated program schema*. Each clause of the instantiated program schema is the starting point for a derivation. Inference steps are performed on each clause of the partially instantiated schema until the complete derivation of the target program clause. A derivation is performed as a continual equivalence rewriting of formulas. The purpose of a derivation is, given a theory $T$, an atomic formula $C$ and a conjunction of atomic formulas $\Phi$, find a formula $\forall(R \rightarrow U)$, where $R$ is an atomic formula and $U$ is any logical formula, such that,

$$ T \cup \{\forall(R \rightarrow U)\} \vdash \Phi \land R \rightarrow C $$

The desired formula $\forall(R \rightarrow U)$ is constructed from a sequence of equivalences used to successively rewrite the formula R. A derivation can be terminated when the unknown relation R has an instantiation $U$ which make the conclusion to follow from the condition.

**Example:** The synthesis of a logic program for the program relation $\text{subset}(X, Y)$ will be shown. The specification of the program relation $\text{subset}(X, Y)$ is as follows.

$$ \forall X, Y(\text{subset}(X, Y) \rightarrow \forall V(\text{on}(V, X) \rightarrow \text{on}(V, Y))) $$

Initially, it is decided sets to be represented by simple lists. Next, the first argument is
chosen as the induction argument. A program schema which recursively inspects every element of a list is also selected.

Program schema:
\[
F([], Z) \rightarrow R1(Z) \\
F([U|Y], Z) \rightarrow F(Y, Z) \land R2(U, Y, Z)
\]

Instantiated program schema:
\[
subset([], Z) \rightarrow R1(Z) \\
subset([U|Y], Z) \rightarrow subset(Y, Z) \land R2(U, Y, Z)
\]

The specification theory \( T \) for this example consists of the following:

1. The specification of the program relation \( subset(X, Y) \).
2. The theory for simple lists.
   a. The definition of the list data structure.
      \[
      \forall X (\text{list}(X) \leftrightarrow X = [] \lor \exists U, Y (X = [U|Y] \land \text{list}(Y))
      \]
   b. Distinctness properties for list.
      \[
      \forall X, Y (\lnot [X = Y]) \\
      \forall U1, X1, U2, X2 ([U1|X1] = [U2|X2] \rightarrow U1 = U2 \land X1 = X2)
      \]
   c. Specification of membership operation on the data structure list, i.e. \( \text{on(Element, List)} \).
      \[
      \forall U, X (\text{on}(U, X) \leftrightarrow \exists V, Y (X = [V|Y] \land (U = V \lor \text{on}(U, Y))) \\
      \forall U (\text{on}(U, [])) \rightarrow \text{false} \\
      \forall V, Y, U (\text{on}(U, [V|Y]) \rightarrow U = V \land \text{on}(U, Y))
      \]

Derivations for the synthesis for this example:
\[
T \cup \{ \forall Z (R1(Z) \leftrightarrow U1) \} \vdash \forall Z (\text{subset}([], Z) \leftarrow R1(Z)) \\
T \cup \{ \forall U, Y, Z (R2(U, Y, Z) \leftrightarrow U2) \} \vdash \\
\forall U, Y, Z (\text{subset}([U|Y], Z) \leftarrow \text{subset}(Y, Z) \land R2(U, Y, Z))
\]

Derived Program:
\[
\text{subset}([], Z) \\
\text{subset}([U|Y], Z) \leftarrow \text{subset}(Y, Z) \land \text{on}(U, Z)
\]

A prototype system has been developed based on this approach which generates successive partial derivation plans for the next transformations up to the comparison between conclusion part and condition part.
This approach supports the synthesis of normal logic programs. The largest example that is illustrated in [40] is sorting. Synthesis of non-trivial programs does not seem to be straightforward because the transformation steps are small and they result in long sequences of derivation steps.

1.4 Discussion of Related Work wrt to the Requirements for a Schema-Based Approach

1. Well-engineered schemata. The schemata in [5], [30], [40], [52], [53], [80], [86], [90] depend on the underlying data structures. In addition, within the same data type different inductive cases are distinguished which result in different schemata, e.g. processing all or some of the elements of a list. Such schemata are not flexible enough to be applied to the construction of large classes of programs. Over-generalization with very little useful information to support the construction process may occur in the schemata in [90] unless other criteria are taken into consideration such as programming techniques as in [30]. Each schema clause in [40] consists of an inductive argument and its recursive subgoals including their arguments. These schemata do not have adequate information to support derivation of instances which do not have undefined components.

The approaches in [20], [21], [25], [26], [31], [73], [74], [97] propose more abstract sets of schemata. These schemata are independent of the particular data types except the tupling schema in [20], [21] and some of the schemata in [31], [73], [74] but these are derived from other more general schemata which are independent of particular data types. The schemata in [20], [21], [25], [26] have sufficient information to support the construction process. The schemata in [31], [97] are higher-order programs and the ones in [73], [74] are quasi-higher-order programs. These schemata are part of the constructed programs.

2. Small set of schemata. The methods in [20], [21], [25], [26], [30], [31], [73], [74], [97] propose small and fixed sets of schemata which are manageable by humans, while the others approaches in the survey do not. The schemata in [30] support the computation of a limited class of programs while the ones in [20], [21], [25], [26], [31], [73], [74], [97] are more flexible and computationally more powerful.

3. Schema selection should be based on program design criteria. The selection of schemata in [5], [30], [40], [52], [53], [80], [86] is based on the experience of the programmer with a particular problem domain. The programmer is expected to have in his mind an abstraction of the program solving the problem. This is not straightforward for new and complex application domains. The schemata in [20], [21], and to some extent
in [73], [74] represent program construction strategies so their selection fits into the human program design process. Selection is not needed in [25], [26] because one schema is only used.

4. **Schemata which capture the expected mode and type of arguments.** None of the approaches we are aware of that are included in our survey supports schemata which capture knowledge about the expected mode and type of the arguments of their literals. However, the methods in [20], [21], [86] use modes. Mode declarations are supplied as meta-knowledge to the system in [86] in order to determine which variables will be unified. Logic descriptions are constructed in [20], [21] which satisfy certain mode requirements expressed by call and success patterns.

5. **Simple schema instantiation operation with instances without uninstantiated components.** The instantiation operation in [5], [30] requires the user to specify all the uninstantiated components so the generated instances do not have undefined parts. The schema instantiation in [86], [90] is performed by the system by specializing the schema. The schema instances do not have uninstantiated components but may have as in [86] redundant components, i.e. clauses. There is no need for instantiation operation in [31], [97] due to higher-order unification and in [73], [74] due to the well known technique presented in [93]. The schema instantiation operation in [20], [21] is performed automatically by the system. The schema instances contain undefined literals which have to be defined by the user. There is no schema instantiation operation in [25], [26]. The Divide-and-Conquer schema guides the successive refinements of logic algorithms. There is a simple partial instantiation operation in [40] which involves the recursive schema predicate variables. The important part of the program is derived by formal transformations. The instantiation in [80] requires interaction with the user in order to specify the techniques and the elaborations which will be applied. Their application is performed automatically by the system. The instantiation in [52], [53] requires interaction by the user in order to specify the enhancements and the components that they will be applied to. In addition, some abstract predicates generated by the enhancement tools have to be defined by the programmer.

6. **Combination of instances of schemata.** The methods in [5], [20], [21], [30], [80], [90], [97] discuss program construction by applying neither more than one schemata nor the same schema more than once. The composition of programs which are extensions of the same skeletons is discussed in [52], [86]. It is shown in [26] that new sets of examples and properties can be inferred during synthesis. The synthesis method is applied to these sets of examples and properties for the synthesis of a procedure which is called by the initial one. It is discussed in [40] that during the derivation of a program clause it is possible to start the synthesis of a new predicate. If the form $U$ of the relation $R$ in a partially instantiated schema clause does not have the desired
form then a procedure for the relation \( R \) has to be derived. A new schema is selected and the formula \( \forall (R \iff U) \) is taken as the specification of the relation \( R \) for the new synthesis. The combination of different schemata is illustrated in [31], [73], [74]. That is, predicates are defined in terms of other predicates which are in turn defined by different or same schemata, i.e. higher-order or quasi higher-order procedures.

7. **Construction of non-trivial logic programs.** The methods in [5], [20], [21], [25], [26], [30], [31], [40], [80], [86], [97] do not illustrate their capability of constructing non-trivial logic programs as defined in Section 1.2.2. The method in [73], [74] illustrates the construction of complex programs such as parsers, interpreters and meta-interpreters but programs with negation are not illustrated. The method of skeletons and techniques in [52] supports the construction of non-trivial programs. It is illustrated by the construction of a tracer for Prolog.

8. **Construction of maintainable logic programs.** Maintenance can only be discussed with respect to non-trivial programs. The methods we are aware of, presented in our survey, do not discuss such issues, nor do the ones which construct non-trivial logic programs.

9. **Correctness.** Programs in [90] are specified by sets of positive and negative examples. Verification is discussed with respect to these sets of examples. In a similar direction, programs in [25], [26] are specified by sets of examples and properties. Verification is discussed with respect to these sets of examples and properties. Specifications in [20], [40] are expressed as formulas in first-order logic and correctness is ensured by construction. It is shown in [20] that the logic descriptions are correct by construction. The development method in [40] produces programs which are partially correct with respect to a first-order logic specification. It is also shown that there exists a specification called the implemented specification which is stronger that the actual specification. The completed definition of the implemented relation is complete with respect to the implemented specification. The remaining schema-based methods do not discuss correctness with respect to specifications.

This discussion shows that none of the approaches we are aware of and they are included in this survey satisfies all the requirements of a schema-based approach.

### 1.5 The Method of this Thesis

Initially, we discuss the requirements of the schema-based method of this thesis with respect to the requirements in Section 1.2.2. Next, we discuss the general features of this method with respect to the requirements in Section 1.2.1. Finally, we make a short introduction to this method.
1.5.1 Requirements of a New Schema-Based Method

1. **Well-engineered schemata.** This approach consists of schemata independent of particular data types. Their design has been based on semantic criteria.

2. **Small set of schemata.** A set of 5 schemata are proposed which we find very expressive. They are flexible enough to be applied to large classes of problems.

3. **Schema selection should be based on program design criteria.** The schemata of this approach represent program design decisions that programmers follow in developing programs.

4. **Schemata which capture the expected mode and type of predicates.** Each predicate variable of a schema has a mode schema and a type schema. Predicates in instances of schemata have expected mode and type which are derived from the mode and type schemata of the corresponding predicate variable.

5. **Simple schema instantiation operation with instances without uninstantiated components.** The instantiation operation is simple. It requires an undefined predicate, the type and the expected mode of its arguments and a schema which will instantiate it. The instantiation of an undefined predicate by a schema is performed automatically based on the expected modes of the arguments of the predicate. The instances of schemata do not have uninstantiated components. That is, they are formulas in the target language.

6. **Combination of instances of schemata.** This method provides the necessary mechanisms for combining the instances of 5 schemata. These mechanisms are top-down stepwise refinement, expected modes and types. In addition, this construction method provides informal guidance to the refinement process. That is, each literal schema informally defines a subtask that has to be completed by its instances. Refining an undefined predicate by one of the 5 schemata puts forward new subtasks that have to be met by subsequent refinements.

7. **Construction of non-trivial logic programs.** This method supports the construction of normal logic programs. It refines a top-level predicate up to the level of data type (DT) operations which are part of this method and their implementations are provided. This method support the construction of non-trivial logic programs as they have been defined in Section 1.2.2. This is, illustrated by examples in Appendix B.

8. **Construction of maintainable logic programs.** The domain knowledge, i.e. the data types, are separated from the programming knowledge, i.e. the program schemata. The structural details of the domain knowledge are confined into the DT operations. The arguments of the schemata and the ones of their instances are variables. The
consequence of this is that the constructed programs have very clear structure so they are maintained easily.

9. **Correctness.** This method is supported by a correctness method which is based on first-order logic.

The schemata in the method of this thesis as the ones in [20], [21], [25], [26] and partly in [73], [74] are based on program design strategies. This feature makes the schemata independent of the syntax of logic programs. The design of such schemata has been based on semantic criteria. Another strong feature 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.

### 1.5.2 The General Features of this Method

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. The system in its current state is supported by a mode analysis tool.

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 of this thesis 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.

4. The method is built around abstract entities. The program schemata correspond to *procedural or structural abstractions*. Reusability is performed at the design level. The design decisions, i.e. the program schemata, are reused not the code. Other reusable components at the code level are the DT operations.

5. We have developed a correctness proof method that is based on formal specifications rather than testing.

6. Program schemata and DT operations enhance structure and maintainability of the constructed programs.
1.5.3 Introduction of this Schema-Based Method

Program Development by Stepwise Refinements

The main components of this logic program development method are top-down design, program schemata, data types and modes. A program schema is a representation-independent algorithm. A data type consists of a set of values and a set of operations which are applied on these values. The (call) mode of a predicate indicates how its arguments are instantiated when that predicate is called.

The construction process proceeds top-down by successively refining undefined predicates. A refinement consists of either the application of a program schema or the introduction of a DT operation and equality. Refinements by applying schemata introduce new undefined predicates. Refinement by DT operations do not create new undefined predicates. Such refinements terminate the refinement process in the paths of refinement trees.

Initially, the top-level undefined predicate is refined. Next, undefined predicates that are created by application of schemata are refined until no undefined predicate remains. The construction process is semi-automatic. That is, the refinements are carried out automatically by the system. The program designer specifies an undefined predicate and the refinement which will refine it. In addition, in refinements by DT operations he/she specifies the matching arguments of the undefined predicate with the ones of the DT operation or equality which refines it.

Supporting Tools

A mode analysis tool is integrated into this development system. Static mode inference is performed dynamically after the matching of the arguments of each DT operation or equality with the arguments of the predicate it defines. The aim of mode analysis during development is to verify that each DT operation should have inferred mode which is subsumed by its declared one.

Another tool which has been used by this method is a partial evaluator [28], [81]. The inefficiencies in the programs that may be caused due to their very structured form are remedied by partial evaluation. Partial evaluation is used to remove the design layer from the constructed programs thus reducing their size and making them possibly more efficient.
Program Correctness

A correctness proof scheme is proposed for proving the correctness of the constructed programs with respect to formal specifications. Each program schema is associated with a proof scheme. Correctness proofs are guided by the constructed programs. That is, the proof scheme that is followed in a proof is the one which corresponds to the design schema which has been applied for the definition of the predicate of the correctness theorem. The aim is to prove the correctness of the top-level predicate. The overall correctness proof follows a top-down scheme as the top-down structure of the constructed programs. Correctness proofs for predicates in lower levels up to the level of DT operations or user-defined DT operations are used as lemmata for the correctness proofs of the predicates in higher levels up to the top-level predicate. The correctness proof of each predicate with respect to its specification is performed independently from the rest proofs.

1.6 Thesis Outline

This thesis is organized as follows. The main software development paradigms including their advantages and disadvantages over the properties of correctness, efficiency and maintainability are presented in Chapter 1. The paradigms which have been followed in logic programming are discussed as well. Next, the general requirements for a new software development method and the requirements for a new schema-based method are presented. Next, a survey of the schema-based methods is presented. This is followed by a critical discussion of these methods with respect to the requirements for a new schema-based method. Finally, the method of this thesis is discussed with respect to the general requirements for a new software development method and with respect to the requirements for a new schema-based method. This is followed by an introduction into the schema-based method of this thesis.

An introduction into a method for constructing typed, logic programs by stepwise top-down design using schemata and DTs is presented in Chapter 2. An introduction and the definition of the fundamental concepts of DTs, the design schema language and the schema instantiation operation are also presented.

The method of this thesis is presented in details into Chapter 3. Initially, design by stepwise top-down refinement and modes are introduced. Next, the 5 schemata of this method are presented. This is followed by a presentation of the refinement operations. That is, refinement by a schema and refinement by DT operations or equality. Next, the derivation of the expected modes of predicates and the mode validation which is performed during construction of basic clauses are presented. Next, construction of partial and complete
logic programs by this method is presented. This presentation includes refinement trees, and derivation of signatures and modes of the predicates of constructed programs. An example illustrates the detail construction of a program and its associated refinement tree. Next, it is shown that the constructed programs are polymorphic many-sorted formulas and they can be run without run-time type checking. Next, built-in DTs, user-defined DTs and DTs supporting meta-programming are presented. Finally, examples of constructed programs are shown.

The mode analysis procedure is presented in Chapter 4. Initially, an overview of domain theory and abstract interpretation is presented. Next, the abstract analysis framework is presented. This is followed by the mode analysis algorithm and a discussion on the implementation of this algorithm. An example which illustrates the application of mode analysis during program development is also presented. Next, it is shown that programs constructed by this method are well-modeled. Finally, a short discussion of the mode analysis method follows.

In Chapter 5 a correctness method for the logic programs that are constructed by this program construction method is presented. Initially, we discuss the general form of logic specifications supported by this method. Next, the correctness method is presented. Next, a general scheme for proving correctness is presented which is followed by correctness proof schemes for the 5 program schemata. If the top-level predicate has been defined by a schema refinement then the corresponding correctness proof scheme should be followed for proving its correctness with respect to its specification. Next, the application of proof schemes is presented. The transformation of specifications into an intermediate structured form is proposed. Next, the correctness of constructed programs with respect to the structured form of their specifications is presented. Finally, the correctness method is illustrated with an example.

The implementation of the software system which supports the method of this thesis is presented in Chapter 6. Initially, the main components of the system and their functionalities are presented. Next, the representation of design knowledge, i.e. program schemata, DT operations and equality, is discussed. A meta-language has been developed in order to represent refinement trees. Next, the representation and the interpretation of the refinement trees is presented. Each refinement tree is interpreted into a logic program which is represented by a sorted binary tree. This representation of logic programs is presented in the next section. One of the aims of this method is the runnable programs to be efficient. This is done by partially evaluating the constructed programs using Mixtus [81] and SP [28]. The partial evaluation of constructed programs is discussed in Section 6.6. The time and space requirements of a set of programs constructed by this method and the corresponding ones generated by partial evaluation are measured and compared.
Initially, a critical comparison of the method of this thesis with the ones in related work is presented in Chapter 7. It is also argued that program reasoning by this method should be performed at the level of refinement trees which are high-level designs. Next, the limitations of this method and the requirements which make this method practical are discussed. Next, the contributions of this thesis are presented. Finally, the research directions for further expansion of this thesis are discussed.

At the end of this thesis there are five appendices. Appendix A has a partial program for insertion sort constructed by this method. It also includes the implementation of a complex DT operation for a user-defined DT.

Two examples of non-trivial logic programs that have been developed by this method are presented in Appendix B. That is, a program for the construction of a depth-first search spanning tree of a strongly connected directed graph and a program implementing a unification algorithm.

The correctness proofs of two programs developed by this method are presented in Appendix C.

The syntax of the refinement meta-language expressed in BNF rules is presented in Appendix D.

Appendix E has two sample sessions of the program development system of this thesis. One example involves the construction of a program which finds the sum of the elements of a sequence. The second example involves the construction of a program for insertion sort.

Finally, a set of refinement trees is presented in Appendix F. These refinement trees illustrate the argument discussed in Section 7.1.4 that program reasoning by this method should be performed at the level of refinement trees.
Chapter 2

Typed Program Schemata

2.1 Introduction

In this chapter we introduce a method for constructing typed moded logic programs by stepwise top-down design using schemata, Data Types (DTs) and modes [61]. The method is based on the instantiation of program schemata.

A program design schema for a language $L$, or simply schema for $L$, represents a set of programs in $L$ having a common form. Each such program is called an instance of the schema. Schema-based program design consists of putting together schema instances to make complex programs. One approach to program design is top-down design, which is the process of successive refinement of the parts of a design until there are no "undefined" parts.

The data manipulated by the program are an additional component, independent from the schemata. This construction method separates the form of a program from the representation of the data manipulated by the program. The result of the top-down design is a structured logic program. We also call the constructed logic programs by this method executable designs or designs because their structure reflects the design decisions that have been applied during their construction. To summarise the results of our approach, we argue that our method replaces the well-known equation [96]

$$ \textit{Algorithms} + \textit{Data Structures} = \textit{Programs} $$
by the equation

\[ \text{Schemata} + \text{DTs} = \text{Executable Designs} \]

This chapter presents the main components of a schema-based method for constructing typed logic programs. That is, schemata, DTs and the schema instantiation operation.

### 2.2 Data Types

The construction of typed logic programs is considered in this thesis. The language underlying this approach is many sorted logic with parametric polymorphism [36], [37]. The alphabet of the language contains disjoint infinite sets \( V, F, P, A, C \) 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 thesis 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 to functions.

**Definition 2.2.1:** Let \( \alpha \) be a parameter. Let \( B \) and \( C \) stand for basic types and type constructors of arity \( \geq 1 \), respectively. A type is defined inductively as follows:

1. A parameter \( \alpha \) is a type.
2. A basic type \( B \) is a type.
3. 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 parameterised (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. For instance, we can define the type \( \text{seq}(\alpha) \) where
α is a parameter indicating the type of the elements of the sequence. If α is instantiated to all possible ground types then we will get the types which seq(α) represents. If α is instantiated to the ground type set(N) (set of integers) then the ground type seq(set(N)) is a sequence whose elements are sets of integers, and so on.

2.3 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 $\land$, $\neg$ and $\rightarrow$, when they occur within schemata, stand for the same object level symbols with their usual meaning.
- The lower case Greek letters $\alpha$, $\beta$ and $\epsilon$, possibly subscripted and/or superscripted are parameter variables and they stand for parameters.
- The lower case Greek letters $\tau$, $\rho$, $\gamma$ and $\delta$, possibly subscripted and/or superscripted are type variables and they stand for arbitrary types.

Definition 2.3.1: 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 2.3.2: A literal schema has the form $A$ or $\neg A$, where $A$ is an atom schema.

Definition 2.3.3: A clause schema has the form $A \leftarrow L_1 \land \ldots \land L_n$, where $A$ is an atom schema and $L_i (1 < i < n)$ are literal schemata.

Definition 2.3.4: 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 $\alpha_i (1 < i < n)$ are parameter variables.

Definition 2.3.5: 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 $\alpha_i (1 < i < n)$.
**Definition 2.3.6**: 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 2.3.7**: A *typed procedure schema* is a set of typed clause schemata with the same predicate variable in the head of each clause.

**Definition 2.3.8**: 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**
- $Type(P) : \alpha_1 \times \alpha_2$
- $Type(Q) : \alpha_1$
- $Type(R) : \alpha_2$
- $Type(S) : \alpha_1 \times \alpha_2$

**Clause Schemata**
- $P(u_1, u_2) \leftarrow Q(u_1) \land R(u_2)$
- $P(u_1, u_2) \leftarrow S(u_1, u_2)$

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 $\alpha_1, \alpha_2$ cannot be renamed.

**2.4 Schema Instantiation**

**Definition 2.4.1**: Let $\Sigma$ be a typed program schema. A *predicate substitution* for $\Sigma$ is defined to be $\{P_i/p_i, \ldots, P_m/p_m\}$ 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 $\Sigma$. 

35
If a predicate variable $P$ occurs in two clause schemata in $\Sigma$ then it is the same $P$, and a predicate substitution for $P$ assigns the same predicate symbol in each occurrence of $P$.

**Definition 2.4.2:** Let $\Sigma$ 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 $\Sigma$ is defined to be $\theta_1 \cup \ldots \cup \theta_r$.

**Definition 2.4.3:** Let $\alpha_1, \ldots, \alpha_n$ be the parameter variables that occur in the type schemata for the predicate variables of schema $\Sigma$. Given types $\tau_1, \ldots, \tau_n$, $\{\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n\}$ is defined to be a type substitution for $\Sigma$.

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

**Definition 2.4.4:** Let $\Sigma$ be a typed program schema, and let $\Theta_P$ and $\Theta_T$ be a predicate substitution and a type substitution respectively for $\Sigma$. Let $P$ be a predicate symbol in $\Sigma$ 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 2.4.5:** Let $\Sigma$ be a typed program schema. Let $\Theta_P$, $\Theta_A$ and $\Theta_T$ be predicate, argument and type substitutions of schema $\Sigma$ respectively. A schema substitution $\Theta$ consists of the three components $\Theta_P$, $\Theta_A$ and $\Theta_T$. An instance of schema $\Sigma$ is defined to be $\Sigma\Theta_P\Theta_A\Theta_T$.

Note that in the rest of this thesis we will use $\Sigma\Theta$ to stand for $\Sigma\Theta_P\Theta_A\Theta_T$.

In program construction, we aim to construct polymorphic many-sorted programs. Therefore we will require instances of schemata to be polymorphic many-sorted formulas. We assume the definitions of polymorphic many-sorted term, atom and formula from [36]. In Section 3.5 and in Section 3.8 we will see that the components $\Theta_P$, $\Theta_A$ and $\Theta_T$ are constructed in such a way as to produce polymorphic many-sorted formulas.
Chapter 3

A Schema-Based Method for Developing Logic Programs

3.1 Introduction

This chapter presents a method for constructing typed, moded logic programs by stepwise top-down design using schemata, DTs and modes [61].

The choice of the set of available schemata is very important for a schema-based method. A small set of schemata, 5, have been defined which are independent of specific data types. These schemata are problem-independent representations of algorithm design strategies. Each schema contains enough structure to be useful in the design process. We have found this set of schemata to be very expressive. Each schema consists of a set of clause schemata together with type and mode schemata for each predicate variable occurring in them. Modes in this method support the application of the refinement operations. The type and mode schemata play a significant role in the construction process.

The choice of the DTs is also important for this method. We propose a set of built-in DTs which form a kernel for defining and implementing user-defined DTs. User-defined DTs can be specified in terms of the built-in ones or other previously defined DTs. They are implemented by using this schema-based method.

Definition 3.1.1: Let $\text{Prog}$ be a normal logic program. Let $p$ be an n-ary predicate symbol
in \textit{Prog}. The predicate $p/n$ is called an \textit{undefined predicate} if there are no clauses whose heads contain the predicate symbol $p$.

\textbf{Definition 3.1.2}: Let \textit{Prog} be a normal logic program. The \textit{definition} (or \textit{procedure}) of the predicate $p/n$ is the subset of clauses of \textit{Prog} consisting of all clauses with the predicate symbol $p$ in their heads.

A program is constructed by this method by successively refining undefined predicates. The lowest refinement level involves refinement by DT operations. Four refinement operations are used in the refinement process.

- \textit{Tupling} is an operation applied to a predicate which is about to be defined, in order to convert a predicate of any arity, to a binary predicate. This is necessary because the top predicate variables of the 5 schemata are binary.
- \textit{Refinement by a schema} which is basically the instantiation operation.
- \textit{Flattening} which is the opposite of tupling, converts an instance of a schema whose arguments are tuples of variables into an instance each of whose arguments is a variable.
- \textit{Refinement by a DT operation} which involves giving the definition of an undefined predicate by a DT operation.

Finally, we show that the programs which are constructed by this method satisfy the following properties. They are polymorphic many-sorted programs. They satisfy the head condition and the transparency condition \cite{36}, \cite{37} which 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.

This program development method assumes that the programmer has a algorithmic understanding of his/her problem. The definitions of the 5 schemata include informal guidance to the programmer for the next design steps. That is, the programmer “knows” which schema or DT operation to apply at each design step in order to meet her/his design goals.

\section{Design by Stepwise Top-Down Refinement}

Top-down design is a process of successive refinement of a design. At each stage in the process there are a number of “undefined” parts of the design. A refinement step replaces some undefined part by a more detailed construct, which itself may contain further undefined
subparts as well as constructs from some executable target language $L$. The process is finished when the design is entirely in $L$.

The main justification of top-down design is that it gives a goal-oriented structure to the design process, and suggests a “compositional” style of program in which each refinement step can be independently justified or proved correct. Apart from this, very little is implied by the phrase “top-down design”. The actual refinement steps may take many forms.

Our belief is that, if tools to support top-down design are to be practical, refinement steps should correspond to the set of basic constructs in the target language $L$. That is, each refinement step should introduce one language construct. For example, if $L$ is a Pascal-like language we could insist that each refinement would introduce only one while, repeat, if-then-else or statement composition construct. The target language $L$ therefore plays a very important role in top-down design since each design decision has to be expressed by a single construct. The constructs available in $L$ should be sufficiently high-level to be regarded as part of the design rather than implementation.

In the approach of this thesis each refinement step introduces either one instance of a program schema or a DT operation. In a loose sense, therefore, the choice of schemata and DTs defines a “design language” whose basic constructs consist of the set of schemata and the set of DT operations.

The separation of schemata from DTs is an important aspect of this approach since each schema is independent of any particular types. Design decisions therefore need not confuse control with data. This is unlike most schemata in the logic programming literature, which are associated with specific types (such as lists and trees).

### 3.3 Modes

The mode of arguments of a predicate can be seen in two possible states, that is before the call and after the success of the predicate. The mode of a predicate in these two states is called *call mode* and *success mode* respectively.

The *(call)* mode of a predicate indicates how its arguments are instantiated when that predicate is called. The *success mode* of a predicate indicates how its arguments are instantiated when that predicate succeeds.

The program schemata in this approach have been constructed with the left-to-right depth-first computation rule of Prolog in mind. The call modes in this method depend on this
computation rule.

The modes which are used in this thesis are i and d. They stand for “input” and “don’t know” respectively. i identifies the arguments which are expected to be ground. These terms are characterized as “input”. An argument with any instantiation can be assigned mode d.

Modes in this thesis 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 modes of the arguments of the undefined predicate.

The modes associated with schema arguments are called “expected” modes. By contrast, the modes inferred by the mode analysis procedure are the actual runtime modes. Expected modes and inferred modes do not necessarily agree. In fact an expected i mode might be d at runtime, and an expected d mode might be i. Hence the expected modes only provide a heuristic for splitting arguments into input and output groups. However the heuristic is safe in the sense that an argument that is expected to be i will either be inferred to be i, or not used at all.

Modes are also used for the refinement of an undefined predicate by a DT operation or equality. Given a partial design 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 modes of calls to DT predicates for compatibility with the declared modes. The mode inference procedure of this thesis is an abstract interpretation similar in many respects to other mode inference procedures such that in [18].

3.4 Five Design Schemata

Algorithm design strategies are used for the design of algorithms [3], [45]. A small set of algorithm design strategies which is powerful and flexible enough would be desirable. In addition, we like to express algorithm design strategies independently of particular problem representations in order to be able to apply them to problems which require different representations.

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 realisations/instantiations for each problem to which they are applied.
Definition 3.4.1: A mode schema for predicate variable $P$ of arity $n$, denoted by $\text{Mode}(P)$, is $\text{Mode}(P) = m_1, \ldots, m_n$ where each $m_i$ ($1 \leq i \leq n$) is either $i$ or $d$.

Given $P(u_1, \ldots, u_n)$ (resp. $p(t_1, \ldots, t_n)$), an atom schema (resp. atom), where $\text{Mode}(P)$ (resp. $\text{Mode}(p)$) is $m_1, \ldots, m_n$, we sometimes use the notation $\text{Mode}(u_j) = m_j$ (resp. $\text{Mode}(t_j) = m_j$).

The mode schema for a predicate variable represents the expected use of the arguments, and will be used during schema refinement as a heuristic to partition the arguments into two groups.

Definition 3.4.2: 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 method which are problem-independent algorithm design strategies. 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. Each schema has its own contribution to the overall computational power of this set of 5 schemata. We illustrate by examples their computational power. That is, two examples of non-trivial programs constructed by the method of this thesis are shown in Appendix B.

3.4.1 Comparison with Similar Schemata in the Literature

We are not aware of any proposal in the literature for a set of schemata analogous to Subgoal, Case and Search schemata of this thesis.

A divide and conquer schema which is analogous to Incremental and Divide-and-conquer schemata of this thesis is proposed in [25], [26]. The approach in [25], [26] does not make instantiations of schemata to construct logic programs. The divide and conquer schema in [25], [26] guides the synthesis of logic programs. The Incremental schema of this thesis is assumed in [25], [26] as a special case of divide and conquer schema. It should be noted that the Incremental schema does not perform a divide and conquer strategy. It rather processes the elements of an inductive DT one-by-one. In the form given here, Incremental, cannot be seen as a degenerate case of Divide-and-conquer. The representations of the smaller problems which the divide and conquer strategy creates have to be of the same type as the
initial one. The representations of smaller problems in Incremental strategy need not be of the same type as the original one. For example, if the initial problem is represented by \( \text{seq}(\alpha) \) the Incremental strategy divides it into subproblems \( P_2 \) and \( P_3 \) which are represented by \( \text{seq}(\alpha) \) and \( \alpha \) respectively. A real divide and conquer strategy would divide \( P_1 \) into \( P_2 \) and \( P_3 \) such that both are represented by \( \text{seq}(\alpha) \). That is, the subproblems in divide and conquer strategy should have same representation as the initial problem. In untyped logic you may neglect this difference but not in typed logic.

Incremental could be seen as a degenerate case of Divide-and-conquer if the types in Divide-and-conquer were slightly generalized. However it is worth keeping these two separate since Incremental is used much more frequently than Divide-and-conquer.

The logic description schema in [20], [21] which applies when a generalization strategy is not followed seems to be analogous to this Incremental schema. If we have a closer look in the two schemata we discover important differences. The literal schema minimal/1 in [20], [21] does not correspond to Terminating/1 literal schema of Incremental. The instantiation of minimal/1 is based on the minimum element of a well-founded set. The refinement of Terminating/1 is based on the particular problem and the inductively defined DT. Terminating/1 is intended to terminate the recursive decomposition of the elements of the inductively defined DT no matter whether we have the minimal element of a well-founded set or not. For example, the implementation in this method of the efface/3 predicate which is presented in [20], [21] shows this difference. efface(X,L1,L2) is true if the list L2 is the list L1 without the first occurrence of X. The minimal(L) in [20], [21] where L is a list is instantiated to the empty list, i.e. \( L = [\] \), which is the minimal element of a well-founded order on lists. In this method, the instance predicate of Terminating(X,L) is instantiated by \( L = <\text{} :: \text{} \rangle \wedge H = X \) where L is a sequence and :: is the constructor for sequences. The case \( L = <> \) is false and it is not considered by the method of this thesis. The case \( L = <\text{} :: \text{} \rangle \wedge H = X \) is implemented in [20], [21] by the instance of the recursive schema clause. Unfortunately, it is not given any formal or informal meaning to the literal schemata in [20], [21] so it is difficult to discuss possible analogies. Let us compare the literal schema directly_solve/2 in [20], [21] with the literal schema Initial_results/2 of Incremental which seem to be corresponding at least in their position in the two schemata. directly_solve/2 implies that the solution for this case is trivial and it is solved directly without recursion. Initial_results/2 does not imply that the construction of the solution is trivial. Instances of this literal schema may be defined recursively.

Finally, it should be noted that the definitions of the schemata in [20], [21], [25], [26] do not include type and mode schemata as do the schemata of this thesis.

Each schema is presented and discussed in turn. The predicate variables are given names
that indicate their intended meaning. The intended meaning for all literal schemata is described informally.

### 3.4.2 Incremental.

The *Incremental* schema assumes that the input data is an inductively defined type $\tau$. The inductive definition of $\tau$ includes a constructor operation which builds an element of $\tau$ from another element of $\tau$ 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**
- $Type(Incr) : \alpha_1 \times \alpha_2$
- $Type(Terminating) : \alpha_1$
- $Type(Initial\_result) : \alpha_1 \times \alpha_2$
- $Type(Deconstruction) : \alpha_1 \times \alpha_3 \times \alpha_1$
- $Type(Non\_initial\_result) : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2$

**Mode Schemata**
- $Mode(Incr) : i, d$
- $Mode(Terminating) : i$
- $Mode(Initial\_result) : i, d$
- $Mode(Deconstruction) : i, d, d$
- $Mode(Non\_initial\_result) : i, i, i, d$

**Incremental Schema**

\[
Incr(u_1, u_2) \leftarrow
\]
\[
\begin{align*}
& Terminating(u_1) \land \\
& Initial\_result(u_1, u_2)
\end{align*}
\]

\[
Incr(u_1, u_2) \leftarrow
\]
\[
\begin{align*}
& \neg Terminating(u_1) \land \\
& Deconstruction(u_1, v_1, v_2) \land \\
& Incr(v_2, v_3) \land \\
& Non\_initial\_result(u_1, v_1, v_3, v_2)
\end{align*}
\]

Note that for all schemata the schema argument variable $u_1$ is assumed to represent the input data and $u_2$ the results of the computations. The data represented by the other
arguments will be discussed with the corresponding literal schema where they appear.

- **Terminating \( u_1 \):** 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. \( u_1 \), 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 \( u_1, u_2 \):** 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 \( u_2 \) by using arguments from the tuple \( u_1 \).

- **Deconstruction \( u_1, v_1, v_2 \):** Refining instances of this literal schema should aim at the decomposition of an argument from the tuple of input data, i.e. \( u_1 \), whose type is defined inductively. This decomposition should split that argument into an element and the remaining part which are represented by the arguments \( v_1 \) and \( v_2 \) respectively. The remaining part, i.e. \( v_2 \), is expected to be further decomposed. Note that the remaining part has type same as the type of the inductive argument, i.e. \( u_1 \). On the other hand the component, i.e. \( v_1 \), may have different type than the one of the inductive argument.

- **Non\_initial\_result \( u_1, v_1, v_3, u_2 \):** 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 \( u_2 \) by using \( u_1, v_1 \) and \( v_3 \). Note, that the tuple of arguments \( v_3 \) represents the results from processing the remaining part of the argument which is decomposed, i.e. \( v_2 \).

It is worth noting that there are problems whose implementation requires some arguments of the tuple \( u_1 \) to be used in instances of atom schema Non\_initial\_result. \( u_1 \) is used as argument in atom schema Non\_initial\_result in order to support the construction of such programs. Another solution is to pass these arguments through the argument variable \( v_1 \). That is, the required arguments will have to be unified with corresponding arguments of the tuple \( v_1 \) in instances of atom schema Deconstruction. The last solution adds complexity into the constructed programs by applying additional design decisions.

### 3.4.3 Divide-and-conquer.

The Divide-and-conquer schema assumes that the type \( \tau \) of an argument in the tuple of the input data \( u_1 \) 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. The schema is of the following form.

**Type Schemata**

- Type(DivConq): \( \alpha_1 \times \alpha_2 \)
- Type(Terminating): \( \alpha_1 \)
- Type(Initial.result): \( \alpha_1 \times \alpha_2 \)
- Type(Decomposition): \( \alpha_1 \times \alpha_1 \times \alpha_1 \)
- Type(Composition): \( \alpha_1 \times \alpha_2 \times \alpha_2 \times \alpha_2 \)

**Mode Schemata**

- Mode(DivConq): \( i, d \)
- Mode(Terminating): \( i \)
- Mode(Initial.result): \( i, d \)
- Mode(Decomposition): \( i, d, d \)
- Mode(Composition): \( i, i, i, d \)

**Divide-and-conquer Schema**

\[
\begin{align*}
\text{DivConq}(u_1, u_2) & \leftarrow \\
& \quad \text{Terminating}(u_1) \land \\
& \quad \text{Initial.result}(u_1, u_2) \\
\text{DivConq}(u_1, u_2) & \leftarrow \\
& \quad \neg \text{Terminating}(u_1) \land \\
& \quad \text{Decomposition}(u_1, v_1, v_2) \land \\
& \quad \text{DivConq}(v_1, v_2) \land \\
& \quad \text{DivConq}(v_3, v_4) \land \\
& \quad \text{Composition}(u_1, v_3, v_4, u_2)
\end{align*}
\]

Note that the same comments as the ones for Incremental schema apply for the literal schemata \( \text{Terminating}(u_1) \) and \( \text{Initial.result}(u_1, u_2) \) of this schema as well.

- \( \text{Decomposition}(u_1, v_1, v_2) \): Refining instances of this literal schema should aim at the decomposition of an argument with inductively defined type from the tuple \( u_1 \). This decomposition should split the input data represented by this argument into two parts. The decomposed data are represented by the arguments \( v_1 \) and \( v_2 \) which have type same as the argument \( u_1 \).

- \( \text{Composition}(u_1, v_3, v_4, u_2) \): Refining instances of this literal schema should aim at the construction of the solution for the general cases of the problem. This solution
is constructed by composing the partial solutions resulting from the processing of
the arguments which represent the two parts of the input data. The solution is
constructed into the tuple $u_2$ by using the arguments $v_3$ and $v_4$ which represent the
partial solutions and possibly some argument(s) of $u_1$.

Note that the same comments apply for the use of the schema argument variable $u_1$ in
the atom schema Composition as the ones for the atom schema Non_initial_result of
Incremental schema.

### 3.4.4 Subgoal and Case Schemata.

Problem reduction is a problem solving method where the initial problem description is
reduced in two or more subproblems which can be further reduced into smaller subproblems
and so on [72]. Each of the subproblems can be solved by any method. There are and and or
reductions. In and reductions, all subproblems imply the initial problem. In or reductions,
each subproblem implies the initial problem. and and or reductions are represented by
Subgoal and Case schemata respectively.

#### 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. The
schema is as follows.

**Type Schemata**

- $Type(SubGoal): \alpha_1 \times \alpha_2$
- $Type(SubGoal1): \alpha_1 \times \alpha_3 \times \alpha_2$
- $Type(SubGoal2): \alpha_1 \times \alpha_3 \times \alpha_2$
- ...  
- $Type(SubGoalN): \alpha_1 \times \alpha_3 \times \alpha_2$

**Mode Schemata**

- $Mode(SubGoal): i, d$
- $Mode(SubGoal1): i, d, d$
- $Mode(SubGoal2): i, i, d$
- ...  
- $Mode(SubGoalN): i, i, d$
Subgoal Schema

\[ \text{SubGoal}(u_1, u_2) \leftarrow \]
\[ \text{SubGoal1}(u_1, v, u_2) \land \]
\[ \text{SubGoal2}(u_1, v, u_2) \land \]
\[ \ldots \]
\[ \text{SubGoalN}(u_1, v, u_2) \]

- \( \text{SubGoal1}(u_1, v, u_2) \): Refining instances of this literal schema should aim at the construction of solution for the first subproblem of the initial problem. The arguments of the tuple \( u_2 \) which are assumed to represent the first subproblem are instantiated by using \( u_1 \). If the tuple \( v \) is not empty its argument has to be instantiated by the instances of this literal schema. The argument \( v \) is intended to transfer intermediate computations from instances of \( \text{SubGoal1}(u_1, v, u_2) \) to instances of next atom schemata, i.e. \( \text{SubGoal2}(u_1, v, u_2), \ldots, \text{SubGoalN}(u_1, v, u_2) \).

- \( \text{SubGoal2}(u_1, v, u_2) \): Refining instances of this literal schema should aim at the construction of solution for the second subproblem of the initial problem. The arguments of the tuple \( u_2 \) which are assumed to represent the second subproblem are instantiated by using \( u_1 \). If the tuple \( v \) is not empty its argument can be used by the instances of this schema literal.

- \( \text{SubGoalN}(u_1, v, u_2) \): Same as \( \text{SubGoal2}(u_1, v, u_2) \).

Case.

The Case schema reduces a problem to two or more independent subproblems. 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. The schema is as follows.

Type Schemata

\[ \text{Type(Case)} : \alpha_1 \times \alpha_2 \]
\[ \text{Type(Case1)} : \alpha_1 \times \alpha_2 \]
\[ \text{Type(Case2)} : \alpha_1 \times \alpha_2 \]
\[ \ldots \]
\[ \text{Type(CaseN)} : \alpha_1 \times \alpha_2 \]

Mode Schemata

\[ \text{Mode(Case)} : i, d \]
\[ \text{Mode(Case1)} : i, d \]
Mode(Case2) : i, d

Mode(CaseN) : i, d

**Case Schema**

\[ \text{Case}(u_1, u_2) \leftarrow \text{Case1}(u_1, u_2) \]

\[ \text{Case}(u_1, u_2) \leftarrow \text{Case2}(u_1, u_2) \]

\[ \ldots \]

\[ \text{Case}(u_1, u_2) \leftarrow \text{CaseN}(u_1, u_2) \]

- **Case1**\((u_1, u_2)\): Refining instances of this literal schema should aim at the construction of a solution for the first independent subproblem. The solution will be constructed into arguments of \(u_2\) by using arguments of the tuple \(u_1\).

- **Case2**\((u_1, u_2)\): Refining instances of this literal schema should aim at the construction of a solution for the second independent subproblem. The solution will be constructed into the same argument(s) of \(u_2\) as the ones used for the solution of the first independent subproblem. Similarly, arguments of the tuple \(u_1\) are used for constructing the solution for this independent subproblem as well.

- **CaseN**\((u_1, u_2)\): Same as **Case2**\((u_1, u_2)\).

### 3.4.5 Search.

A state of a problem is a particular configuration of the problem representation. The space of states of a problem are 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, 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. The schema is as follows.

**Type Schemata**

- **Type(Search)**: \( \alpha_1 \times \alpha_2 \)
- **Type(Initial_state_solution)**: \( \alpha_1 \times \alpha_3 \times \alpha_2 \)
- **Type(Search1)**: \( \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2 \)
- **Type(Search_termination)**: \( \alpha_3 \times \alpha_2 \)
- **Type(Solution_assignment)**: \( \alpha_2 \times \alpha_2 \)
- **Type(Apply_operation)**: \( \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2 \)
- **Type(Backtracking)**: \( \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2 \)

**Mode Schemata**

- **Mode(Search)**: \( i, d \)
- **Mode(Initial_state_solution)**: \( i, d, d \)
- **Mode(Search1)**: \( i, i, i, d \)
- **Mode(Search_termination)**: \( i, i \)
- **Mode(Solution_assignment)**: \( i, d \)
- **Mode(Apply_operation)**: \( i, i, i, d, d \)
- **Mode(Backtracking)**: \( i, i, d, d \)

**Search Schema**

- **Search**(\( u_1, u_2 \)) \[ \rightarrow \]
  
  - **Initial_state_solution**(\( u_1, v_1, v_2 \)) \( \land \) 
  - **Search1**(\( u_1, v_1, v_2, u_2 \))

- **Search1**(\( u_1, v_1, v_2, u_2 \)) \[ \rightarrow \]
  
  - **Search_termination**(\( v_1, v_2 \)) \( \land \) 
  - **Solution_assignment**(\( v_2, u_2 \))

- **Search1**(\( u_1, v_1, v_2, u_2 \)) \[ \rightarrow \]
  
  - \( \neg \text{Search_termination}(v_1, v_2) \land \) 
  - **Apply_operation**(\( u_1, v_1, v_2, v_3, v_4 \)) \( \land \) 
  - **Search1**(\( u_1, v_3, v_4, u_2 \))

- **Search1**(\( u_1, v_1, v_2, u_2 \)) \[ \rightarrow \]
  
  - \( \neg \text{Search_termination}(v_1, v_2) \land \) 
  - \( \neg \text{Apply_operation}(u_1, v_1, v_2, \ldots) \land \) 
  - **Backtracking**(\( v_1, v_2, v_3, v_6 \)) \( \land \) 
  - **Search1**(\( u_1, v_5, v_6, u_2 \))

49
It is worth noting that the parameter $a_3$ is instantiated to a data type which represents the implementation of the search stack. In addition, anonymous variables are assumed to be existentially quantified immediately before the atom they occur.

- **Initial-state solution**($u_1$, $v_1$, $v_2$): Refining instances of this literal schema should aim at the construction of the initial state of the space of states and of the initial solution. The instances of this schema literal should make appropriate initializations to the search stack represented by $v_1$ and to the arguments of $v_2$ which represent the solution by using the input data, i.e. $u_1$.

- **Search termination**($v_1$, $v_2$): Refining instances of this literal schema should aim at the construction of a test which determines if search should stop. Search may stop because the space of states as represented by the stack $v_1$ has been exhausted. Search may also stop because the desired solution represented by the tuple $v_2$ has been constructed no matter what is the state of the space of states.

- **Solution assignment**($v_2$, $u_2$): Refining instances of this literal schema should assign the constructed solution represented by arguments of $v_2$ to the corresponding arguments of $u_2$. The intended use of $u_2$ is to pass the constructed solution upwards.

- **Apply operation**($u_1$, $v_1$, $v_2$, $v_3$, $v_4$): Refining instances of this literal schema should aim at the implementation of the operations which can be applied to states of the state space. Its instances should construct the new search stack represented by the argument $v_3$ and the new solution represented by $v_4$. The new search stack is constructed by using the old search stack represented by $v_1$ and possibly arguments from the tuple $u_1$. The new solution is constructed by using the old solution represented by arguments of the tuple $v_2$, the old search stack $v_1$ and possibly arguments from $u_1$.

- **Backtracking**($v_1$, $v_2$, $v_3$, $v_6$): Refining instances of this literal schema should aim at the implementation of a backward move in the search tree represented by the search stack. Its instances should construct the new search stack represented by $v_5$ and the new solution represented by $v_6$. The new search stack $v_5$ is constructed by using the old search stack represented by $v_1$. The new solution $v_6$ is constructed by using the old search stack, i.e. $v_1$, and the old solution represented by $v_2$.

**Proposition 3.4.1**: The schemata *Incremental*, *Divide-and-conquer*, *Subgoal*, *Case* and *Search* are polymorphic many-sorted program schemata.

**Proof**: The construction of these schemata has been based on definition 2.2.8. That is, by construction they are polymorphic many-sorted program schemata.
3.5 Refinement Operations

The aim of this section is to derive the components of the schema substitution $\Theta$, namely $\Theta_P$, $\Theta_A$ and $\Theta_T$. In addition, it presents the refinement operations which are performed in order to define an undefined predicate.

**Definition 3.5.1:** Let $p/n$ be a predicate with type $\text{Type}(p) = \tau_1 \times \ldots \times \tau_n$. 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 $p/n$.

Note that each $x_i$ $(1 \leq i \leq n)$ has type $\tau_i$ in $p(x_1, \ldots, x_n)$. In addition, if $t_1, \ldots, t_n$ are terms then $(t_1, \ldots, t_n)$ is a term.

There are two main kinds of refinements, *refinement by a schema* and *refinement by a DT operation or equality*.

### 3.5.1 Refinement by a Schema

Three refinement operations, namely *tupling*, *schema refinement of a typed atom* and *flattening* are applied in order to define a predicate by instantiating a typed, moded program schema $\Sigma$. The three refinement operations are as follows.

1. Initially, the arguments of the predicate that we want to define are divided into two tuples based on their expected modes. The expected mode of the top-level predicate also is given by the programmer. The expected mode of a predicate $p$ is derived from the expected mode of the predicate variable for which $p$ was substituted. The corresponding tuples of types are constructed as well. That is, the tuples of arguments $x^i, x^d$ and the tuples of types $\tau^i, \tau^d$ are constructed.

2. Next, the schema substitution $\Theta$ is derived in order to construct an instance $\Sigma \Theta$ of schema $\Sigma$. For each schema clause $C_1, \ldots, C_r$ in $\Sigma$ a clause $c'_1, \ldots, c'_r$ of the instance is derived respectively. That is, $c'_i = C_i \Theta_P \Theta_A$ $(1 \leq i \leq r)$ where $\Theta_P$ is the predicate substitution and $\Theta_A$ is the argument substitution of the schema substitution $\Theta$. For each $P/p \in \Theta_P$ the type of the predicate $p$ is derived from the type of the corresponding predicate variable $P$. That is, $\text{Type}(p) = \text{Type}(P) \Theta_T$ where $\Theta_T$ is the type substitution of $\Theta$.
3. Finally, the clauses \( c_1, \ldots, c_r \) are derived by flattening the corresponding clauses \( c'_1, \ldots, c'_r \). In addition, the types of the predicates in \( c'_1, \ldots, c'_r \) are flattened. That is, the tuple structure is removed.

### 3.5.1.1 Tupling a Typed cga

**Definition 3.5.2.** Let \( p(x_1, \ldots, x_n) \) be a typed cga with type \( \tau_1 \times \cdots \times \tau_n \) and expected mode \( m_1, \ldots, m_n \). The tuples \( \bar{x}^i \) and \( \bar{x}^d \) are defined as follows.

\[
\bar{x}^i = (y_1, \ldots, y_{n_1}) \quad \text{where} \quad \text{Mode}(y_j) = i \quad (1 \leq j \leq n_1), \quad y_j \in \{x_1, \ldots, x_n\}.
\]

\[
\bar{x}^d = (z_1, \ldots, z_{n_2}) \quad \text{where} \quad \text{Mode}(z_j) = d \quad (1 \leq j \leq n_2), \quad z_j \in \{x_1, \ldots, x_n\}.
\]

\( \{y_1, \ldots, y_{n_1}\} \cup \{z_1, \ldots, z_{n_2}\} = \{x_1, \ldots, x_n\} \) and \( \{y_1, \ldots, y_{n_1}\} \cap \{z_1, \ldots, z_{n_2}\} = \emptyset \). The variables in the tuples \( \bar{x}^i \) and \( \bar{x}^d \) have the same order that they have in \( p(x_1, \ldots, x_n) \). \( \bar{\tau}^i \) and \( \bar{\tau}^d \) are defined to be the tuples of types of \( \bar{x}^i \) and \( \bar{x}^d \) respectively.

The *tupling* refinement operation constructs the tuples of variables and types \( \bar{x}^i, \bar{x}^d, \bar{\tau}^i \) and \( \bar{\tau}^d \).

### 3.5.1.2 Schema Refinement of a Typed Atom

**Derivation of Predicate Substitution**

Let \( \Sigma \) be a typed, moded program schema. Let \( P, P_1, \ldots, P_k \) be the predicate variables which occur in \( \Sigma \). Let \( P \) be the top predicate variable of \( \Sigma \). Let \( p/n \) be an undefined predicate and let \( p(x_1, \ldots, x_n) \) be a typed atom for \( p/n \). The predicate substitution \( \Theta_P \) is derived as follows.

\[
\Theta_P = \{P/p, P_1/p_1, \ldots, P_k/p_k\}
\]

where \( p_1, \ldots, p_k \) are distinct fresh predicate symbols not so far used in the program being constructed.
Derivation of Argument Substitution

Let \( \Sigma \) be a typed, moded program schema, whose argument variables have been renamed so that no argument variable appears in more than one schema clause. Let \( C \) be a typed clause schema of schema \( \Sigma \). Let the atom schema \( Q(u_1, \ldots, u_n) \) be the head the clause \( C \). Let \( p/n \) be an undefined predicate and let \( p(\overline{t}_1, \ldots, \overline{t}_n) \) be a typed atom for \( p/n \). The argument substitution \( \theta \) for clause \( C \) and \( p(\overline{t}_1, \ldots, \overline{t}_n) \) is derived as follows.

\[
\theta = \{ u_1/\overline{t}_1, \ldots, u_n/\overline{t}_n, u'_1/\overline{y}_1, \ldots, u'_d/\overline{y}_d \}
\]

where \( u'_1, \ldots, u'_d \) are schema arguments variables which occur in the body of clause \( C \) and \( \overline{y}_1, \ldots, \overline{y}_d \) are tuples of arity 1 of fresh individual variables, \( \overline{y}_j = (y_j) \) (1 \( \leq j \leq d \)).

Note that \( \{ u_1/\overline{t}_1, \ldots, u_n/\overline{t}_n \} = mgu(Q(u_1, \ldots, u_n)\Theta_P, p(\overline{t}_1, \ldots, \overline{t}_n)) \) where \( \Theta_P \) is a predicate substitution for \( \Sigma \) and \( Q/p \in \Theta_P \). In addition, if \( C_1, \ldots, C_r \) are the clauses of \( \Sigma \) with argument substitutions \( \theta_1, \ldots, \theta_r \) respectively then \( \Theta_A = \theta_1 \cup \ldots \cup \theta_r \). It is shown later in this section that the final argument substitution is derived by extending \( \Theta_A \). Finally, it is worth noting that the head atom schema in the clauses of the top procedure schema is unified with a binary typed atom because the top predicate variables of the 5 schemata have arity 2. If a program schema consists of more than one procedure schema then only top predicate variable has to be binary.

Derivation of Type Substitution

**Definition 3.5.3:** Let \( \Sigma \) be a program schema with top predicate variable \( P \) of arity 2. Let \( Type(P) = \alpha_1 \times \alpha_2 \). Let \( p(\overline{x}^t, \overline{x}^d) \) be a typed atom with types \( \overline{\rho}^t \) and \( \overline{\rho}^d \). Let \( \varepsilon_1, \ldots, \varepsilon_k \) be the parameter variables of the type schemata of \( \Sigma \) other than the parameter variables occurring in \( Type(P) \). The type substitution \( \Theta_T \) for the type schemata of \( \Sigma \) and \( p(\overline{x}^t, \overline{x}^d) \) is derived as follows.

\[
\Theta_T = \{ \alpha_1/\overline{\rho}^t, \alpha_2/\overline{\rho}^d, \varepsilon_1/\overline{\beta}_1, \ldots, \varepsilon_k/\overline{\beta}_k \}
\]

where \( \overline{\beta}_1, \ldots, \overline{\beta}_k \) are tuples of fresh parameters of arity 1.

Extending the Argument Substitution

The argument substitution is extended so that variables whose type is a tuple (of arity greater than 1) are replaced by an appropriate tuple of variables. This is necessary in order to remove all tuples from the clauses by applying the flattening operation which is defined below.
Let $\Theta'_A$ be an argument substitution and let $\Theta_T$ be a type substitution for schema $\Sigma$. For each $u/y \in \Theta'_A$ where

1. $\bar{y}$ has arity 1, i.e. $\bar{y} = (y)$, and
2. schema type of $u$ is $\alpha$, and
3. arity of $\bar{f}$ is greater than 1, $\alpha/\bar{f} \in \Theta_T$

replace $\bar{y}$ by a tuple of fresh variables $\bar{y}'$ with arity same as the arity of $\bar{f}$. The extended argument substitution $\Theta_A$ with respect to $\Theta_T$ is derived as follows.

$$\Theta_A = \Theta'_A \setminus \{u/y\} \cup \{u/\bar{y}'\}$$

**Application of Schema Substitution $\Theta$**

Let $\Sigma$ be a typed, moded program schema with clauses $C_1, \ldots, C_r$ renamed so that no argument variable appears in more than one clause. Let $\Theta$ be a schema substitution for $\Sigma$ with components $\Theta_P, \Theta_A$ and $\Theta_T$.

An instance $\Sigma \Theta$ is derived as follows. A clause $c'_i$ ($1 \leq i \leq r$) is derived for each schema clause $C_i$ by applying the predicate and argument substitutions to $C_i$. That is, $c'_i = C_i \Theta_P \Theta_A$. For each $p_i \in \Theta_P$ ($1 \leq i \leq m$) its type $Type(p_i)$ is $Type(P_i) \Theta_T$.

### 3.5.1.3 Flattening

**Flattening Tupled Clauses**

**Definition 3.5.4:** Let $a = p(t_1, \ldots, t_n)$ be an atom, where each $t_i$ ($1 \leq i \leq n$) is a tuple of variables. $flatten(a)$ is defined to be $p(y_1, \ldots, y_s)$ where $y_1, \ldots, y_s$ is the sequence of variables in $t_1, \ldots, t_n$ and $s \geq n$. Let $c$ be the program clause $h \leftarrow l_1, \ldots, l_k$ where $h$ is atom and $l_i$ ($1 \leq i \leq k$) are literals.

$$flatten(c) = flatten(h \leftarrow l_1, \ldots, l_k) =$$

$$flatten(h) \leftarrow flatten(l_1), \ldots, flatten(l_k)$$

where

$flatten(l_i) = \neg flatten(a_i)$, ($1 \leq i \leq k$) if $l_i$ is a negative literal $\neg a_i$;

$flatten(l_i) = flatten(a_i)$, ($1 \leq i \leq k$) if $l_i$ is an atom $a_i$. 

54
The final clauses \( c_1, \ldots, c_r \) are derived by flattening the corresponding tupled clauses \( c'_1, \ldots, c'_r \). That is, \( c_i = \text{flatten}(c'_i) \) for each \( 1 \leq i \leq r \).

**Flattening Tupled Types**

**Definition 3.5.5:** Let \( \tau = (\tau_{11} \times \ldots \times \tau_{n1}) \times \ldots \times (\tau_{1k} \times \ldots \times \tau_{nk}) \) be a type. \( \text{flatten} \tau \) is defined to be \( \tau_{11} \times \ldots \times \tau_{n1} \times \ldots \times \tau_{1k} \times \ldots \times \tau_{nk} \).

The type of each predicate in the schema instance \( \Sigma \Theta \) is flattened in order to derive its final type. That is, for each \( P/p \in \Theta p \) the final type for \( p \) is derived as follows.

\[
\text{Type}(p) = \text{flatten} \text{type}(\text{Type}(p)).
\]

**Lemma 3.5.1:** Let \( \Sigma \) be a schema, \( \Theta \) a schema substitution and \( A \) a typed mode ca. Let \( c_1, \ldots, c_r \) be the flattened schema instance obtained from \( A, \Sigma \) and \( \Theta \). Then each clause \( c_i \) \( (1 \leq i \leq r) \) is a polymorphic many-sorted formula.

**Proof:** In each clause schema of \( \Sigma \) every argument variable has the same type wherever it occurs in the clause schema by definition 2.3.6. Let \( u_1, \ldots, u_m \) be the argument variables in some clause schema in \( \Sigma \) with corresponding parameter variables \( \alpha_1, \ldots, \alpha_m \) respectively. Let \( \{\alpha_1/\tau_1, \ldots, \alpha_m/\tau_m\} \subseteq \Theta_T \). Let \( \Theta_A \) be the extended argument substitution applied to \( \Sigma \). Let \( \{u_1/t_1, \ldots, u_m/t_m\} \subseteq \Theta_A \). By construction, \( t_1, \ldots, t_m \) do not share any variables, and no \( t_i \) \( (1 \leq i \leq m) \) contains repeated occurrences of a variable. Hence the type of each variable in \( t_i \) is uniquely determined by \( \tau_i \). Flattening does not affect the type of any variable.

Hence each occurrence of a variable in \( c \) has the same type. Let \( x_i \) have type \( \tau_i \) in all occurrences. Let \( \beta^i_1, \ldots, \beta^i_{k_i} \) be distinct parameters assigned to the \( k_i \) occurrences of \( x_i \). Then \( c \) is a polymorphic many-sorted formula iff

\[
\bigcup\{\beta^i_1 = \tau_i, \ldots, \beta^i_{k_i} = \tau_i\} \cup \{\beta^i_1 = \ldots = \beta^i_{k_i}\} \mid 1 \leq i \leq m
\]

has an mgu. An mgu of this set clearly exists.

A clause satisfies the **head condition** if the tuple of types of the arguments of the head of a clause is a variant of the type for the predicate in the head [36].

**Lemma 3.5.2:** Each clause in a flattened schema instance satisfies the head condition.

**Proof:** Let \( c \) be a clause in a flattened schema instance. By lemma 3.5.1 each variable in \( c \) has the same type wherever it occurs in \( c \). The arguments of the head are distinct variables.
3.5.1.4 Examples

Let \( p(x_1, x_2, x_3) \) be a typed cga whose arguments have types \( \text{seq}(	ext{str}), \alpha, \text{seq}(	ext{str}) \) respectively in atom \( p(x_1, x_2, x_3) \) and let \( \text{str} \) be the basic type string. Let \( \text{Mode}(p) = (i, i, d) \) be the expected mode of predicate \( p/3 \). \( p(x_1, x_2, x_3) \) will be refined by using the Incremental and the Search schemata in examples 1 and 2 respectively. Note that parentheses are omitted from tuples of arity one.

**Example 1:** Let assume that \( \{C_1, C_2\} \) are renamed clauses of Incremental schema so that no argument variable appears in more than one clause.

\[
C_1 : \text{Incr}(u_1^1, u_2^1) \leftarrow \\
\quad \text{Terminating}(u_1^1) \wedge \\
\quad \text{Initial.result}(u_1^1, u_2^1)
\]

\[
C_2 : \text{Incr}(u_1^2, u_2^2) \leftarrow \\
\quad \neg \text{Terminating}(u_1^2) \wedge \\
\quad \text{Deconstruction}(u_1^2, v_1^2, v_2^2) \wedge \\
\quad \text{Incr}(v_2^2, v_3^2) \wedge \\
\quad \text{Non.initial.result}(u_1^2, v_1^2, v_3^2, u_2^2)
\]

The type schemata of Incremental schema are following.

**Type Schemata**

- \( \text{Type(Incr)} : \alpha_1 \times \alpha_2 \)
- \( \text{Type(Terminating)} : \alpha_1 \)
- \( \text{Type(Initial.result)} : \alpha_1 \times \alpha_2 \)
- \( \text{Type(Deconstruction)} : \alpha_1 \times \alpha_3 \times \alpha_1 \)
- \( \text{Type(Non.initial.result)} : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2 \)

1. **Tupling:** Construction of \( \bar{x}^i, \bar{x}^d, \bar{x}^i \) and \( \bar{x}^d \).

\[
\bar{x}^i = (x_1, x_2) \\
\bar{x}^d = (x_3)
\]
\( \tau^i = (\text{seq}(\text{str}), \alpha) \)
\( \tau^d = (\text{seq}(\text{str})) \)

2. Schema refinement of a typed atom:
   a. Derivation of the components of schema substitution \( \Theta \):

Predicate substitution \( \Theta_P \).
\[ \Theta_P = \{ \text{Incr} / p, \text{Terminating} / p_1, \text{Initial_result} / p_2, \text{Deconstruction} / p_3, \text{Non_initial_result} / p_4 \} \]

Argument substitution \( \Theta_A \).
\[ \theta_1 = \{ u_1^1 / (x_1, x_2), u_2^3 / (x_3) \} \]
\[ \theta_2 = \{ u_1^2 / (x_1, x_2), u_2^2 / (x_3), v_1^2 / (y), v_2^3 / (w), v_3^2 / (z) \} \]
\[ \Theta_A' = \theta_1 \cup \theta_2 \]

Type substitution \( \Theta_T \).
\[ \Theta_T = \{ \alpha_1 / \tau^i, \alpha_2 / \tau^d \} \circ \{ \alpha_3 / (\beta) \} = \{ \alpha_1 / (\text{seq}(\text{str}), \alpha), \alpha_2 / (\text{seq}(\text{str})), \alpha_3 / (\beta) \} \]

Extending the argument substitution \( \Theta_A \).
\[ \Theta_A = \{ u_1^1 / (x_1, x_2), u_2^1 / (x_3), u_2^2 / (x_3), v_1^2 / (y), v_2^3 / (w_1, w_2), v_3^2 / (z) \} \]

c. Application of schema substitution \( \Theta \):

Derivation of clauses \( \{ c_1', c_2' \} \).
\[ c_1' = C_1 \Theta_P \Theta_A = \]
\[ p((x_1, x_2), x_3) \leftarrow p1((x_1, x_2)) \wedge \]
\[ p2((x_1, x_2), x_3) \]
\[ c_2' = C_2 \Theta_P \Theta_A = \]
\[ p((x_1, x_2), x_3) \leftarrow \neg p1((x_1, x_2)) \wedge \]
\[ p3((x_1, x_2), y, (w_1, w_2)) \wedge \]
\[ p((w_1, w_2), z) \wedge \]
\[ p4((x_1, x_2), y, z, x_3) \]

Derivation of types for each \( P/p \in \Theta_P \)
\[ \text{Type}(p) = \text{Type}(\text{Incr}) \Theta_T = (\text{seq}(\text{str}), \alpha) \times (\text{seq}(\text{str})) \]
3. Flattening:

a. Flattening tupled clauses: Construction of clauses \{c_1, c_2\} by flattening the clauses \{c'_1, c'_2\}. That is, \( r_i = flatten(r'_i) \) (1 ≤ i ≤ 2).

\[
\begin{align*}
c_1 &= p(x_1, x_2, x_3) \leftarrow \\
&\quad \quad p_1(x_1, x_2) \land \\
&\quad \quad p_2(x_1, x_2, x_3) \\
c_2 &= p(x_1, x_2, x_3) \leftarrow \\
&\quad \quad \neg p_1(x_1, x_2) \land \\
&\quad \quad p_3(x_1, x_2, y_1, w_1, w_2) \land \\
&\quad \quad p(w_1, w_2, x_1) \land \\
&\quad \quad p_4(x_1, x_2, y_1, z_1, x_3)
\end{align*}
\]

b. Flattening tupled types: For each predicate \( p/n \) in instance \( \Sigma \Theta \) its type is flattened, i.e. \( Type(p) = flatten.type(Type(p)) \).

\[
\begin{align*}
Type(p) &= seq(str) \times \alpha \times seq(str) \\
Type(p1) &= seq(str) \times \alpha \\
Type(p2) &= seq(str) \times \alpha \times seq(str) \\
Type(p3) &= seq(str) \times \alpha \times \beta \times seq(str) \times \alpha \\
Type(p4) &= seq(str) \times \alpha \times \beta \times seq(str) \times seq(str)
\end{align*}
\]

Example 2: Let assume that \{C_1, C_2, C_3, C_4\} are renamed clauses of Search schema so that no argument variable appears in more than one clause.

\[
\begin{align*}
\text{Search Schema} \\
C_1 &: Search(u_1^1, u_2^1) \leftarrow \\
&\quad Initial.state.solution(u_1^1, v_1^1, v_2^1) \land \\
&\quad Search1(u_1^1, v_1^1, v_2^1, w_2^1) \\
C_2 &: Search1(u_1^2, v_1^2, v_2^2, w_2^2) \leftarrow
\end{align*}
\]
\[\text{Search}\_\text{termination}(v_1^2, v_2^2) \land \text{Solution}\_\text{assignment}(v_2^2, u_2^2)\]

\[C_3 : \text{Search1}(u_1^3, v_1^3, v_2^3, u_2^3) \rightarrow \neg \text{Search}\_\text{termination}(v_1^3, v_2^3) \land \text{Apply}\_\text{operation}(u_1^3, v_1^3, v_2^3, v_3^3, v_4^3) \land \text{Search1}(u_1^3, v_3^3, v_4^3, u_2^3)\]

\[C_4 : \text{Search1}(u_1^4, v_1^4, v_2^4, u_2^4) \rightarrow \neg \text{Search}\_\text{termination}(v_1^4, v_2^4) \land \neg \text{Apply}\_\text{operation}(u_1^4, v_1^4, v_2^4, v_3^4, v_4^4) \land \text{Backtracking}(v_1^4, v_2^4, v_3^4, v_4^4) \land \text{Search1}(u_1^4, v_3^4, v_4^4, u_2^4)\]

The type schemata of Search schema are following.

**Type Schemata**

\[\text{Type}(\text{Search}) : \alpha_1 \times \alpha_2\]
\[\text{Type}(\text{Initial}\_\text{state}\_\text{solution}) : \alpha_1 \times \alpha_3 \times \alpha_2\]
\[\text{Type}(\text{Search1}) : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_2\]
\[\text{Type}(\text{Search}\_\text{termination}) : \alpha_3 \times \alpha_2\]
\[\text{Type}(\text{Solution}\_\text{assignment}) : \alpha_2 \times \alpha_2\]
\[\text{Type}(\text{Apply}\_\text{operation}) : \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2\]
\[\text{Type}(\text{Backtracking}) : \alpha_3 \times \alpha_2 \times \alpha_3 \times \alpha_2\]

1. **Tupling:** Construction of \(\bar{x}^i, \bar{x}^d, \bar{f}^i\) and \(\bar{f}^d\).

\[\bar{x}^i = (x_1, x_2)\]
\[\bar{x}^d = (x_3)\]
\[\bar{f}^i = (\text{seq(str)}, \alpha)\]
\[\bar{f}^d = (\text{seq(str)})\]

2. **Schema Refinement of a typed atom:**

a. **Derivation of the Components of Schema Substitution \(\Theta\):**

Predicate substitution \(\Theta_P\).

\[\Theta_P = \{\text{Search}/p, \text{Initial}\_\text{state}\_\text{solution}/p1, \text{Search1}/p2, \text{Search}\_\text{termination}/p3, \text{Solution}\_\text{assignment}/p4, \text{Apply}\_\text{operation}/p5, \text{Backtracking}/p6\}\]

Argument substitution \(\Theta'_A\).
\[ \theta_1 = \{ u_1^1/(x_1,x_2), u_2^1/(x_3), v_1^1/(w_1), v_1^1/(y_1) \} \]

\[ \theta_2 = \{ u_2^1/(x_1,x_2), v_1^1/(w_1), v_2^1/(y_1), u_3^2/(x_3) \} \]

\[ \theta_3 = \{ u_1^1/(x_1,x_2), v_2^1/(w_1), v_3^3/(y_1), w_3^3/(x_3), v_4^3/(z_1), v_4^3/(r_1) \} \]

\[ \theta_4 = \{ u_1^1/(x_1,x_2), v_1^1/(w_1), v_2^1/(y_1), w_2^2/(x_3), v_3^3/(z_2), v_4^3/(r_2), v_5^4/(t_1), v_6^6/(s_1) \} \]

\[ \Theta' = \theta_1 \cup \theta_2 \cup \theta_3 \cup \theta_4 \]

**Type substitution** \( \Theta_T \).

\[ \Theta_T = \{ \alpha_1/\bar{\tau}^i, \alpha_2/\bar{\tau}^d \} \circ \{ \alpha_3/(\beta) \} = \{ \alpha_1/(\text{seq}(\text{str}), \alpha), \alpha_2/(\text{seq}(\text{str})), \alpha_3/(\beta) \} \]

**Extending the argument substitution** \( \Theta_A \).

\[ \Theta_A = \{ u_1^1/(x_1,x_2), u_2^1/(x_3), v_1^1/(w_1), v_2^1/(y_1), w_3^3/(x_3), v_4^3/(z_1), v_4^3/(r_1) \} \]

**c. Application of schema substitution** \( \Theta \):

**Derivation of clauses** \( \{ c'_1, c'_2, c'_3, c'_4 \} \).

\[ c'_1 = C_1 \Theta_P \Theta_A = \]

\[ p((x_1,x_2), x_3) \leftarrow \]

\[ p_1((x_1,x_2), w_1, y_1) \land \]

\[ p_2((x_1,x_2), w_1, y_1, x_3) \]

\[ c'_2 = C_2 \Theta_P \Theta_A = \]

\[ p_2((x_1,x_2), w_1, y_1, x_3) \leftarrow \]

\[ p_3(w_1, y_1) \land \]

\[ p_4(y_1, x_3) \]

\[ c'_3 = C_3 \Theta_P \Theta_A = \]

\[ p_2((x_1,x_2), w_1, y_1, x_3) \leftarrow \]

\[ \neg p_3(w_1, y_1) \land \]

\[ p_5((x_1,x_2), w_1, y_1, z_1, r_1) \land \]

\[ p_2((x_1,x_2), z_1, r_1, x_3) \]

\[ c'_4 = C_4 \Theta_P \Theta_A = \]

\[ p_2((x_1,x_2), w_1, y_1, x_3) \leftarrow \]

\[ \neg p_3(w_1, y_1) \land \]

\[ \neg p_5((x_1,x_2), w_1, y_1, z_2, r_2) \land \]

\[ p_6(w_1, y_1, t_1, s_1) \land \]

\[ p_2((x_1,x_2), t_1, s_1, x_3) \]

**Derivation of types for each** \( P/p \in \Theta_P \).
3. Flattening:

a. Flattening tupled clauses: Construction of clauses \{c_1, c_2, c_3, c_4\} by flattening the clauses \{c'_1, c'_2, c'_3, c'_4\}. That is, \(c_i = \text{flatten}(c'_i)\) (\(1 \leq i \leq 4\)).

\[
c_1 = p(x_1,x_2,x_3) \leftarrow p_1(x_1,x_2,w_1,y_1) \land p_2(x_1,x_2,w_1,y_1,x_3)
\]

\[
c_2 = p_2(x_1,x_2,w_1,y_1,x_3) \leftarrow p_3(w_1,y_1) \land p_4(y_1,x_3)
\]

\[
c_3 = p_2(x_1,x_2,w_1,y_1,x_3) \leftarrow \neg p_3(w_1,y_1) \land p_5(x_1,x_2,w_1,y_1,z_1,r_1) \land p_2(x_1,x_2,z_1,r_1,x_3)
\]

\[
c_4 = p_2(x_1,x_2,w_1,y_1,x_3) \leftarrow \neg p_3(w_1,y_1) \land \neg p_5(x_1,x_2,w_1,y_1,z_2,r_2) \land p_6(w_1,y_1,t_1,s_1) \land p_2(x_1,x_2,t_1,s_1,x_3)
\]

b. Flattening tupled types: For each predicate \(p/n\) in instance \(\Sigma\) its type is flattened, i.e. \(\text{Type}(p) = \text{flatten\_type}(\text{Type}(p))\).

\[
\text{Type}(p) = \text{seq}(\text{str}) \times \alpha \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_1) = \text{seq}(\text{str}) \times \alpha \times \beta \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_2) = \text{seq}(\text{str}) \times \alpha \times \beta \times \text{seq}(\text{str}) \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_3) = \beta \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_4) = \text{seq}(\text{str}) \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_5) = \text{seq}(\text{str}) \times \alpha \times \beta \times \text{seq}(\text{str}) \times \beta \times \text{seq}(\text{str})
\]

\[
\text{Type}(p_6) = \beta \times \text{seq}(\text{str}) \times \beta \times \text{seq}(\text{str})
\]
3.5.2 Refinement by Data Type Operations or Equality

The refinement operation, refinement by a DT operation or equality, is presented in this section.

3.5.2.1 Derivation of Basic Clauses

In the following DT predicate means the predicate symbol of a DT operation.

Definition 3.5.6: Let \( p(x_1, \ldots, x_n) \) be a typed cga of predicate \( p/n \). Let \( q(y_1, \ldots, y_k) \) be a typed cga of either a DT predicate or equality predicate. Basic clauses are clauses either of the form

- \( p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k) \), \( p(x_1, \ldots, x_n) \leftarrow \neg q(y_1, \ldots, y_k) \) where \( k \leq n \), and \( \{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\} \), or
- \( p(x_1, \ldots, x_n) \leftarrow eq(y, d) \), \( p(x_1, \ldots, x_n) \leftarrow \neg eq(y, d) \) where \( y \) is a variable, \( d \) is a constant and \( y \in \{x_1, \ldots, x_n\} \).

Let \( p(x_1, \ldots, x_n) \) be a typed cga of an undefined predicate. Let \( q(y_1, \ldots, y_k) \) be a typed cga of either a DT predicate or equality where \( \{y_1, \ldots, y_k\} \cap \{x_1, \ldots, x_n\} = \emptyset \) and \( n \geq k \). Let us assume that the arguments \( z_1, \ldots, z_k \), where \( \{z_1, \ldots, z_k\} \subseteq \{x_1, \ldots, x_n\} \), are unified with \( y_1, \ldots, y_k \) respectively with substitution \( \theta \), i.e. \( \theta = \{z_1/\ y_1, \ldots, z_k/\ y_k\} \). The definition of \( p(x_1, \ldots, x_n) \) in terms of a DT operation or equality will create a basic clause as follows.

\[
\begin{align*}
p(x_1, \ldots, x_n) & \leftarrow q(y_1, \ldots, y_k) \theta \\
p(x_1, \ldots, x_n) & \leftarrow \neg q(y_1, \ldots, y_k) \theta
\end{align*}
\]

Let \( p(x_1, \ldots, x_n) \) be a typed cga of an undefined predicate. Let \( eq(y, d) \) be a typed cga of equality where \( y \) is a variable and \( d \) is a constant. Let us assume that \( p(x_1, \ldots, x_n) \) is refined by \( eq(y, d) \) where \( y \notin \{x_1, \ldots, x_n\} \) and \( n \geq 1 \). Let us assume that the argument \( z \) where \( z \in \{x_1, \ldots, x_n\} \) is unified with \( y \) with substitution \( \theta \), i.e. \( \theta = \{z/\ y\} \). The definition of \( p(x_1, \ldots, x_n) \) in terms of \( eq(y, d) \) will create a basic clause as follows.

\[
\begin{align*}
p(x_1, \ldots, x_n) & \leftarrow eq(y, d) \theta \\
p(x_1, \ldots, x_n) & \leftarrow \neg eq(y, d) \theta
\end{align*}
\]
3.5.2.2 Type Substitution in Basic Clauses

The derivation of type substitution of a basic clause is shown in this section.

Type Substitution

In order to derive the type substitution $\Theta_T$ of a basic clause we distinguish the following two cases.

1. Let us assume a basic clause of the following form.

\[
p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k) \\
p(x_1, \ldots, x_n) \leftarrow \neg q(y_1, \ldots, y_k)
\]

where $\{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\}$, $Type(p) = \tau_1 \times \ldots \times \tau_n$ and $Type(q) = \rho_1 \times \ldots \times \rho_k$. In addition, $\{\delta_1, \ldots, \delta_k\} \subseteq \{\tau_1, \ldots, \tau_n\}$ are the types of $\{z_1, \ldots, z_k\}$ respectively where $\{z_1, \ldots, z_k\} \subseteq \{x_1, \ldots, x_n\}$ and $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_k\}$. Then, the type substitution of such a clause is derived as follows.

\[
\Theta_T = mgu(\{\delta_1 = \rho_1, \ldots, \delta_k = \rho_k\})
\]

2. Let us assume a basic clause of the following form.

\[
p(x_1, \ldots, x_n) \leftarrow eq(y, d) \\
p(x_1, \ldots, x_n) \leftarrow \neg eq(y, d)
\]

where $Type(p) = \tau_1 \times \ldots \times \tau_n$, $y = x_i$ (1 $\leq i \leq n$) and $d$ is a constant of type $\rho$. Then, the type substitution of such a clause is derived as follows.

\[
\Theta_T = mgu(\tau_i, \rho)
\]

It is worth noting that each argument of a DT predicate or equality has a type in $c$ which is an instance of the corresponding declared one. That is, let $p(x_1, \ldots, x_n) \leftarrow l$ be a basic clause where $l$ is a literal. Let $q/k$ be a DT predicate or equality with declared type $Type(q) = \rho_1 \times \ldots \times \rho_k$. Let $q(y_1, \ldots, y_k)$ be the atom of $l$ where $k \leq n$. For each $y_j$ (1 $\leq j \leq k$), if $y_j = x_i$ then $\tau_i \Theta_T = \rho_j \Theta_T$. Similarly, let $eq(y, d)$ be the atom of $l$ where $d$ is a constant of type $\rho$ and $y$ is a variable. If $y = x_i$ then $\tau_i \Theta_T = \rho \Theta_T$.

Lemma 3.5.3: Let $p(x_1, \ldots, x_n)$ be a typed moded cga with type $Type(p) = \tau_1 \times \ldots \times \tau_n$ before construction of its defining clause. Let $c$ be a clause of the form
1. \( p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k) \) or \( p(x_1, \ldots, x_n) \leftarrow \neg q(y_1, \ldots, y_k) \) where \( \{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\} \) or

2. \( p(x_1, \ldots, x_n) \leftarrow eq(y, d) \) or \( p(x_1, \ldots, x_n) \leftarrow \neg eq(y, d) \) where \( y \in \{x_1, \ldots, x_n\} \).

Let us assume that \( c \) has been constructed from \( p(x_1, \ldots, x_n) \) with type substitution \( \Theta_T \).

After the construction of the clause \( Type(p) = \tau_1 \Theta_T \times \ldots \times \tau_n \Theta_T \). Let \( Type(q) = \rho_1 \times \ldots \times \rho_k \).

Then \( c \) is a polymorphic many-sorted formula satisfying the head condition.

**Proof:** \( x_1, \ldots, x_m \) are distinct variables and \( y_1, \ldots, y_k \) are distinct variables. Suppose \( y_1, \ldots, y_k \) occur in \( x_1, \ldots, x_n \) say \( x_{i_1}, \ldots, x_{i_k} \). Then by construction the type of each \( y_j \) \( (1 \leq j \leq k) \) in \( x_1, \ldots, x_n \), say \( \delta_j \) is \( \tau_j \Theta_T \), which is an instance of \( \rho_j \), its type in \( q(y_1, \ldots, y_k) \).

It follows immediately that \( c \) satisfies the head condition.

### 3.5.2.3 Examples of Basic Clause Construction

**Example 1:** Let \( p/3 \) be an undefined predicate with type \( Type(p) = seq(\alpha_1) \times seq(\alpha_2) \times seq(\alpha_3) \). Let \( p(x_1,x_2,x_3) \) be a typed cga of the undefined predicate \( p/3 \) where \( x_1,x_2,x_3 \) have types \( seq(\alpha_1), seq(\alpha_2), seq(\alpha_3) \) respectively in atom \( p(x_1,x_2,x_3) \). Let \( concat/3 \) be a DT predicate with declared type \( Type(concat) = seq(\alpha_1) \times seq(\alpha_1) \times seq(\alpha_1) \). Let \( concat(y_1,y_2,y_3) \) be a renamed typed cga so that no variable appears in atom \( p(x_1,x_2,x_3) \). \( y_1,y_2,y_3 \) have types \( seq(\alpha_1), seq(\alpha_1), seq(\alpha_1) \) respectively in atom \( concat(y_1,y_2,y_3) \). Let assume that the arguments \( x_1,x_2,x_3 \) are unified with \( y_1,y_2,y_3 \) respectively with \( \theta = \{x_1/y_1, x_2/y_2, x_3/y_3\} \). The constructed basic clause \( c \) is \( p(x_1,x_2,x_3) \leftarrow concat(y_1,y_2,y_3) \theta \), i.e. \( p(x_1,x_2,x_3) \leftarrow concat(x_1,x_2,x_3) \).

The type substitution \( \Theta_T \) of clause \( c \) is \( \Theta_T = \text{mgu}((seq(\alpha_1), seq(\alpha_2), seq(\alpha_3)), (seq(\alpha), seq(\alpha), seq(\alpha))) = \{\alpha_1/\alpha_3, \alpha_2/\alpha_3, \alpha/\alpha_3\} \).

**Example 2:** Let \( p/3 \) be an undefined predicate with type \( Type(p) = \text{int} \times \text{str} \times \alpha_1 \). Let \( p(x_1,x_2,x_3) \) be a typed cga of the undefined predicate \( p/3 \) where \( x_1,x_2,x_3 \) have types \( \text{int}, \text{str}, \alpha_1 \) respectively in atom \( p(x_1,x_2,x_3) \). Let \( string_concat/3 \) be a DT predicate with type \( Type(string_concat) = \text{str} \times \text{str} \times \text{str} \). Let \( string_concat(y_1,y_2,y_3) \) be a renamed typed cga so that no variable appears in atom \( p(x_1,x_2,x_3) \). \( y_1,y_2,y_3 \) have types \( \text{str}, \text{str}, \text{str} \) respectively in atom \( string_concat(y_1,y_2,y_3) \). Let assume that the arguments \( x_1,x_2,x_3 \) are selected for unification with \( y_1,y_2,y_3 \) respectively. The type unification, i.e. \( \text{mgu}((\text{int},\text{str},\alpha_1), (\text{str},\text{str},\text{str})) \), will fail with an error message for type inconsistency.

**Example 3:** Let \( p/3 \) be an undefined predicate with type \( Type(p) = \text{int} \times \alpha_1 \times \alpha_2 \). Let
p(x₁,x₂,x₃) be a typed cga of the undefined predicate p/3 where x₁,x₂,x₃ have types int, α₁, α₂ respectively in atom p(x₁,x₂,x₃). Let eq/2 be equality with declared type Type(eq) = α × α. Let eq(y₁,y₂) be a renamed typed cga of equality so that no variable appears in atom p(x₁,x₂,x₃). y₁,y₂ have types α, α in atom eq(y₁,y₂). Let assume that the arguments x₁,x₂ are unified with y₁,y₂ respectively with θ = {x₁/y₁, x₂/y₂}. The constructed basic clause c is p(x₁,x₂,x₃) ← eq(y₁,y₂) θ, i.e. p(x₁,x₂,x₃) ← eq(x₁,x₂).

The type substitution Θτ of clause c is Θτ = mgu((int, α₁), (α, α)) = {α₁/int, α/int}.

Example 4: Let p/2 be an undefined predicate with type Type(p) = α₁ × α₂. Let p(x₁,x₂) be a typed cga of the undefined predicate p/2 where x₁,x₂ have types α₁, α₂ respectively in atom p(x₁,x₂). Let eq/2 be equality with declared type Type(eq) = α × α. Let eq(y₁,d) be a renamed atom of equality so that no variable appears in atom p(x₁,x₂). d is a constant of type int and y₁ is a variable whose type in atom eq(y₁,d) is int. Let assume that the argument x₁ is unified with y₁ and θ = {x₁/y₁}. The constructed basic clause c is p(x₁,x₂) ← eq(y₁,d) θ, i.e. p(x₁,x₂) ← eq(x₁,d).

The type substitution Θτ of clause c is Θτ = mgu(α₁, int) = {α₁/int}.

3.6 Mode Derivation and Validation

This section presents the derivation of the expected modes of the predicates in flattened instances of schemata. In addition, it presents mode validation which is performed during construction of basic clauses in order to verify the modes of the matched arguments.

3.6.1 Derivation of Expected Modes in Flattened Clauses

Definition 3.6.1: Let Σ be a typed, moded program schema with substitution Θ. Let p(l₁,...,lₙ) be an atom in schema instance ΣΘ where P/p ∈ Θ_P. Let Mode(P) = Mode(p) = m₁,...,mₙ. flatten_mode(p(l₁,...,lₙ)) = m₁',...,mₖ' where flatten(p(l₁,...,lₙ)) is p(s₁,...,s_k), k ≥ n and m_j' = i (1 ≤ j ≤ k) if s_j was obtained from l_i (1 ≤ i ≤ n) and m_i = i otherwise m_j' = d.

Strictly speaking, we should introduce new predicate symbols. Although the same predicate symbols are used in the flattened clauses as in the original clauses, the modes of the predicates are changed. That is, Mode(p) is changed to flatten_mode(p) for any p in a flattened clause.
Examples

Let p/3 be a predicate with expected mode \( \text{Mode}(p) = (i, i, d) \). Let \( p(x_1, x_2, x_3) \) be a typed cga whose arguments \( x_1, x_2, x_3 \) have types seq(str), a, seq(str) respectively in atom \( p(x_1, x_2, x_3) \). Let us assume that \( p(x_1, x_2, x_3) \) is defined by Incremental and Search schemata. The derivation of the expected modes of the predicates which occur in the instances of Incremental and Search schemata are shown in examples 1 and 2 respectively.

Example 1: The mode schemata for Incremental schema are the following.

\[
\begin{align*}
\text{Mode Schemata} \\
\text{Mode(Incr)} : & i, d \\
\text{Mode(Terminating)} : & i \\
\text{Mode(Initial_result)} : & i, d \\
\text{Mode(Deconstruction)} : & i, d, d \\
\text{Mode(Non_initial_result)} : & i, i, i, d
\end{align*}
\]

Let us assume the example for the Incremental schema of Section 3.5.1.4. The predicate substitution of Incremental schema is \( \Theta_P = \{ \text{Incr}/p, \text{Terminating}/p_1, \text{Initial_result}/p_2, \text{Deconstruction}/p_3, \text{Non_initial_result}/p_4 \} \). For each \( P/p \in \Theta_P \) the expected mode of the predicate \( p/n \) is derived as follows.

\[
\begin{align*}
\text{Mode}(p) = & \text{flatten_mode}(p((x_1,x_2), x_3)) = i, i, d \\
\text{Mode}(p_1) = & \text{flatten_mode}(p_1((x_1,x_2))) = i, i \\
\text{Mode}(p_2) = & \text{flatten_mode}(p_2((x_1,x_2), x_3)) = i, i, d \\
\text{Mode}(p_3) = & \text{flatten_mode}(p_3((x_1,x_2), y, (w_1,w_2))) = i, i, d, d, d \\
\text{Mode}(p_4) = & \text{flatten_mode}(p_4((x_1,x_2), y, z, x_3)) = i, i, i, i, d
\end{align*}
\]

Example 2: The mode schemata for Search schema are the following.

\[
\begin{align*}
\text{Mode Schemata} \\
\text{Mode(Search)} : & i, d \\
\text{Mode(Initial_state_solution)} : & i, d, d \\
\text{Mode(Search1)} : & i, i, d \\
\text{Mode(Search_termination)} : & i, i \\
\text{Mode(Solution_assignment)} : & i, d \\
\text{Mode(Apply_operation)} : & i, i, i, d, d \\
\text{Mode(Backtracking)} : & i, i, d, d
\end{align*}
\]
Let us assume the example for the Search schema of Section 3.5.1.4. The predicate substitution of Search schema is $\Theta_P = \{\text{Search}/p, \text{Initial\_state\_solution}/p_1, \text{Search1}/p_2, \text{Search\_termination}/p_3, \text{Solution\_assignment}/p_4, \text{Apply\_operation}/p_5, \text{Backtracking}/p_6\}$. For each $P/p \in \Theta_P$ the expected mode of the predicate $p/n$ is derived as follows.

\[
\begin{align*}
\text{Mode}(p) &= \text{flatten\_mode}(p((x_1, x_2), x_3)) = i, i, d \\
\text{Mode}(p_1) &= \text{flatten\_mode}(p_1((x_1, x_2), w_1, y_1)) = i, i, d, d \\
\text{Mode}(p_2) &= \text{flatten\_mode}(p_2((x_1, x_2), w_1, y_1, x_3)) = i, i, i, d \\
\text{Mode}(p_3) &= \text{flatten\_mode}(p_3(w_1, y_1)) = i, i \\
\text{Mode}(p_4) &= \text{flatten\_mode}(p_4(y_1, x_3)) = i, d \\
\text{Mode}(p_5) &= \text{flatten\_mode}(p_5((x_1, x_2), w_1, y_1, z_1, r_1)) = i, i, i, d, d \\
\text{Mode}(p_6) &= \text{flatten\_mode}(p_6(w_1, y_1, t_1, s_1)) = i, i, d, d
\end{align*}
\]

3.6.2 Mode Validation in Basic Clauses

A mode inference procedure has been implemented in this thesis which will be presented in details in Chapter 4. In this section we just summarize its use during program construction.

The mode inference procedure 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 predicates are used consistently with their declared call modes. Note that the mode inference procedure assumes a left-right computation rule, and hence gives useful results only when the refinements are carried out in a left-right order as well. The DT predicates are assumed to succeed with all their arguments grounded.

The inferred mode of a predicate $p/n$, denoted by $\text{Mode}(p)$, stands for the mode of $p/n$ which is generated by the mode inference procedure during program construction.

**Definition 3.6.2:** Let us assume that $m_1, \ldots, m_n$ and $m'_1, \ldots, m'_n$ are two modes of predicate $p/n$. The mode $m'_1, \ldots, m'_n$ is *subsumed* by the mode $m_1, \ldots, m_n$ iff either $m'_i = m_i$ or $m'_i = i$ and $m_i = d$ ($1 \leq i \leq n$).

Let $p/n$ be a DT predicate or equality. The declared mode of $p/n$, denoted by $\text{Mode}(p)$, is specified by the programmer.

Let us assume that a basic clause $c$ has been constructed by the method of this thesis where $c : p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k)$ or $c : p(x_1, \ldots, x_n) \leftarrow \neg q(y_1, \ldots, y_k)$ or $c : p(x_1, \ldots, x_n) \leftarrow eq(y, d)$ or $c : p(x_1, \ldots, x_n) \leftarrow \neg eq(y, d)$. The inferred call mode of $q/k$ or $eq/2$ has to be
subsumed by its declared one.

3.7 The Construction Process

This section presents the construction process as 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 3.7.1: A refinement step or simply a refinement is defined to be either of the following actions.

1. The definition of an undefined predicate by instantiating a design schema. Such refinements are called schema refinements. A schema refinement consists of three refinement operations, namely tupling, schema refinement of a typed atom and flattening. In addition, it involves the derivation of the expected modes of the predicates in the instances of the schemata.

2. 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 3.7.2: Let \( \text{Prog} \) be a normal logic program constructed by this method. Its set of clauses is \( S_1 \cup S_2 \cup S_3 \), where \( S_1 \) consists of flattened instances of schemata, \( S_2 \) consists of basic clauses and \( S_3 \) consists of definitions of DT operations and equality. We call \( \text{Prog} \) a Schema-Instance program or SI-program.

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 \( \tau_i \) 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 \( \tau_i \) of each \( x_i \) in \( p(x_1, \ldots, x_n) \) 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. Next, this process is defined in more detail.
3.7.1 Refinement Trees

**Definition 3.7.3:** A *refinement tree* is defined as follows. Every *non-leaf node* is labeled by

1. a typed, moded cga of some predicate,
2. a schema Σ standardized apart from any other schema in the refinement tree,
3. a schema substitution Θ for Σ, with components Θ₀, Θ₁, and Θ₂.

The type and mode of the predicate at the root is given by the programmer. Every non-leaf node N, labelled with predicate p, schema Σ and schema substitution Θ, has child node M iff M is labelled by q and

a. \( Q/q \in Θ₀,

b. \( \text{Type}(q) = \text{flatten type}(\text{Type}(Q)Θ₁ \ldots Θ_k), \) where \( Θ₁ \ldots Θ_k \) is the sequence of schema substitutions labelling the sequence of nodes preceding M in an inorder left-right traversal of the refinement tree,

c. \( \text{Mode}(q) \) is derived as shown in Definition 3.6.1.

A leaf has either of the following forms.

1. It is labeled by a typed, moded cga of an unrefined predicate.
2. It is labeled by
   a. a typed, moded cga of a predicate,
   b. a typed, moded cga of a DT predicate or equality,
   c. an injective mapping of the variables of the DT predicate or equality to the variables of the predicate in a.

A *refinement tree* is *complete* if it does not have any leaf of the first kind.

The program construction process is represented by a refinement tree. Each node of a complete refinement tree represents either a schema refinement or a DT refinement. The construction of a program corresponds to an inorder, left-right traversal of the refinement tree.
Definition 3.7.4: Let \( N_1, N_2, \ldots, N_k \) be the sequence of nodes in an inorder left-right traversal of the refinement tree. Let \( \Theta_1^T, \Theta_2^T, \ldots, \Theta_k^T \) be the corresponding type substitutions. It is assumed that the empty substitution is associated with leaf nodes for unrefined predicates. The accumulated type substitution at node \( N_i \), denoted by \( \Psi_i^T \), is defined as follows.

1. Node \( N_1 \) (root node): \( \Psi_1^T = \varepsilon \).
2. Node \( N_i \) where \( i > 1 \): \( \Psi_i^T = \Theta_i^T \circ \cdots \circ \Theta_1^T \).

The global type substitution for a refinement tree is \( \Theta_1^T \circ \cdots \circ \Theta_k^T \).

Definition 3.7.5: The global predicate substitution for a refinement tree, denoted by \( \Psi_P \), is defined as follows.

\[
\Psi_P = \bigcup \{ \Theta_P^i \mid \Theta_P^i \text{ is a predicate substitution in the refinement tree} \}
\]

It is assumed that the empty predicate substitution is associated with each leaf node.

3.7.2 Signatures and Modes of the Predicates of an SI-Program

In an SI-program, the types and modes of DT and equality predicates are declared by the programmer. The types and modes of all other predicates are derived during program construction.

Each SI-program is associated with its refinement tree which has a global type substitution \( \Psi_T \) and a global predicate substitution \( \Psi_P \). Let \( \text{Prog} \) be an SI-program. Let \( \Psi_T \) and \( \Psi_P \) be the global type substitution and the global predicate substitution respectively for the refinement tree of \( \text{Prog} \). Let us assume that \( P/p \in \Psi_P \). The type of \( p \) is derived as follows.

\[
\text{Type}(p) = \text{flatten\_type}(\text{Type}(P)\Psi_T).
\]

The call modes of the predicates of an SI-program are derived, apart from those of DT operations and equality, by the mode inference procedure applied to the constructed SI-program (see Chapter 4).
3.7.3 An Example

The above discussion is illustrated by an example. Let \( \text{sum}/2 \) be an undefined predicate with \( \text{Type}(	ext{sum}) = \text{seq}(\alpha_1) \times \alpha_1 \) and expected mode \( \text{Mode}(\text{sum}) = (i, d) \). Let us assume that a programmer wants to construct a program for \( \text{sum}/2 \). Let \( \text{sum}(x_1, x_2) \) be a typed cga. The predicate \( \text{sum}(x_1, x_2) \) is true iff \( x_2 \) is the sum of integers in sequence \( x_1 \).

The construction of the program and the objects in the refinement tree for this example are presented in this section. The refinement tree is shown in Figure 3.1. For brevity, the full schemata and schema substitutions are not shown in the figure but are abbreviated by giving the schema name and the DT operation. Note that each node of the refinement tree can be uniquely identified by the name of the predicate.

**Constructed SI-Program.**

The constructed SI-program is as follows.

*Signatures*

- tail: \( \text{seq}(\alpha_1) \times \text{seq}(\alpha_1) \)
- neutral_add_subtr_int: int
- empty_seq: \( \text{seq}(\alpha_1) \)
- head: \( \text{seq}(\alpha_1) \times \alpha_1 \)
- plus_int: \( \text{int} \times \text{int} \times \text{int} \)
- sum: \( \text{seq}(\text{int}) \times \text{int} \)
- p0: \( \text{seq}(\text{int}) \)
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: i, d
neutral_add_subtr_int: d
empty_seq: d
head: i, d
plus_int: i, i, d
sum: i, d
p0: i
p1: i, d
p4: i, d, d
p2: i, d, d
p3: i, i, i, d
p5: i, i, d

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).
\[ p_4(x_1, x_3, x_4) \rightarrow \text{head}(x_1, x_3) \]

6. Refinement 6: The DT predicate tail/2 refines \( p_5(x_1, x_3, x_4) \).
\[ p_5(x_1, x_3, x_4) \leftarrow \text{tail}(x_1, x_4) \]

7. Refinement 7: The DT predicate plus_int/2 refines \( p_3(x_1, x_3, x_5, x_2) \).
\[ p_3(x_1, x_3, x_5, x_2) \leftarrow \text{plus_int}(x_3, x_5, x_2) \]

Definitions of DT and Equality Predicates

- \( \text{empty_seq}([]) \)
- \( \text{neutral_add_subtr_int}(0) \)
- \( \text{head}([h \mid t], h) \)
- \( \text{tail}([h \mid t], t) \)
- \( \text{plus_int}(x_1, x_2, x_3) \leftarrow x_3 \text{ is } x_1 + x_2 \)

Derivation of Signatures.

The first step in the derivation of the signatures is to compute the global type substitution \( \Psi_T \) and the global predicate substitution \( \Psi_P \) for this refinement tree. Then the signatures of the predicates of the SI-program are derived by using \( \Psi_T \) and \( \Psi_P \).

Renamed type schemata of Incremental schema in the node for predicate sum/2.
- \( \text{Type}(\text{Incr}) : \alpha_1 \times \alpha_2 \)
- \( \text{Type}(\text{Terminating}) : \alpha_1 \)
- \( \text{Type}(\text{Initial.result}) : \alpha_1 \times \alpha_2 \)
- \( \text{Type}(\text{Deconstruction}) : \alpha_1 \times \alpha_3 \times \alpha_4 \)
- \( \text{Type}(\text{Non.initial.result}) : \alpha_1 \times \alpha_3 \times \alpha_4 \times \alpha_2 \)

Renamed type schemata of Subgoal schema in the node for predicate r0/3.
- \( \text{Type}(\text{SubGoal}) : \alpha_1^2 \times \alpha_2^2 \)
- \( \text{Type}(\text{SubGoal1}) : \alpha_1^2 \times \alpha_3^2 \times \alpha_2^2 \)
- \( \text{Type}(\text{SubGoal2}) : \alpha_1^2 \times \alpha_3^2 \times \alpha_2^2 \)

The type substitutions \( \Theta_T^i \ (1 \leq i \leq 7) \) of the nodes of the refinement tree which are visited during the traversal are as follows.
Node for the predicate sum/2.

\[ \Theta^f_1 = \{a_1/(\text{seq}(a_1)), a_1/(\text{int}), a_2/(\beta)\} \]

Node for the predicate p0/1.

\[ \Theta^f_2 = \{a_4/(\text{int})\} \]

Node for the predicate q0/2.

\[ \Theta^f_3 = \{a_1/(\text{int})\} \]

Node for the predicate r0/3.

\[ \Theta^f_4 = \{a_1/(\text{seq}(\text{int})), a_2/(\beta, \text{seq}(\text{int})), a_3/(\beta)\} \]

Node for the predicate p1/3.

\[ \Theta^f_5 = \{a_5/(\text{int}), \beta/(\text{int})\} \]

Node for the predicate q1/3.

\[ \Theta^f_6 = \{a_6/(\text{int})\} \]

Node for the predicate s0/4.

\[ \Theta^f_7 = \emptyset \]

Global type substitution \( \Psi_T = \Theta_T^f \circ \Theta_T^f \circ \Theta_T^f \circ \Theta_T^f \circ \Theta_T^f \circ \Theta_T^f \).

\[ \Psi_T = \{a_1/(\text{seq}(\text{int})), a_2/(\text{int}), a_3/(\text{int}), a_4/(\text{int}), a_5/(\text{int}), a_6/(\text{int}), \beta/(\text{int}), \alpha/(\text{int})\} \]

Predicate substitutions for each non-leaf node of the refinement tree.

\[ \Theta^f_p = \{\text{Incr}/\text{sum}, \text{Terminating}/p0, \text{Initial.result}/p1, \text{Deconstruction}/p2, \text{Non.initial.result}/p3\} \]

\[ \Theta^f_p = \{\text{SubGoal}/p2, \text{SubGoal1}/p4, \text{SubGoal2}/p5\} \]

Global predicate substitution \( \Psi_P = \Theta_P^f \cup \Theta_P^f \).

\[ \Psi_P = \{\text{Incr}/\text{sum}, \text{Terminating}/p0, \text{Initial.result}/p1, \text{Deconstruction}/p2, \text{Non.initial.result}/p3, \text{SubGoal}/p2, \text{SubGoal1}/p4, \text{SubGoal2}/p5\} \]

Signatures

\[ \text{Type}(\text{sum}) = \text{flatten.type}(\text{Type}(\text{Incr}) \Psi_T) = \text{flatten.type}(a_1 \times a_2 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \]

\[ \text{Type}(p0) = \text{flatten.type}(\text{Type}(\text{Terminating}) \Psi_T) = \text{flatten.type}(a_1 \Psi_T) = \text{seq}(\text{int}) \]

\[ \text{Type}(p1) = \text{flatten.type}(\text{Type}(\text{Initial.result}) \Psi_T) = \text{flatten.type}(a_1 \times a_2 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \]

\[ \text{Type}(p2) = \text{flatten.type}(\text{Type}(\text{Deconstruction}) \Psi_T) = \text{flatten.type}(a_1 \times a_2 \times a_1 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \times \text{seq}(\text{int}) \]

\[ \text{Type}(p3) = \text{flatten.type}(\text{Type}(\text{Non.initial.result}) \Psi_T) = \text{flatten.type}(a_1 \times a_2 \times a_2 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \times \text{int} \times \text{int} \]

\[ \text{Type}(p4) = \text{flatten.type}(\text{Type}(\text{SubGoal1}) \Psi_T) = \text{flatten.type}(a_2 \times a_3 \times a_2 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \times \text{seq}(\text{int}) \]

\[ \text{Type}(p5) = \text{flatten.type}(\text{Type}(\text{SubGoal2}) \Psi_T) = \text{flatten.type}(a_2 \times a_3 \times a_2 \Psi_T) = \text{seq}(\text{int}) \times \text{int} \times \text{seq}(\text{int}) \]
Note that the signatures of the DT operations are declared by the user.

Components Associated with the Nodes of the Refinement Tree.

The labels associated with each node of the refinement tree are as follows. Note that the types of predicates in the nodes of the refinement tree are the types they have before the corresponding refinement is applied. The type substitution of basic clauses is shown in order to clarify the refinements on leaf nodes.

1. **Node with typed cga for predicate sum/1.** The typed cga of the predicate of this node is $\text{sum}(x_1, x_2)$, $\text{Type}(\text{sum}) = \text{seq}(\alpha_1) \times \alpha_1$ and $\text{Mode}(\text{sum}) = i, d$.

   Renamed schema clauses
   
   $$\text{Incr}(u_1^1, u_2^1) \leftarrow \text{Terminating}(u_1^1) \land \text{Initial\_result}(u_1^1, u_2^1)$$
   $$\text{Incr}(u_1^2, u_2^2) \leftarrow \neg\text{Terminating}(u_1^2) \land \text{Deconstruction}(u_1^2, v_1^1, v_2^1) \land \text{Incr}(v_1^2, v_2^2) \land \text{Non\_initial\_result}(u_1^2, v_1^1, v_2^1, u_2^2)$$

   Renamed type schemata

   $$\text{Type}(\text{Incr}) : \alpha_1^1 \times \alpha_2^1$$
   $$\text{Type}(\text{Terminating}) : \alpha_1^1$$
   $$\text{Type}(\text{Initial\_result}) : \alpha_1^1 \times \alpha_2^1$$
   $$\text{Type}(\text{Deconstruction}) : \alpha_1^1 \times \alpha_1^2 \times \alpha_1^1$$
   $$\text{Type}(\text{Non\_initial\_result}) : \alpha_1^1 \times \alpha_1^2 \times \alpha_2^1 \times \alpha_2^1$$

   Predicate substitution.

   $$\Theta_P = \{\text{Incr}/\text{sum}, \text{Terminating}/p0, \text{Initial\_result}/q0, \text{Deconstruction}/r0, \text{Non\_initial\_result}/s0\}.$$  

   Argument substitution.

   $$\Theta_A = \{u_1^1/(x1), u_2^1/(x2), u_1^2/(x1), u_2^2/(x2), v_1^1/(x3), v_2^2/(x4), v_2^2/(x5)\}$$

   Type substitution.
\[ \Theta_T^1 = \{ \alpha_1^1/(\text{seq}(\alpha_1)), \alpha_2^1/(\alpha_1), \alpha_3^1/(\beta) \} \]

Accumulated type substitution.
\[ \Psi_T^1 = \emptyset \]

2. **Node with typed cga for predicate p0/1.** The typed cga of the undefined predicate of this node is \( p0(x_1) \), \( Type(p0) = \text{seq}(\alpha_1) \). The typed cga of the DT predicate empty_seq/1 is empty_seq(y1), \( Type(\text{empty_seq}) = \text{seq}(\alpha_4) \).

Mapping of variables.
\[ \theta = \{ y_1/x_1 \} \]
Accumulated type substitution.
\[ \Psi_T^2 = \Theta_T^1 \circ \Theta_T^1 = \{ \alpha_1^1/(\text{seq}(\alpha_1)), \alpha_2^1/(\alpha_1), \alpha_3^1/(\beta) \} \]
Type substitution.
\[ \Theta_T^3 = \{ \alpha_4/\alpha_1 \} \]

3. **Node with typed cga for predicate p1/2.** The typed cga of the predicate of this node is \( p1(x_1,x_2) \). \( Type(p1) = \text{seq}(\alpha_1) \times \alpha_1 \). The typed cga of the DT predicate neutral_add_subtr_int/1 is neutral_add_subtr_int(y1), \( Type(\text{neutral_add_subtr_int}) = \text{seq}(\text{int}) \).

Mapping of variables.
\[ \theta = \{ y_1/x_2 \} \]
Accumulated type substitution.
\[ \Psi_T^3 = \Theta_T^1 \circ \Theta_T^2 = \{ \alpha_1^1/(\text{seq}(\alpha_1)), \alpha_2^1/(\alpha_1), \alpha_2^1/\text{int} \} \]
Type substitution.
\[ \Theta_T^4 = \{ \alpha_1/\text{int} \} \]

4. **Node with typed cga for predicate p2/3.** The typed cga of the predicate of this node is \( p2(x_1,x_3,x_4) \). \( Type(p2) = \text{seq}(\text{int}) \times \beta \times \text{seq}(\text{int}) \) and \( Mode(p2) = i, d, d \).

Renamed schema clauses
\[ Subgoal(u_1^2,u_2^2) \leftarrow \]
\[ Subgoal1(u_1^2,u_2^2) \wedge \]
\[ Subgoal2(u_1^2,u_2^2) \]

Renamed type schemata
\[ Type(SubGoal) = \alpha_1^1 \times \alpha_2^1 \]
\[ Type(SubGoal1) = \alpha_1^2 \times \alpha_2^2 \times \alpha_2^2 \]
\[ Type(SubGoal2) = \alpha_1^3 \times \alpha_2^3 \times \alpha_2^2 \]

Predicate substitution.
\[\Theta_\gamma^4 = \{Subgoal/p2, SubGoal1/p4, SubGoal2/p5\}\]

Argument substitution.
\[\Theta_\gamma^4 = \{u_1^2/(x1), u_2^2/(x3,x4), v^2/(())\}\]

Type substitution.
\[\Theta_\gamma^4 = \{\alpha_1^2/(\text{seq}(\text{int})), \alpha_2^2/(\beta,\text{seq}(\text{int})), \alpha_3^2/()\}\]

Accumulated type substitution.
\[\Psi_\gamma^4 = \Theta_\gamma^4 \circ \Theta_\gamma^4 = \{\alpha_1^2/(\text{seq}(\text{int})), \alpha_2^2/(\text{int}), \alpha_3^2/(\beta), \alpha_4/\text{int}, \alpha_1/\text{int}\}\]

5. Node with typed cga for predicate p4/3. The typed cga of the undefined predicate of this node is p4(x1,x3,x4), Type(p4) = seq(int) × β × seq(int). The typed cga of the DT predicate head/2 is head(y1,y2), Type(head) = seq(α5) × seq(α5).

Mapping of variables.
\[\theta = \{y1/x1,y2/x3\}\]

Accumulated type substitution.
\[\Psi_\gamma^5 = \Theta_\gamma^5 \circ \Theta_\gamma^4 = \{\alpha_1^2/(\text{seq}(\text{int})), \alpha_2^2/(\text{int}), \alpha_3^2/(\beta), \alpha_4/\text{int}, \alpha_1/\text{int}\}\]

Type substitution.
\[\Theta_\gamma^5 = \{\alpha_5/\text{int}, \beta/\text{int}\}\]

6. Node with typed cga for predicate p5/3. The typed cga of the predicate of this node is p5(x1,x3,x4), Type(p5) = seq(int) × int × seq(int). The typed cga of the DT predicate tail/2 is tail(y1,y2), Type(tail) = seq(α5) × seq(α5).

Mapping of variables.
\[\theta = \{y1/x1,y2/x3\}\]

Accumulated type substitution.
\[\Psi_\gamma^6 = \Theta_\gamma^6 \circ \Theta_\gamma^5 \circ \Theta_\gamma^4 = \{\alpha_1^2/(\text{seq}(\text{int})), \alpha_2^2/(\text{int}), \alpha_3^2/(\beta), \alpha_4/\text{int}, \alpha_1/\text{int}\}\]

Type substitution.
\[\Theta_\gamma^6 = \{\alpha_6/\text{int}\}\]

7. Node with typed cga for predicate p3/4. The typed cga of the predicate of this node is p3(x1,x3,x5,x2), Type(p3) = seq(int) × int × int × int. The typed cga of the DT predicate plus_int/3 is plus_int(y1,y2,y3), Type(plus_int) = int × int × int × int.

Mapping of variables.
\[\theta = \{y1/x3,y2/x5,y3/x2\}\]

Accumulated type substitution.
\[\Psi_\gamma^7 = \Theta_\gamma^7 \circ \Theta_\gamma^6 \circ \Theta_\gamma^5 \circ \Theta_\gamma^4 = \{\alpha_1^2/(\text{seq}(\text{int})), \alpha_2^2/(\text{int}), \alpha_3^2/(\beta), \alpha_4/\text{int}, \alpha_1/\text{int}\}\]

Type substitution.
\[\Theta_\gamma^7 = \emptyset\]
3.8 Properties of SI-Programs

The properties which are satisfied by the SI-programs are shown in this section. That is, SI-programs are polymorphic many-sorted programs. SI-programs do not require run-time type checking. In addition, SI-programs need no run time mode checking for DT predicates, but this will be discussed in Chapter 4.

3.8.1 SI-Programs are Polymorphic Many-Sorted Programs

SI-programs are required to be polymorphic many-sorted formulas. It will be now proved that this is the case. In the following we assume the definitions of polymorphic many-sorted term, atom and formula from [36].

**Proposition 3.8.1:** Let \( \text{Prog} = S_1 \cup S_2 \cup S_3 \) be the clauses of an SI-program where \( S_1 \) are flattened instances of schemata, \( S_2 \) are basic clauses and \( S_3 \) are the definitions of DT operations and equality. Let us assume that clauses \( S_3 \) are polymorphic many-sorted formulas. Then \( \text{Prog} \) is a polymorphic many-sorted formula.

**Proof:** By lemmas 3.5.1 and 3.5.3 clauses in \( S_1 \) and \( S_2 \) are polymorphic many-sorted formulas immediately after their construction. It remains to prove that they are polymorphic many-sorted formulas at the end of the construction process, when the global type substitution is applied.

1. **Clauses in flattened schema instances (\( S_1 \)):** As shown in lemma 3.5.1, the fact that they are polymorphic many-sorted formulas follows solely from the fact that each variable in the clause has the same type. Hence any further instantiation of the type does not create a formula which is not polymorphic many-sorted.

2. **Basic clauses (\( S_2 \)):** Let us assume the basic clause \( p(x_1, \ldots, x_n) \leftarrow I \) where \( I \) is a literal and \( x_1, \ldots, x_n \) are distinct variables. By lemma 3.5.3, basic clauses are polymorphic many-sorted formulas immediately after their construction, since at that point each head variable either does not occur in \( I \), or has a type which is an instance of its type in \( I \). Further instantiating the type of \( p \) cannot affect that property. Hence it is a polymorphic many-sorted formula under the global type substitution.
3.8.2 SI-Programs Require No Run-Time Type Check

A desirable property of typed logic languages is that goals can be run without run-time type checking. Let $\mathcal{L}$ be a typed logic language whose programs and goal are assumed to be polymorphic many-sorted. Let $G$ be a goal in $\mathcal{L}$ and $\text{Prog}$ be a program in $\mathcal{L}$. The head condition [69] and the transparency condition have to be satisfied by the type system of $\mathcal{L}$ in order to able to run $G$ under untyped SLDNF-resolution without run-time type checking [36], [37].

1. A clause satisfies the head condition if the tuple of types of the arguments of the head of a clause is a variant of the declared type for the predicate in the head.

2. A type declaration for a function is transparent if every parameter which appears in the declared domain type of the function appears also in its declared range type.

Let $\mathcal{L}$ be the polymorphic many-sorted language of this method. Let $G$ be a well-typed goal in $\mathcal{L}$, i.e. $G$ is a conjunction of polymorphic many-sorted literals in $\mathcal{L}$.

**Proposition 3.8.2**: Let $\text{Prog}$ be an SI-program with clauses $S_1 \cup S_2 \cup S_3$ where $S_1$ are flattened instances of schemata, $S_2$ are basic clauses and $S_3$ are definitions of DT operations and equality. Let us assume that clauses $S_3$ satisfy the head condition. Then all clauses in $\text{Prog}$ satisfy the head condition.

**Proof**:

1. **Clauses in flattened schema instances ($S_1$)**: Let $c = p(x_1, \ldots, x_n) \leftarrow l_1, \ldots, l_k$ be a clause in a flattened schema instance where $l_i (1 \leq i \leq k)$ are literals and $x_1, \ldots, x_n$ are distinct variables. By lemma 3.5.2 $c$ satisfies immediately the head condition after refinement. $c$ has the property that every occurrence of a variable in $c$ has the same argument type. Suppose $\text{Type}(p) = \tau_1 \times \ldots \times \tau_n$ immediately after refinement. Then $x_1, \ldots, x_n$ also have types $\tau_1, \ldots, \tau_n$ in the body. Hence any further instantiation of $\tau_1, \ldots, \tau_n$ during program construction applies both in the head and the body, and hence $c$ continues to satisfy the head condition.

2. **Basic clauses ($S_2$)**: Let $c$ be a clause of the form

   (a) $p(x_1, \ldots, x_n) \leftarrow q(y_1, \ldots, y_k)$ or $p(x_1, \ldots, x_n) \leftarrow \neg q(y_1, \ldots, y_k)$ where $\{y_1, \ldots, y_k\} \subseteq \{x_1, \ldots, x_n\}$

   (b) $p(x_1, \ldots, x_n) \leftarrow eq(y, d)$ or $p(x_1, \ldots, x_n) \leftarrow \neg eq(y, d)$ where $y \in \{x_1, \ldots, x_n\}$, since $x_1, \ldots, x_n$ are distinct variables.
By lemma 3.5.3 these satisfy the head condition immediately after refinement. The types of \( q \) and \( eq \) remain unchanged during program construction. The type of \( p \) may become further instantiated, but this does not affect the head condition.

The clauses defining DT and equality predicates by assumption satisfy the head condition and any function symbols in them are assumed to satisfy the transparency condition. The only non-variables in \( S_1 \) and \( S_2 \) are constants, whose types trivially satisfy the transparency condition. Hence, SI-programs require no run-time type checking, except for the goal.

3.9 Data 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 \( \mathbb{Z} \) (integers), \( \mathbb{N} \) (naturals), \( \mathbb{Q} \) (rationals) and strings. The operations of these DTs have been implemented as predicates. A programmer can implement his own DTs by using this method.

3.9.1 Built-in Data Types

A complete example of the built-in DT for sequences is now given. The following notation is used in the specifications of the operations of sequences. \( <> \) stands for the empty sequence. \( :: \) stands for the term constructor for sequences. \( x :: q \) is a sequence with head \( x \) and tail \( q \). Its relational form is \( \text{seq-cons}(q, x, r) \).

Example 1: The specification of the data type \( \text{seq}(r) \) is as follows.

Signatures

\[
\begin{align*}
\text{empty_seq} & : \text{seq}(\alpha) \\
\text{head} & : \text{seq}(\alpha) \times \alpha \\
\text{tail} & : \text{seq}(\alpha) \times \text{seq}(\alpha) \\
\text{seq_cons} & : \text{seq}(\alpha) \times \alpha \times \text{seq}(\alpha) \\
\text{front} & : \text{seq}(\alpha) \times \text{seq}(\alpha) \\
\text{last} & : \text{seq}(\alpha) \times \alpha \\
\text{length_seq} & : \text{seq}(\alpha) \times \mathbb{N} \\
\text{seq_nth_elem} & : \text{seq}(\alpha) \times \mathbb{N} \times \alpha
\end{align*}
\]
elem_occs_seq: \(\text{seq}(\alpha) \times \alpha \times N\)
concat: \(\text{seq}(\alpha) \times \text{seq}(\alpha) \times \text{seq}(\alpha)\)
reverse: \(\text{seq}(\alpha) \times \text{seq}(\alpha)\)

Modes
empty_seq: \(d\)
head: \(i, d\)
tail: \(i, d\)
seq_cons: \(i, i, d\)
front: \(i, d\)
last: \(i, d\)
length_seq: \(i, d\)
seq_nth_elem: \(i, i, d\)
elem_occs_seq: \(1, 1, d\)
concat: \(i, i, d\)
reverse: \(i, d\)

FOL specifications of the DT operations.
\[
\forall q/\text{seq}(\alpha) (\text{empty_seq}(q) \leftrightarrow q =<>)
\]
\[
\forall q/\text{seq}(\alpha), h/\alpha (\text{head}(q, h) \leftrightarrow (q \neq <> \land \exists t/\text{seq}(\alpha) q = h :: t))
\]
\[
\forall q/\text{seq}(\alpha), t/\text{seq}(\alpha) (\text{tail}(q, t) \leftrightarrow (q \neq <> \land \exists h/\alpha q = h :: t))
\]
\[
\forall q/\text{seq}(\alpha), x/\alpha, r/\text{seq}(\alpha) (\text{seq_cons}(q, x, r) \leftrightarrow (r \neq <> \land \text{head}(r, x) \land \text{tail}(r, q)))
\]
\[
\forall q/\text{seq}(\alpha), n/N (\text{length_seq}(q, n) \leftrightarrow ((q =<> \land n = 0) \lor
\exists h/\alpha, t/\text{seq}(\alpha), n1/N (q = h :: t \land \text{length_seq}(t, n1) \land n = n1 + 1)))
\]
\[
\forall q/\text{seq}(\alpha), n/N, x/\alpha (\text{seq_nth_elem}(q, n, x) \leftrightarrow
(n > 0 \land \exists m/N (\text{length_seq}(q, m) \land n \leq m)) \land
\exists h/\alpha, t/\text{seq}(\alpha) (q = h :: t \land (n = 1 \land x = h) \lor
(n > 1 \land \exists n1/N (n1 = n - 1 \land \text{seq_nth_elem}(t, n1, x))))))
\]
\[
\forall q/\text{seq}(\alpha), r/\text{seq}(\alpha) (\text{front}(q, r) \leftrightarrow (q \neq <> \land
\exists n1, n2/N (\text{length_seq}(r, n1) \land \text{length_seq}(q, n2) \land n1 = n2 - 1) \land
\forall i/N (1 \leq i \leq n1 \rightarrow (\text{seq_nth_elem}(t, i, x) \leftrightarrow \text{seq_nth_elem}(q, i, x)))))
\]
\[
\forall q/\text{seq}(\alpha), x/\alpha (\text{last}(q, x) \leftrightarrow
\exists n/N (q \neq <> \land \text{length_seq}(q, n) \land \text{seq_nth_elem}(q, n, x)))
\]
\[
\forall q/\text{seq}(\alpha), x/\alpha, n/N, (\text{elem_occs_seq}(q, x, n) \leftrightarrow
\exists h/\alpha, t/\text{seq}(\alpha), n1/N (q = h :: t \land \text{elem_occs_seq}(t, x, n1) \land
((x = h \land n = n1 + 1) \lor (x \neq h \land n = n1))))
\]
\[
\forall q/\text{seq}(\alpha), q2/\text{seq}(\alpha), r/\text{seq}(\alpha) (\text{concat}(q1, q2, r) \leftrightarrow
((q1 =<> \land r = q2) \lor \exists h/\alpha, t/\text{seq}(\alpha) (q1 = h :: t \land
\exists s/\text{seq}(\alpha) (\text{concat}(t, q2, s) \land r = h :: s))))
\]
\[
\forall q/\text{seq}(\alpha), r/\text{seq}(\alpha) (\text{reverse}(q, r) \leftrightarrow ((q =<> \land r =<>) \lor
\exists h/\alpha, t/\text{seq}(\alpha) (q1 = h :: t \land
\exists s/\text{seq}(\alpha) (\text{concat}(t, q2, s) \land r = h :: s))))
\]
3.9.2 User-defined Data Types

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. Let us assume that the user wants to define a DT $\tau$. Some built-in DTs or previously defined user-defined DTs are selected for the representation of $\tau$. Its operations are implemented using this method and the underlying DTs. The specifications of the operations of $\tau$ are expressed in the theory of the underlying DTs.

Note that the user cannot introduce new function and constant symbols, but uses the symbols available in the built-in DTs. The reasons for this are twofold. Firstly, the program construction method is strongly oriented to building more complex programs and objects on top of well-defined simpler components. The available built-in types provide a sufficiently rich set of components on which to build more complex types. This encourages a more systematic approach to construction. Secondly, in order to implement DTs containing new functions symbols, clauses containing functions in their heads, or equality atoms containing compound terms, would be required. Neither of these can be constructed by the refinement method, which leads to simpler refinement operations.

Example 1: The user-defined data type for stacks is specified and implemented using sequences. That is, the DT for stacks is defined to be $\text{seq}(\tau)$ and its operations have been implemented using the DT operations for $\text{seq}(\tau)$. In addition, the operations for stacks are specified in terms of the operations of $\text{seq}(\tau)$.

Signatures

- empty_stack: $\text{seq}(\alpha)$
- top: $\text{seq}(\alpha) \times \alpha$
- pop: $\text{seq}(\alpha) \times \text{seq}(\alpha)$
- push: $\text{seq}(\alpha) \times \alpha \times \text{seq}(\alpha)$

Modes

- empty_stack: d
- top: i, d
- pop: i, d
- push: i, i, d

FOL specifications of the DT operations of stacks.
\[ \forall s/\text{seq}(\alpha)(\text{empty-stack}(s) \rightarrow s = \langle \rangle) \]
\[ \forall s, s_1/\text{seq}(\alpha), st/\alpha (\text{push}(s, st, s_1) \leftarrow \text{seq-cons}(s, st, s_1)) \]
\[ \forall s, s_1/\text{seq}(\alpha) (\text{pop}(s, s_1) \leftarrow \text{tail}(s, s_1)) \]
\[ \forall s/\text{seq}(\alpha), st/\alpha (\text{top}(s, st) \leftarrow \text{head}(s, st)) \]

**Example 2:** A user-defined data type for graphs is specified and implemented using the built-in DTs sets and tuples and the user-defined DT relations. That is, the DT for graphs is defined to be \( \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \). Note that the DT for relations is defined to be \( \text{set}(\text{tuple}(\alpha, \alpha)) \). The operations of the DT for graphs have been implemented using the DT operations for sets, tuples, relations and graphs.

**Signatures**

- \( \text{empty-graph} : \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)
- \( \text{make-graph} : \text{set}(\text{tuple}(\alpha, \alpha)) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)
- \( \text{vertices} : \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{set}(\alpha) \)
- \( \text{edges} : \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{set}(\text{tuple}(\alpha, \alpha)) \)
- \( \text{edge-add} : \text{tuple}(\alpha, \alpha) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)
- \( \text{edge-del} : \text{tuple}(\alpha, \alpha) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)
- \( \text{vertex-add} : \alpha \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)
- \( \text{vertex-del} : \alpha \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \times \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \)

**Modes**

- \( \text{empty-graph} : d \)
- \( \text{make-graph} : i, d \)
- \( \text{vertices} : i, d \)
- \( \text{edges} : i, d \)
- \( \text{edge-add} : i, i, d \)
- \( \text{edge-del} : i, i, d \)
- \( \text{vertex-add} : i, i, d \)
- \( \text{vertex-del} : i, i, d \)

**FOL specifications of the DT operations of graphs.**

\[ \forall g/\text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha)))(\text{empty-graph}(g) \rightarrow \exists v/\text{set}(\alpha), e/\text{set}(\text{tuple}(\alpha, \alpha))(g = (v, e) \land \text{empty-set}(v) \land \text{empty-set}(e))) \]
\[ \forall e/\text{set}(\text{tuple}(\alpha, \alpha)), g/\text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha)))(\text{make-graph}(e, g) \rightarrow \exists v/\text{set}(\alpha), e_1/\text{set}(\text{tuple}(\alpha, \alpha)), d, r/\text{set}(\alpha)(\text{dom}(e, d) \land \text{ran}(e, r) \land \text{union}(d, r, v) \land g = (v, e_1))) \]
∀v/tuple(set(α), set(tuple(α, α))), v/set(α)(vertices(g, v) ←
  ∃e/set(tuple(α, α)) g = (v, e))
∀v/tuple(set(α), set(tuple(α, α))), e/set(α)(edges(g, e) ←
  ∃e/set(tuple(α, α)) g = (v, e))
∀e/set(tuple(α, α)), g1, g2/tuple(set(α), set(tuple(α, α)))(edge_add(e, g1, g2) ←
  ∃a, b/α, v1, v2/set(α), e1, e2/set(tuple(α, α))(g1 = (v1, e1) ∧
    first(e, a) ∧ member_set(a, v1) ∧ second(e, b) ∧ member_set(b, v1) ∧
    set_elem(e, e1, e2) ∧ g2 = (v1, e2)))
∀e/set(tuple(α, α)), g1, g2/tuple(set(α), set(tuple(α, α)))(edge_del(e, g1, g2) ←
  ∃v1/set(α), e1, e2/set(tuple(α, α))(g1 = (v1, e1) ∧
    member_set(e, e1) ∧ delete(e, e1, e2) ∧ g2 = (v1, e2)))
∀v/α, g1, g2/tuple(set(α), set(tuple(α, α)))(vertex_add(v, g1, g2) ←
  ∃v1, v2/set(α), e1/set(tuple(α, α))(g1 = (v1, e1) ∧
    set_elem(v, v1, v2) ∧ g2 = (v2, e1)))
∀v/α, g1, g2/tuple(set(α), set(tuple(α, α)))(vertex_del(v, g1, g2) ←
  ∃v1, v2/set(α), e1, e2/set(tuple(α, α))(g1 = (v1, e1) ∧
    delete(v, v1, v2) ∧ ∀ev1, ev2/α ((member_set((ev1, ev2), e1) ∧
      (ev1 = v ∨ ev2 = v)) ← delete((ev1, ev2), e1, e2) ∧
      g2 = (v1, e2))))

The implementation in this method of a DT operation for graphs, i.e. make_graph/2, is shown in Section 3.10. The other user-defined DT operations are implemented in a similar way.

3.9.3 Data Types Supporting Meta-programming

This method can be used to construct meta-programs. We have implemented a user-defined DT, called f_struct from Functional STRUCTure, which supports the manipulation of the basic syntactic objects of a logic programming language. The constants in the DT f_struct come from the built-in DT str which is used by f_struct. More complex syntactic objects of such a language can be represented and implemented on top of f_struct and other built-in and user-defined DTs. The ground representation is assumed for representing the programs of the other language.

A unification algorithm has been implemented for terms represented as f_struct. Its implementation has been based on the DT operations of f_struct and on the DT operations of other built-in and user-defined DTs. This unification algorithm is shown in Appendix B.
DT f.struct

The built-in DTs str and seq(α) are imported by f.struct for its implementation. The objects in f.struct are represented by sequences of strings, given by the following BNF syntax.

\[
\begin{align*}
<f\_struct> & ::= <identifier> | \\
& \quad <identifier> '(' <sequence\_f\_struct> ')' \\
<sequence\_f\_struct> & ::= <f\_struct> | \\
& \quad <f\_struct> \{ ',', <f\_struct> \}
<identifier> & ::= ''' <character> {, <character>} '''
\end{align*}
\]

Identifiers are either variables, or constants or function names.

The implemented DT operations of f.struct are the following.

Signatures

- string.to.f.struct: str × seq(str)
- f.struct.to.string: seq(str) × str
- f.struct.args: seq(str) × seq(seq(str))
- nth.arg.f.struct: seq(str) × nat × seq(str)
- f.struct.f.name: seq(str) × str
- arity: seq(str) × nat
- const.f.struct: seq(str)
- var.f.struct: seq(str)
- comp.f.struct: seq(str)

Modes

- string.to.f.struct: i, d
- f.struct.to.string: i, d
- f.struct.args: i, d
- nth.arg.f.struct: i, i, d
- f.struct.f.name: i, d
- arity: i, d
- const.f.struct: i
- var.f.struct: i
- comp.f.struct: i

The informal definitions of these predicate are following. string.to.f.struct(s, f) is true iff \( f \) is the f.struct of the string \( s \). f.struct.to.string(\( f \), s) is true iff \( s \) is the string correspond-
ing to the f\_struct \( f \). \texttt{f\_struct\_args}(f, s) is true iff \( s \) is a sequence with the arguments of the f\_struct \( f \). \texttt{nth\_arg\_f\_struct}(f, n) is true iff \( n \) is the \( n \)th argument of the f\_struct \( f \). \texttt{f\_struct\_name}(f, s) is true iff \( s \) is the identifier of the f\_struct \( f \). \texttt{arity}(f, n) is true iff \( n \) is the arity of the f\_struct \( f \). \texttt{const\_f\_struct}(s) is true iff the f\_struct \( s \) is a constant. \texttt{var\_f\_struct}(s) is true iff the f\_struct \( s \) is a variable. \texttt{comp\_f\_struct}(f) is true iff the f\_struct \( f \) is a compound term.

The implementation of the DT operation \texttt{string\_to\_f\_struct}/2 is shown in Appendix A. Note that f\_struct is analogous to the ADT Units in [36].

### 3.10 Examples

**Example 1: Insertion sort.** The predicate \texttt{insSort}(s, q) is true iff sequence \( q \) is a sorted permutation of sequence \( s \). Insertion sort removes one by one the elements of sequence \( s \) and inserts them into the ordered sequence \( q \) by preserving the order of elements. Let the type of \texttt{insSort}/2 be \( \text{Type}(\text{insSort}) = \text{seq}(\alpha) \times \text{seq}(\alpha) \) and let its expected mode be \( \text{Mode}(\text{insSort}) = (i, d) \).

**Type and Mode Declarations.**

**Signatures**
- \texttt{tail}: seq(al) \times seq(al)
- \texttt{seq\_cons}: seq(al) \times al \times seq(al)
- \texttt{empty\_seq}: seq(al)
- \texttt{head}: seq(al) \times al
- \texttt{le\_int}: int \times int
- \texttt{insSort}: seq(int) \times seq(int)
- \texttt{p2}: seq(int) \times int \times seq(int)
- \texttt{p0}: seq(int)
- \texttt{p1}: seq(int) \times seq(int)
- \texttt{p14}: seq(int) \times int \times seq(int) \times int \times seq(int) \times int \times seq(int)
- \texttt{p12}: seq(int) \times int \times seq(int) \times int
- \texttt{p10}: seq(int) \times int \times seq(int)
- \texttt{p11}: seq(int) \times int \times seq(int)
- \texttt{p13}: seq(int) \times int \times seq(int) \times int
- \texttt{p15}: seq(int) \times int \times seq(int) \times int \times seq(int) \times int \times seq(int)
- \texttt{p16}: seq(int) \times int \times seq(int) \times int \times seq(int) \times int \times seq(int)
- \texttt{p4}: seq(int) \times int \times seq(int)
p3: seq(int) × int × seq(int) × seq(int)
p5: seq(int) × int × seq(int)
p6: seq(int) × int × seq(int)
p7: seq(int) × int × seq(int) × seq(int)
p8: seq(int) × int × seq(int) × int × seq(int) × int × seq(int)
p9: seq(int) × int × seq(int) × int × seq(int) × seq(int)

Modes

- tail: i, d
- seq_cons: i, i, d
- empty_seq: d
- head: i, d
- le_int: i, i
- insSort: i, d

Refinements

1. Schema: Incremental

In order to distinguish the case when \( \nu \) is not empty from the case when \( \nu \) is empty the names Subgoal-A and Subgoal-B are used respectively in the constructed programs. In addition, SubgoalN-A and SubgoalN-B stand for Subgoal schema with \( N \) literal schemata in its body where \( N \geq 3 \). SubgoalA and SubgoalB stand for Subgoal schema with 2 literal schemata in its body.
\text{insSort}(x_1, x_4) \leftarrow p_0(x_1) \land p_1(x_1, x_4) \\
\text{insSort}(x_1, x_4) \leftarrow \neg p_0(x_1) \land p_2(x_1, x_2, x_3) \land \\
\text{insSort}(x_3, x_5) \land p_3(x_1, x_2, x_5, x_4)

\text{Refinement by DT operations or Equality}
\begin{align*}
p_0(x_1) &\leftarrow \text{empty_seq}(x_1) \\
p_1(x_1, x_4) &\leftarrow \text{empty_seq}(x_4)
\end{align*}

\text{Schema: } Subgoal.B
\begin{align*}
p_2(x_1, x_2, x_3) &\leftarrow p_4(x_1, x_2, x_3) \land p_5(x_1, x_2, x_3)
\end{align*}

\text{Schema: Incremental}
\begin{align*}
p_3(x_1, x_2, x_5, x_4) &\leftarrow p_6(x_1, x_2, x_5) \land p_7(x_1, x_2, x_5, x_4) \\
p_3(x_1, x_2, x_5, x_4) &\leftarrow \neg p_6(x_1, x_2, x_5) \land p_8(x_1, x_2, x_5, x_6, x_7, x_8, x_9) \land \\
p_3(x_7, x_8, x_9, x_{11}) &\land p_9(x_1, x_2, x_5, x_6, x_{11}, x_4)
\end{align*}

\text{Refinement by DT operations or Equality}
\begin{align*}
p_4(x_1, x_2, x_3) &\leftarrow \text{head}(x_1, x_2) \\
p_5(x_1, x_2, x_3) &\leftarrow \text{tail}(x_1, x_3)
\end{align*}

\text{Schema: Case}
\begin{align*}
p_6(x_1, x_2, x_5) &\leftarrow p_{10}(x_1, x_2, x_5) \\
p_6(x_1, x_2, x_5) &\leftarrow p_{11}(x_1, x_2, x_5)
\end{align*}

\text{Refinement by DT operations or Equality}
\begin{align*}
p_7(x_1, x_2, x_5, x_4) &\leftarrow \text{seq_cons}(x_5, x_2, x_4)
\end{align*}

\text{Schema: Subgoal3.B}
\begin{align*}
p_8(x_1, x_2, x_5, x_6, x_7, x_8, x_9) &\leftarrow p_{14}(x_1, x_2, x_5, x_6, x_7, x_8, x_9) \land \\
p_{15}(x_1, x_2, x_5, x_6, x_7, x_8, x_9) &\land p_{16}(x_1, x_2, x_5, x_6, x_7, x_8, x_9)
\end{align*}

\text{Refinement by DT operations or Equality}
\begin{align*}
p_9(x_1, x_2, x_5, x_6, x_{11}, x_4) &\leftarrow \text{seq_cons}(x_{11}, x_6, x_4) \\
p_{10}(x_1, x_2, x_5) &\leftarrow \text{empty_seq}(x_5)
\end{align*}

\text{Schema: SubgoalA}
\begin{align*}
p_{11}(x_1, x_2, x_5) &\leftarrow p_{12}(x_1, x_2, x_5, x_{10}) \land p_{13}(x_1, x_2, x_5, x_{10})
\end{align*}

\text{Refinement by DT operations or Equality}
\begin{align*}
p_{14}(x_1, x_2, x_5, x_6, x_7, x_8, x_9) &\leftarrow \text{head}(x_5, x_6)
\end{align*}
implementations of built-in dt operations.

dt operations for sequences

\[ \text{empty_seq}([]) \]

\[ \text{head}([h \mid t], h) \]

\[ \text{tail}([h \mid t], t) \]

\[ \text{seq_cons}(q, x, [x \mid q]) \]

equality

\[ \text{eq}(x, x) \]

dt operations for integers

\[ \text{le_int}(n_1, n_2) \leftarrow n_1 \leq n_2 \]

the refinement tree for insertion sort is shown in figure 3.2

implementations of dt operations for user-defined dt's

this example is intended to show the implementation of user-defined dt operations using this methodology. the implementation of a more complex dt operation for the dt \( f \text{-struct} \), i.e. \( \text{string-to-f-struct/2} \), is shown in appendix a.

example 2: the predicate \( \text{make_graph}(e, g) \) is true iff \( g \) is a graph with edges the set \( e \) and vertices the set of vertices which are used in \( e \). let us assume the typed \( \text{cga make_graph}(e, g) \) whose arguments \( e \) and \( g \) have types \( \text{set}(\text{tuple}(\alpha, \alpha)) \) and \( \text{tuple}(\text{set}(\alpha), \text{set}(\text{tuple}(\alpha, \alpha))) \) respectively. let us assume that the expected mode of \( \text{make_graph}/2 \) is \( \text{Mode(make_graph)} = (i, d) \).

type and mode declarations.

signatures

\[ \text{ran}: \text{set}(\text{tuple}(a_1, a_2)) \times \text{set}(a_2) \]

\[ \text{dom}: \text{set}(\text{tuple}(a_1, a_2)) \times \text{set}(a_1) \]
Figure 3.2: Refinement tree for predicate \textit{insSort}/2.

\texttt{insSort(x_1,x_4)}

\textbf{Incremental}

\texttt{p_0(x_1)} \quad \texttt{p_1(x_1,x_4)} \quad \texttt{p_2(x_1,x_2,x_3)} \quad \texttt{p_3(x_1,x_2,x_5,x_4)}

\texttt{empty\_seq(y_1)} \quad \texttt{empty\_seq(y_1)} \quad \textbf{Subgoal\_B} \quad \textbf{Incremental}

\texttt{p_4(x_1,x_2,x_3)} \quad \texttt{p_5(x_1,x_2,x_3)} \quad \texttt{head(y_1,y_2)} \quad \texttt{tail(y_1,y_2)}

\texttt{p_6(x_1,x_2,x_5)} \quad \texttt{p_7(x_1,x_2,x_5,x_4)} \quad \texttt{p_8(x_1,x_2,x_5,x_6,x_7,x_8,x_9)}

\texttt{Case} \quad \texttt{seq\_cons(y_1,y_2,y_3)} \quad \textbf{Subgoal\_3\_B}

\texttt{p_9(x_1,x_2,x_5,x_6,x_11,x_4)} \quad \texttt{seq\_cons(y_1,y_2,y_3)}

\texttt{p_{10}(x_1,x_2,x_5)} \quad \texttt{p_{11}(x_1,x_2,x_5)} \quad \textbf{Subgoal\_A}

\texttt{empty\_seq(y_1)} \quad \texttt{empty\_seq(y_1)}

\texttt{p_{12}(x_1,x_2,x_5,x_{10})} \quad \texttt{p_{13}(x_1,x_2,x_5,x_{10})} \quad \texttt{head(y_1,y_2)} \quad \texttt{le\_int(y_1,y_2)}

\texttt{p_{14}(x_1,x_2,x_5,x_6,x_7,x_8,x_9)} \quad \texttt{head(y_1,y_2)}

\texttt{p_{15}(x_1,x_2,x_5,x_6,x_7,x_8,x_9)} \quad \texttt{tail(y_1,y_2)}

\texttt{p_{16}(x_1,x_2,x_5,x_6,x_7,x_8,x_9)} \quad \texttt{eq(y_1,y_2)}

90
make_pair: \( a_1 \times a_2 \times \text{tuple}(a_1,a_2) \)
make_graph: \( \text{set}(\text{tuple}(a_1,a_1)) \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
union: \( \text{set}(a_1) \times \text{set}(a_1) \times \text{set}(a_1) \)
p0: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
p1: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
p2: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_3 \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
p3: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_3 \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
p4: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_3 \times a_4 \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)
p5: \( \text{set}(\text{tuple}(a_1,a_1)) \times a_3 \times a_4 \times a_5 \times \text{tuple}(\text{set}(a_1),\text{set}(\text{tuple}(a_1,a_1))) \)

Modes
- ran: \( i, d \)
- dom: \( i, d \)
- make_pair: \( i, i, d \)
- make_graph: \( i, d \)
- union: \( i, i, d \)
- p0: \( i, d, d \)
- p1: \( i, i, d \)
- p2: \( i, d, d, d \)
- p3: \( i, i, d, d \)
- p4: \( i, i, d, d, d \)
- p5: \( i, i, i, d, d \)

Refinements.

Schema: Subgoal \( A \)

\[
\text{make_graph}(x_1,x_4) \leftarrow p_0(x_1,x_3,x_4) \land p_1(x_1,x_3,x_4) \\
p_0(x_1,x_3,x_4) \leftarrow p_2(x_1,x_2,x_3,x_4) \land p_3(x_1,x_2,x_3,x_4) \\
p_3(x_1,x_2,x_3,x_4) \leftarrow p_4(x_1,x_2,x_5,x_3,x_4) \land p_5(x_1,x_2,x_5,x_3,x_4)
\]

Refinement by DT operations
- \( p_1(x_1,x_3,x_4) \leftarrow \text{make.pair}(x_3,x_1,x_4) \)
- \( p_2(x_1,x_2,x_3,x_4) \leftarrow \text{dom}(x_1,x_2) \)
- \( p_4(x_1,x_2,x_5,x_3,x_4) \leftarrow \text{ran}(x_1,x_5) \)
- \( p_5(x_1,x_2,x_5,x_3,x_4) \leftarrow \text{union}(x_2,x_5,x_3) \)

The DT operations of sets, tuples and relations which are used in the implementations of make_graph/2 are as follows.

1. DT operations for sets: union/3.
2. DT operations for tuples: \textit{make_pair/3}.

3. DT operations for relations: \textit{dom/2, ran/2}.
Chapter 4

Mode Analysis

4.1 Introduction

This chapter presents the mode analysis procedure used during construction of SI-programs. Mode analysis is a formal framework for static analysis of programs in order to derive runtime properties without running them. Static mode analysis has been studied in [8], [17], [18], [19], [44], [60], [63], [64] among others. The collected mode information in these analysis frameworks is intended to support optimizing compilers to generate efficient code. 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 mode analysis of this thesis supports the program construction process. Its aim is to ensure that the declared modes as defined for the DT operations are consistent with the inferred runtime modes. The mode analysis method is based on minimal function graphs (mfgs).

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 i and d which stand for “ground” and “don’t know” respectively. For example, the DT operation head(sq,el) is defined to be true if el is the first element of the sequence.
SI-programs are intended to satisfy the requirement that each DT operation should have inferred mode which is subsumed by the declared one. The mode of this DT operation is \( \text{Mode}(\text{head}) = (i, d) \) meaning that \( sq \) is supposed to be ground when \( \text{head}(sq, el) \) is called.

Let us assume that \( \text{head}(sq, el) \) 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 the 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).

The minimal function graphs method originally was developed in functional languages in \[42\] for program analysis. This method has been introduced in logic programming in \[94\], \[95\] for analysis-based compiler optimizations and in \[29\] for program specialization. The mfgs approach is designed to generate only the call patterns and their success patterns that are reachable from a given input query.

### 4.1.1 Abstract Interpretation

The standard concepts of abstract interpretation are reviewed in this section. Abstract interpretation is a formal framework for static analysis of the run-time properties of logic programs \[1\], \[7\], \[9\], \[15\], \[34\], \[39\], \[43\].

**Definition 4.1.1:** A poset \((D, \sqsubseteq_D)\) is a complete partially ordered set (cpos) iff every chain \( Y \) in \( D \) has a least upper bound \((\sqcup_Y)\) 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.

We describe functions by their graphs. The graph representation of a function is a set of pairs of input - output values. The partial ordering on functions is defined with respect to their graphs. Let \( f, g : D_1 \rightarrow D_2 \), \( f \sqsubseteq g \) iff \( f \)'s graph is a subset of \( g \)'s graph.

**Definition 4.1.2:** Let \( F : D \rightarrow D \) be a functional and let \( d \) be 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 \( d \subseteq e \).

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.

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

\[ F : \text{Prog} \rightarrow (D \rightarrow D) \]

The concrete semantics of a program \( Pr \in \text{Prog} \) is defined to be \( \text{lfp}(F(Pr)) \). \( F \) is defined in terms of some other functions. Some of them are domain-independent and the others domain-dependent. The latter are referred to here as operations.

An abstract interpretation is defined by:

1. A cpo \((D^\alpha, \subseteq_{D^\alpha})\) where \( D^\alpha \) stands for the abstract domain. The abstract domain is a description of the concrete domain.
2. An abstract semantic function \( F^\alpha \) which defines the abstract denotation of logic programs. \( F^\alpha(Pr) \) is continuous with respect to the ordering \( sqsubseteq_{D^\alpha} \).

\[ F^\alpha : \text{Prog} \rightarrow (D^\alpha \rightarrow D^\alpha) \]

The abstract semantics of \( Pr \) is defined to be \( \text{lfp}(F^\alpha(Pr)) \).
3. The abstraction function \( \alpha \) which is defined to be \( \alpha : D \rightarrow D^\alpha \). The concretization function \( \gamma \) which is defined to be \( \gamma : D^\alpha \rightarrow D \). In addition,
   a. Functions \( \alpha \) and \( \gamma \) are monotonic.
   b. Functions \( \alpha \) and \( \gamma \) are adjoint. That is
      \[ \forall d \in D : d \subseteq_D \gamma(\alpha(d)) \]
      \[ \forall d^\alpha \in D^\alpha : d^\alpha = \alpha(\gamma(d^\alpha)) \]

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

Definition 4.1.3: Given the two semantic functions \( F \) and \( F^\alpha \) and the monotonic concretization function \( \gamma \), \( F^\alpha \) is a safe approximation of \( F \) iff \( \forall Pr \in \text{Prog} \ lfp(F(Pr)) \sqsubseteq_D \gamma(lfp(F^\alpha(Pr))). \)

If each abstract operation \( f^\alpha \) safely approximates the corresponding concrete \( f \) then \( F^\alpha \) is a safe approximation of \( F \).

4.2 Abstract Analysis Framework

Definition 4.2.1: Abstract terms are the terms of the set \( D_{aTerm} = \{i,d\} \).

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, \( i \) "is included by" \( d \) which is denoted by \( i \sqsubseteq_{aTerm} d \).

Definition 4.2.2: \( (D_{aTerm}, \sqsubseteq_{aTerm}) \) is a pointed cpo. The lub \( (\sqcup_{aTerm}) \) and glb \( (\sqcap_{aTerm}) \) of \( (D_{aTerm}, \sqsubseteq_{aTerm}) \) are defined as follows:

\[
\begin{align*}
\sqcup_{aTerm} d &= d \\
\sqcap_{aTerm} d &= i
\end{align*}
\]

Definition 4.2.3: An abstract substitution \( \lambda \) 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, \( \lambda \) and \( \mu \) will denote abstract substitutions.

Definition 4.2.4: 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 4.2.5: Let \( p(m_1, \ldots, m_n) \) and \( p(m_2, \ldots, m_n) \) be two abstract atoms for predicate \( p/n \) where \( m_1, m_2 \in DaTerm \ (1 \leq i \leq n) \)

\[
p(m_1, \ldots, m_n) \sqsubseteq_p p(m_2, \ldots, m_n) \text{ iff } m_1 \sqsubseteq_{aTerm} m_2^1 \land \ldots \land m_n \sqsubseteq_{aTerm} m_n^2
\]

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 \( d \)'s, i.e. \( p(d, \ldots, d) \). The bottom element of \( D_p \) is the atom with arguments only \( i \)'s, i.e. \( p(i, \ldots, i) \). The lub \((\sqcup_p)\) for any two elements in \( D_p \) is defined as follows.

Definition 4.2.6: Let \( p(m_1, \ldots, m_n) \) and \( p(m_1, \ldots, m_n) \) be two abstract atoms for predicate \( p/n \) where \( m_1, m_2 \in DaTerm \) and \( m_1, m_2 \in DaTerm \ (1 \leq i \leq n) \).

\[
p(m_1, \ldots, m_n) \sqcup_p p(m_1, \ldots, m_n) = p(m_1, \ldots, m_n),
\]

where \( m_i = m_1 \sqcup_{aTerm} m_2^i \ (1 \leq i \leq n) \). For example, \( p(i, i, d, i, d) \sqcup_p p(i, d, d, i, i) = p(i, d, d, i, d) \).

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 \( DaAtom \). \( Psymb \) stands for the set of predicate symbols of the predicates in logic program \( Pr \).

Definition 4.2.7: Let \( I_1 \) and \( I_2 \) be two elements of the abstract domain \( DaAtom \). The ordering \( \sqsubseteq_{aAtom} \) on \( DaAtom \) is defined as follows.

\[
I_1 \sqsubseteq_{aAtom} I_2 \text{ iff } I_1 = \perp_{aAtom} \text{ or } I_1 \sqsubseteq_p I_2 \text{ for some } p \in Psymb
\]

\( DaAtom \) is a pointed cpo with ordering \( \sqsubseteq_{aAtom} \).

Definition 4.2.8: Let \( I_1 \) and \( I_2 \) be two elements of the abstract domain \( DaAtom \). The lub \( I_1 \sqcup_{aAtom} I_2 \) is defined as follows.

\[
I_1 \sqcup_{aAtom} I_2 = \begin{cases} I_1 & \text{if } I_2 = \perp_{aAtom} \\ I_2 & \text{if } I_1 = \perp_{aAtom} \\ I_1 \sqcup_p I_2 & \text{otherwise} \end{cases}
\]

97
4.3 Mode Analysis Algorithm

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 \( \text{call}(p(t_1, \ldots, t_n)) \) where \( p(t_1, \ldots, t_n) \) is an abstract atom, refers to the abstract call atom \( \text{call}(p(m_1, \ldots, m_n)) \) corresponding to \( p \). Similarly, \( \text{ans}(I) \) refers to the abstract success atom for \( I \). With this notation, a program description is represented as a set of expressions of the form \( \text{call}(I) \) and \( \text{ans}(f) \) 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 4.3.1:** 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. \( \text{match}(p(t_1, \ldots, t_n), p(m_1, \ldots, m_n)) \) is an operation that is defined as follows.

\[
\text{match}(p(t_1, \ldots, t_n), p(m_1, \ldots, m_n)) = \{t_1/m_1, \ldots, t_n/m_n\}
\]

For example, let's assume the atom \( p(x_1, x_2, x_3) \) and the abstract atom \( p(i, i, d) \) then

\[
\text{match}(p(x_1, x_2, x_3), p(i, i, d)) = \{x_1/i, x_2/i, x_3/d\}
\]

Matching substitutions can be composed, making use of the \( \cap_{a\text{Term}} \) operation.

**Substitution Composition:**

**Definition 4.3.2:** Let \( \lambda_1 = \{x_1/m_1, \ldots, x_{n_1}/m_{n_1}\} \) and \( \lambda_2 = \{y_1/m_1, \ldots, y_{n_2}/m_{n_2}\} \) be two abstract substitutions. The composition of abstract substitutions is denoted by \( \lambda_1 \circ_{a} \lambda_2 \). It is defined as follows.

\[
\lambda_1 \circ_{a} \lambda_2 = \{z/m | x_i/m_i^1 \in \lambda_1 \land y_j/m_j^2 \in \lambda_2 \land x_i = y_j = z \land m = m_i^1 \cap_{a\text{Term}} m_j^2 \} \cup \\
\{x_i/m_i^1 | x_i/m_i^1 \in \lambda_1 \land z \notin \text{dom}(\lambda_2)\} \cup \\
\{y_j/m_j^2 | y_j/m_j^2 \in \lambda_2 \land z \notin \text{dom}(\lambda_1)\}
\]

where \( 0 \leq i \leq n_1 \) and \( 0 \leq j \leq n_2 \).

For example, if \( \lambda_1 = \{x_1/i, x_2/d, x_3/d\} \) and \( \lambda_2 = \{x_1/i, x_3/i\} \) then

\[
\lambda_1 \circ_{a} \lambda_2 = \{x_1/i, x_2/d, x_3/i\}
\]
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.

$T_0 := \{ \text{call}(p(m_1, \ldots, m_n)) \} \cup \{ \text{ans}(dt_1(i, \ldots, i)), \ldots, \text{ans}(dt_k(i, \ldots, i)) \}$

$i := 0$

repeat

$T_{i+1} := \text{lub}(T_i, \text{calls}(Pr, T_i), \text{answers}(Pr, T_i))$

$i := i + 1$

until $T_i = T_{i-1}$

where

$\text{calls}(Pr, T_i) =$

\[
\begin{cases}
\text{call}(p_j(m_1, \ldots, m_k)) & \begin{array}{l}
\hline
h \leftarrow l_1, \ldots, p_j(x_1, \ldots, x_k), \ldots, l_n \in Pr, \\
\{\text{call}(h), \text{ans}(l_1), \ldots, \text{ans}(l_{j-1})\} \subseteq T_i, \\
\Phi = \text{match}(h, \text{call}(h)) \circ \text{match}(l_1, \text{ans}(l_1)) \circ \ldots \\
\circ \text{match}(l_{j-1}, \text{ans}(l_{j-1})) \cup \\
\{x/d \mid x \text{ occurs in } \{x_1, \ldots, x_k\} \text{ and} \\
\text{ does not occur in } l_1, \ldots, l_{j-1}\} \\
\{x_{1/m_1}, \ldots, x_{k/m_k}\} \subseteq \Phi
\end{array}
\end{cases}
\]

$\text{answers}(Pr, T_i) =$

\[
\begin{cases}
n(x_1, \ldots, x_k) \leftarrow l_1, \ldots, l_n \in Pr, \\
\{\text{call}(h), \text{ans}(l_1), \ldots, \text{ans}(l_n)\} \subseteq T_i, \\
\Phi = \text{match}(h, \text{call}(h)) \circ \text{match}(l_1, \text{ans}(l_1)) \circ \ldots \\
\circ \text{match}(l_n, \text{ans}(l_n)), \\
\{x_{1/m_1}, \ldots, x_{k/m_k}\} \subseteq \Phi
\end{cases}
\]

Note that negative literals are ignored in both $\text{calls}(Pr, T_i)$ and $\text{answers}(Pr, T_i)$. The upper bound on program descriptions is as follows.
\[ \text{lub}(T, \text{calls}(Pr, Ti), \text{answers}(Pr, Ti)) = \\
\{\text{call}(I^1 \cup_a \text{Atom}, I^2) \mid \text{call}(I^1) \in T_i \text{ and call}(I^2) \in \text{calls}(Pr, Ti)\} \cup \\
\{\text{ans}(I^1 \cup_a \text{Atom}, I^2) \mid \text{ans}(I^1) \in T_i \text{ and ans}(I^2) \in \text{answers}(Pr, Ti)\} \]

In this definition, it is assumed that \( \text{call}(I) = \text{call}(\bot_a \text{Atom}) \) for a predicate \( p \) if there is no expression in \( T_i \) of form \( \text{call}(p(m_1, \ldots, m_n)) \). A similar comment applies for \( \text{ans}(I) \).

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 \text{call} and \text{success} atoms for each predicate are the \text{lub} of all \text{call} and \text{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 d's for all arguments.

4.4 Examples

The next example illustrates the following:

1. The mode analysis of a partial SI-program.
2. The argument matching of DT operations with the ones of the predicate they define. The detection of an error after matching inappropriate arguments.
3. The satisfaction of mode requirements by the modes of a partial and a complete SI-programs.

Example 1: The predicate \( \text{max.seq}(x_1, x_4) \) where \( \text{Type}(\text{max.seq}) = \text{seq}(\text{int}) \times \text{int} \) and \( \text{Mode}(\text{max.seq}) = (i, d) \) is true iff \( x_4 \) is the maximum element from the elements of sequence \( x_1 \).

Suppose that we want to construct a logic program for the predicate \( \text{max.seq}/2 \). Let us assume the typed cga \( \text{max.seq}(x_1, x_4) \) whose arguments \( x_1, x_4 \) have types seq(int) and int respectively. Let us assume that we have constructed the following partial logic program.

Signatures
\[ \text{empty.seq}: \text{seq}(a1) \]
Modes

empty_seq: d

tail: i, d

head: i, d

max_seq: i, d

p0: i

p1: i, d

p2: i, d, d

p3: undefined, undefined, undefined, undefined

p4: i, d

p5: i, i

p6: i, d, d

p7: i, i, d

p8: i, d, d

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)
This partial program illustrates the modes of the predicates which have been derived at this stage of development. The DT operations have the user-defined declared modes while the modes of the other predicates have been derived by the mode inference procedure during its last call. That is, after matching the arguments of equality 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. It is worth noting that the inferred modes of the DT operations are as follows.

\[
\begin{align*}
    \text{Mode}(\text{empty_seq}) &= i \\
    \text{Mode}(\text{head}) &= i, d \\
    \text{Mode}(\text{tail}) &= i, d
\end{align*}
\]

That is, the inferred mode of empty_seq/1 is \( i \) which is subsumed by its declared one. The other DT operations have same inferred and declared modes.

We are at the stage of matching the arguments of DT operations with the ones of the predicates that they define. We would like to match the arguments of predicate \( p8(x_1,x_5,x_6) \) with the ones of the DT operation head\( (y_1,y_2) \). At this stage of development the system displays the types and the modes of the predicates \( p8/3 \) and head\( /2 \). That is,

\[
\begin{align*}
    \text{Type} \ p8: \ &\text{seq(int)} \times a_2 \times \text{seq(int)} \\
    \text{Type} \ head: \ &\text{seq(a3)} \times a_3 \\
    \text{Mode} \ p8: \ &i, d, d \\
    \text{Mode} \ head: \ &i, d
\end{align*}
\]

If we match the argument pairs \( (x_5,y_2) \) and \( (x_6,y_1) \) the clause \( p8(x_1,x_5,x_6) \leftarrow \text{head}(x_6,x_5) \) 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 \( d, d \) 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 \( i \) to be *ground*. The correct matching of argument pairs is \( (x_5,y_2) \) and \( (x_1,y_1) \). The mode analysis will infer for the predicate head\( /2 \) mode \( i, d \) which is subsumed by the declared one. In this case, the next clause is added to the partial logic program.

\[
p8(x_1,x_5,x_6) \leftarrow \text{head}(x_1,x_5)
\]
If we refine the remaining undefined predicates we will get the following complete program for \texttt{max_seq/2}.

Signatures

- \texttt{tail}: $\text{seq(al)} \times \text{seq(al)}$
- \texttt{head}: $\text{seq(al)} \times a1$
- \texttt{empty_seq}: $\text{seq(a1)}$
- \texttt{ge_int}: $\text{int} \times \text{int}$
- \texttt{le_int}: $\text{int} \times \text{int}$
- \texttt{max_seq}: $\text{seq(int)} \times \text{int}$
- \texttt{p0}: $\text{seq(int)}$
- \texttt{p1}: $\text{seq(int)} \times \text{int}$
- \texttt{p2}: $\text{seq(int)} \times \text{int} \times \text{seq(int)}$
- \texttt{p3}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p4}: $\text{seq(int)} \times \text{seq(int)}$
- \texttt{p5}: $\text{seq(int)} \times \text{seq(int)}$
- \texttt{p6}: $\text{seq(int)} \times \text{int} \times \text{int}$
- \texttt{p7}: $\text{seq(int)} \times \text{int} \times \text{int}$
- \texttt{p8}: $\text{seq(int)} \times \text{int} \times \text{seq(int)}$
- \texttt{p9}: $\text{seq(int)} \times \text{int} \times \text{seq(int)}$
- \texttt{p10}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p11}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p12}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p13}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p14}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$
- \texttt{p15}: $\text{seq(int)} \times \text{int} \times \text{int} \times \text{int}$

Modes

- \texttt{tail}: $i, d$
- \texttt{head}: $i, d$
- \texttt{empty_seq}: $d$
- \texttt{ge_int}: $i, i$
- \texttt{le_int}: $i, i$
- \texttt{max_seq}: $i, d$
- \texttt{p0}: $i$
- \texttt{p1}: $i, d$
- \texttt{p2}: $i, d, d$
- \texttt{p3}: $i, i, i, d$
- \texttt{p4}: $i, d$
Program clauses

\[
\begin{align*}
\text{max_seq}(x_1, x_3) & \leftarrow p_0(x_1) \land p_1(x_1, x_3) \\
\text{max_seq}(x_1, x_3) & \leftarrow \neg p_0(x_1) \land p_2(x_1, x_4, x_5) \land \text{max_seq}(x_5, x_6) \land p_3(x_1, x_4, x_6, x_3) \\
p_0(x_1) & \leftarrow p_4(x_1, x_2) \land p_5(x_1, x_2) \\
p_1(x_1, x_3) & \leftarrow p_6(x_1, x_7, x_3) \land p_7(x_1, x_7, x_3) \\
p_2(x_1, x_4, x_5) & \leftarrow p_8(x_1, x_4, x_5) \land p_9(x_1, x_4, x_5) \\
p_3(x_1, x_4, x_6, x_3) & \leftarrow p_{10}(x_1, x_4, x_6, x_3) \\
p_3(x_1, x_4, x_6, x_3) & \leftarrow p_{11}(x_1, x_4, x_6, x_3) \\
p_4(x_1, x_2) & \leftarrow \text{tail}(x_1, x_2) \\
p_5(x_1, x_2) & \leftarrow \text{empty_seq}(x_2) \\
p_6(x_1, x_7, x_3) & \leftarrow \text{head}(x_1, x_7) \\
p_7(x_1, x_7, x_3) & \leftarrow \text{eq}(x_7, x_3) \\
p_8(x_1, x_4, x_5) & \leftarrow \text{head}(x_1, x_4) \\
p_9(x_1, x_4, x_5) & \leftarrow \text{tail}(x_1, x_5) \\
p_{10}(x_1, x_4, x_6, x_3) & \leftarrow p_{12}(x_1, x_4, x_6, x_3) \land p_{13}(x_1, x_4, x_6, x_3) \\
p_{11}(x_1, x_4, x_6, x_3) & \leftarrow p_{14}(x_1, x_4, x_6, x_3) \land p_{15}(x_1, x_4, x_6, x_3) \\
p_{12}(x_1, x_4, x_6, x_3) & \leftarrow \text{ge_int}(x_4, x_6) \\
p_{13}(x_1, x_4, x_6, x_3) & \leftarrow \text{eq}(x_3, x_4) \\
p_{14}(x_1, x_4, x_6, x_3) & \leftarrow \text{le_int}(x_4, x_6) \\
p_{15}(x_1, x_4, x_6, x_3) & \leftarrow \text{eq}(x_3, x_6)
\end{align*}
\]

The mode analysis of the complete program with respect to goal \(G^o = \text{max_seq}(i, d)\) has generated the modes in the above program. The inferred modes of the DT operations are as follows.

\[
\begin{align*}
\text{Mode}(\text{empty_seq}) &= i \\
\text{Mode}(\text{head}) &= i, d
\end{align*}
\]
\[
\text{Mode(tail)} = \text{i, d} \\
\text{Mode(ge.int)} = \text{i, i} \\
\text{Mode(le.int)} = \text{i, i}
\]

The inferred mode of each DT operation is subsumed by its declared one.

### 4.5 SI-Programs are Well-Modeled

**Definition 4.5.1:** Let \( G^c = \neg p(t_1, \ldots, t_n) \) 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 4.5.2:** Let \( p(t_1, \ldots, t_n) \) 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 \( p(t_1, \ldots, t_n) \) is subsumed by the mode of \( p/n \) if the following hold. If \( m_i = \text{i} \) then \( t_i \ (1 \leq i \leq n) \) is a ground term. If \( m_i = \text{d} \) then \( t_i \ (1 \leq i \leq n) \) is any term.

**Proposition 4.5.1:** Let \( \text{Prog} \) be an SI-Program with top-level predicate \( p/n \). Let \( G^c = \neg p(t_1, \ldots, t_n) \) 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 \( \text{Prog} \) is well-modeled if the evaluation of goal \( G^c \) generates for each predicate of \( \text{Prog} \) calls whose modes are subsumed by the mode of its predicate.

**Proof:** The modes of the predicates of SI-program \( \text{Prog} \) have been 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 \( \text{Prog} \) 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 \( \text{Prog} \) is well-modeled.

### 4.6 Discussion

The refinement of the undefined predicates is performed in left-right order the same as the evaluation order. Suppose that the literal \( l_i \) from the 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 the 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. In addition, the main differences of this mode analysis from any previous work we are aware of is its use to guide
the application of DT refinements to undefined predicates during program development, and the analysis is performed on partial logic programs.
Chapter 5

Correctness of SI-Programs

5.1 Introduction

In this chapter, the problem of proving the correctness of an SI-program with respect to a specification is considered. A specification describes what the eventual software must do and not how to do it. The algorithm determines how the software will achieve the specified functionality. A specification can be expressed in natural language or in a formal language. Natural languages are vague and ambiguous and can thus cause misunderstandings between specifiers and developers. Formal specifications are expressed in formal languages. They are described precisely in a mathematical notation. Their major potential advantage is that implementations can be verified.

There are two approaches to the problem of correctness of a program. One approach is to test the program by a finite set of tests. Testing can demonstrate that a program is free of bugs and behaves as expected for a given finite set of tests. For complex input/output it is hard to know whether it is correct or not. Testing cannot show complete absence of bugs. The other approach is to prove that a program is correct with respect to its specification by formal reasoning.

Given a specification there are two main approaches to proving the correctness of a program. That is,

1. Correctness is ensured during the construction of the program [20], [33], [38], [54],
In this case a specification is successively refined into a program. The construction process consists either of smaller predefined development steps whose correctness has been proved or of equivalence preserving transformations.

2. The program is first constructed by any means and then it is proved to be correct with respect to its specification [6], [13], [14].

First-order logic (FOL) is a commonly used formalism for program specification in imperative, logic and functional programming [4], [6], [13], [14], [16], [20], [23], [32], [33], [38], [51], [54], [67], [68]. The boundaries between specifications and programs are not rigid in logic languages. Their advantage over imperative programming languages is their closeness to FOL. The use of the same formalism for programs and specifications potentially facilitates the verification of programs with respect to their specifications as well as their derivation from specifications.

In the literature, there are several approaches to verifying logic programs, using FOL as a specification language. Sometimes full FOL is used [13], [14], [33], [89], and sometimes a subset of FOL [16], [20], [51], [54]. In these approaches, a specification consists of a set of FOL formulas each formula of which expresses the specification of one relation. This may be specified in terms of other relations down to the level of primitive components. A formal specification in FOL is called a logic specification.

A logic specification in this thesis essentially follows these approaches, but uses typed FOL with equality. Types are an important component of specifications. The nature of the knowledge involved in specifications is typed. Untyped logic specifications are not expressive enough for specifying large software systems. Popular specification languages whose aim is the specification of complex software use typed logic [22], [41], [85].

In this thesis, first a program is constructed using this method then its correctness is proved with respect to its logic specification. That is, we follow the second approach to prove the correctness of SI-programs.

The highly structured form of SI-programs may facilitate the formulation of correctness proofs to some degree. In this chapter, it will be claimed that the structure of an SI-program is reflected in the structure of its correctness proof. In other words, there are "proof schemes" corresponding to design schemata.
5.2 Logic Specifications

For each relation \( p \) to be implemented, there are two versions - its specification \( p^S \) and its implementation \( p \).

**Definition 5.2.1:** Let \( p^S \) be a predicate. The logic specification \( \text{Spec}_p \) for \( p^S \) is defined to be a formula in polymorphic many-sorted FOL of the form

\[
\text{Spec}_p = \forall \bar{x}/\bar{\tau} \ (p^S(\bar{x}) \iff \text{Def}_p)
\]

where \( \bar{x} \) is a tuple of distinct variables and \( \bar{\tau} \) is a tuple of sorts corresponding to the variables in \( \bar{x} \). \( \text{Def}_p \) is a formula in polymorphic many-sorted first-order logic which defines the relation \( p^S(\bar{x}) \).

*Example 1:* The predicate \( \text{sum}^S(q, s) \) where \( \text{Type}(\text{sum}^S) = \text{seq}(Z) \times Z \) is true iff \( s \) is the sum of integers of sequence \( q \).

**Logic specification:**
\[
\forall q/\text{seq}(Z), s/Z \ (\text{sum}^S(q, s) \iff s = \sum_{i=1}^{#q} q_i)
\]

where \( #q \) stands for the length of the sequence \( q \) and \( \sum \) is the summation operator for all numeric data types.

*Example 2:* \( \text{fac}^S(n, f) \) where \( \text{Type}(\text{fac}^S) = \text{N} \times \text{N} \) is true iff \( f \) is the factorial of \( n \).

**Logic specification:**
\[
\forall n/\text{N}, f/\text{N} \ (\text{fac}^S(n, f) \iff f = n!)
\]

*Example 3:* \( \text{incrOrd}^S(q) \) where \( \text{Type}(\text{incrOrd}^S) = \text{seq}(\alpha) \) is true iff the sequence \( q \) is in increasing order.

**Logic specification:**
\[
\forall q/\text{seq}(\alpha) \ (\text{incrOrd}^S(q) \iff \forall i/N_1 \ (1 \leq i \leq (#q - 1) \rightarrow q_i \leq q_{i+1}))
\]
5.3 Correctness Method

The meaning of SI-programs is defined using program completion semantics.

**Definition 5.3.1:** Let $Pr$ be a set of clauses. Let $p(x_1,\ldots,x_n) \leftarrow L_1,\ldots,L_m$ be a clause where $x_1,\ldots,x_n$ are distinct variables and $L_1,\ldots,L_m$ are literals whose arguments are variables or constants. Let $y_1,\ldots,y_d$ be the variables in the body of clause which do not appear in its head. The following clause is equivalent to the previous one

$$p(x_1,\ldots,x_n) \leftarrow \exists y_1 \ldots \exists y_d (L_1 \land \ldots \land L_m)$$

Suppose that there are $r$ such clauses for predicate $p/n$

$$p(x_1,\ldots,x_n) \leftarrow E_1$$

$$\vdots$$

$$p(x_1,\ldots,x_n) \leftarrow E_r$$

where each $E_i (1 \leq i \leq r)$ stands for a corresponding formula of the form $\exists y_1 \ldots \exists y_d (L_1 \land \ldots \land L_m)$. Then the completed definition of the predicate $p/n$ is the formula

$$\forall x_1 \ldots \forall x_n (p(x_1,\ldots,x_n) \leftarrow E_1 \lor \ldots \lor E_r)$$

Let $p_1,\ldots,p_k$ be all the predicate symbols which appear in the head atoms of program clauses in $Pr$. The completion of program $Pr$ denoted by $\text{comp}(Pr)$ is the set of the completions of $p_1,\ldots,p_k$.

Note that this definition of completion is slightly different from the standard one in [12], [56]. We do not add in the $\text{comp}(Pr)$ negative unit clauses for undefined predicates in $Pr$ because these predicates are assumed to be implemented by DT operations. We also omit the equality theory, which is assumed to be part of the theory of DTs.

**Definition 5.3.2:** Let $Pr$ be an SI-program, excluding the DT definitions. Let $A$ be the theory of underlying DTs including the specifications of the DT operations. Then the meaning $\text{Prog}$ of an SI-program is defined as follows.

$$\text{Prog} = \text{comp}(Pr) \cup A$$
Definition 5.3.3: The specification of an SI-program with top level predicate $p$ is a set of formulas including one of the form

$$\forall \bar{x}/\bar{t} (p^S(\bar{x}) \rightarrow \text{Def}_p)$$

Definition 5.3.4: Let $Pr$ be an SI-program with top-level predicate $p$ excluding DT definitions, and $Spec$ its specification. $p$ does not occur in $Spec$ and $p^S$ does not occur in $\text{comp}(Pr)$. $Pr$ is partially correct with respect to $Spec$ if

$$Spec \cup \mathcal{A} \cup \text{comp}(Pr) \models \forall \bar{x}/\bar{t} (p(\bar{x}) \rightarrow p^S(\bar{x}))$$

$Pr$ is complete with respect to $Spec$ if

$$Spec \cup \mathcal{A} \cup \text{comp}(Pr) \models \forall \bar{x}/\bar{t} (p(\bar{x}) \rightarrow p^S(\bar{x}))$$

$Pr$ is totally correct with respect to $Spec$ if it is both partially correct and complete, i.e.

$$Spec \cup \mathcal{A} \cup \text{comp}(Pr) \models \forall \bar{x}/\bar{t} (p(\bar{x}) \rightarrow p^S(\bar{x}))$$

Note that this is equivalent to

$$Spec \cup \mathcal{A} \cup \text{comp}(Pr) \models \forall \bar{x}/\bar{t} (p(\bar{x}) \rightarrow \text{Def}_p)$$

Note that the usefulness of this definition depends on the assumption that $\text{comp}(Pr)$ is consistent, since otherwise, every formula is a consequence of $\text{comp}(Pr)$. We conjecture that SI-programs are stratified (assuming that the data type definitions are stratified), and hence consistent [56].

5.3.1 Examples

Example 1: The predicate $\text{sum}(q,s)$ where $\text{Type}(\text{sum}) = \text{seq}(\mathbb{Z}) \times \mathbb{Z}$ is true iff $s$ is the sum of the sequence of integers $q$.

1. Logic specification for relation $\text{sum}^S(q,s)$ ($Spec$):

$$Spec = \forall q/\text{seq}(\mathbb{Z}), s/\mathbb{Z} (\text{sum}^S(q,s) \rightarrow s = \sum_{i=1}^n q_i)$$
2. Axioms - Lemmas - Specifications of DT operations (A):

Axioms

A1 Domain closure axiom for sequences
\[ \forall s/\text{seq}(\alpha) \ (s =<> \lor \exists h/\text{seq}(\alpha) \ s = h \cdot t) \]

A2 Uniqueness axioms for sequences
i. \[ \forall h/\alpha, t/\text{seq}(\alpha) \ (\neg (h \cdot t =<>)) \]
ii. \[ \forall 1, h2/\alpha, t1, t2/\text{seq}(\alpha) \ (h1 \cdot t1 = h2 \cdot t2 \rightarrow (h1 = h2 \land t1 = t2)) \]

A3 Definition of summation operation over 0 entities
\[ \forall s/\text{seq}(\mathbb{Z}) (s =<> \rightarrow \sum_{i=1}^n s_i = 0) \]

Lemmas

L1 \[ \forall s/\text{seq}(\alpha) (s \neq<> \rightarrow \exists h/\alpha, t/\text{seq}(\alpha) \ s = h \cdot t) \]

L2 \[ \forall h/\alpha, s, t/\text{seq}(\alpha) (s = h \cdot t \rightarrow \forall i/N (2 \leq i \leq \#s \rightarrow s_i = t_{i-1})) \]

L3 \[ \forall s, t/\text{seq}(\alpha), h/\alpha (s = h \cdot t \rightarrow \#s = \#t + 1) \]

L4 \[ \forall h/\alpha, s, t/\text{seq}(\alpha) (s = h \cdot t \rightarrow h = s) \]

Logic specifications of DT operations

\[ \forall q/\text{seq}(\alpha) (\text{empty}_\text{seq}(q) \leftarrow q =<> ) \]
\[ \forall s/\mathbb{Z} (\text{neutral}_\r add._\text{subtr}_\cdot \text{int}(s) \leftarrow s = 0) \]
\[ \forall q/\text{seq}(\alpha), h/\alpha (\text{head}(q, h) \leftarrow q \neq<> \land \exists t/\text{seq}(\alpha) \ q = h \cdot t) \]
\[ \forall q, t/\text{seq}(\alpha) (\text{tail}(q, t) \leftarrow q \neq<> \land \exists h/\alpha \ q = h \cdot t) \]
\[ \forall s1, h, s/\mathbb{Z} (\text{plus}_\text{int}(s1, h, s) \leftarrow s = h + s1) \]

3. Constructed SI-program without the definitions of DT operations (Pr):

\[ \text{sum}(q, s) \leftarrow p1(q) \land p2(q, s) \]
\[ \text{sum}(q, s) \leftarrow \neg p1(q) \land p3(q, h, t) \land \text{sum}(t, s1) \land p4(q, h, s1, s) \]
\[ p1(q) \leftarrow \text{empty}_\text{seq}(q) \]
\[ p2(q, s) \leftarrow \text{neutral}_\r add._\text{subtr}_\cdot \text{int}(s) \]
\[ p3(q, h, t) \leftarrow p5(q, h, t) \land p6(q, h, t) \]
\[ p5(q, h, t) \leftarrow \text{head}(q, h) \]
\[ p6(q, h, t) \leftarrow \text{tail}(q, t) \]
\[ p4(q, h, s1, s) \leftarrow \text{plus}_\text{int}(s1, h, s) \]

4. Completion of the logic program \text{sum}(q, s) (comp(Pr)):

\[ \forall q/\text{seq}(\mathbb{Z}), s/\mathbb{Z} (\text{sum}(q, s) \leftarrow (p1(q) \land p2(q, s)) \lor \]
\[ (\exists h/\mathbb{Z}, t/\text{seq}(\mathbb{Z}), s1/\mathbb{Z} (\neg p1(q) \land \]
\[ p3(q, h, t) \land \text{sum}(t, s1) \land p4(q, h, s1, s)))) \]
\[ \forall q/\text{seq}(Z) \ (p1(q) \rightarrow \text{empty}_\text{seq}(q)) \]
\[ \forall q/\text{seq}(Z), s/Z \ (p2(q, s) \rightarrow \text{neutral}_\text{add}_\text{subtr}_\text{int}(s)) \]
\[ \forall q/\text{seq}(Z), h/Z, t/\text{seq}(Z) \ (p3(q, h, t) \leftrightarrow p5(q, h, t) \land p6(q, h, t)) \]
\[ \forall q/\text{seq}(Z), h/Z, t/\text{seq}(Z) \ (p5(q, h, t) \leftrightarrow \text{head}(q, h)) \]
\[ \forall q/\text{seq}(Z), h/Z, t/\text{seq}(Z) \ (p6(q, h, t) \rightarrow \text{tail}(q, t)) \]
\[ \forall q/\text{seq}(Z), h, s1, s/Z \ (p4(q, h, s1, s) \leftrightarrow \text{plus}_\text{int}(s1, h, s)) \]

5. Correctness theorem and theory to prove it.

\[ \text{comp}(Pr) \cup \text{Spec} \cup A \models \forall q/\text{seq}(Z), s/Z \ (\text{sum}(q, s) \leftrightarrow \text{sum}^S(q, s)) \]

### 5.4 General Scheme to Prove Correctness

Suppose that we have formulated a logic specification for an SI-program. How will we proceed to prove the correctness of the program with respect to its specification?

1. First try to manipulate the specification into a suitable form.

2. Try to apply the proof scheme corresponding to the design schema applied to the top-level predicate.

The "suitable form" is often a "structural decomposition" of the specification. This will be discussed in more details later. "Proof schemes" will be defined corresponding to the design schemas used in the program construction. In this way, the structure of the SI-programs can be exploited when finding a proof. In the future, this may assist an automatic proof tool to generate proofs.

A proof scheme is a systematic plan of proof actions. The proof scheme of the schema which has been applied for the instantiation of the top-level predicate is followed. The proof proceeds in a top-down fashion. If a program has nested instances of schemata the correctness proof may require the proof of some correctness theorems for predicates in lower levels. For such correctness theorems the proof scheme of the schema which has been applied for the instantiation of the predicate in the correctness theorem is followed.

The proof schemes given below generally provide sufficient conditions for proving correctness. They are not intended to be complete. However, many examples of correctness proofs for which these schemes apply have been constructed (see Section 5.7 and Appendix C).
5.5 Schemata-Based Proof Schemes

The correctness of the constructed SI-programs requires the establishment of the equivalence $orall x (p(x) \leftrightarrow \text{Def} p)$ in the theory $\text{comp}(Pr) \cup \text{Spec} \cup A$. The correctness proof method for establishing this equivalence is guided by the design schemata that have been applied to construct the logic program $Pr$. Note that $Pr$ is an SI-program excluding the definitions of the DT operations. Each design schema is associated with a correctness proof scheme. The proof scheme that corresponds to the design schema which has been applied for the definition of the top level predicate $p$ is initially followed.

It is assumed that the implementations of the DT operations satisfy their logic specifications. Otherwise the correctness of the DT specifications would have to be proved as well. In the following, variables are not accompanied by types for simplicity of presentation.

5.5.1 Proof Scheme for Subgoal Schema

If the Subgoal schema has been applied to construct the top-level predicate $p$, then the correctness proof is of the form

$$\forall \bar{x} (p(\bar{x}) \leftrightarrow \text{Def} p)$$

where $\forall \bar{x} (p(\bar{x}) \leftrightarrow \exists \bar{y} (F_1 \land \ldots \land F_k))$ and $F_1, \ldots, F_k$ are first-order formulas. It is assumed that the arguments $\bar{x}$ are identical in each of the above formulas. In this case, perform the following steps, which are sufficient to establish the correctness result.

1. Try to reformulate $\text{Def} p$ as an equivalent formula of the form $\exists \bar{z} (G_1 \land \ldots \land G_k)$ where $\bar{z}$ is of the same type as $\bar{y}$ in $\exists \bar{y} (F_1 \land \ldots \land F_k)$.

2. Try to prove $\forall \bar{x}, \bar{y} (F_1 \leftrightarrow G_1[\bar{z}/\bar{y}]), \ldots, \forall \bar{x}, \bar{y} (F_k \leftrightarrow G_k[\bar{z}/\bar{y}])$.

5.5.2 Proof Scheme for Case Schema

If the Case schema has been applied to construct the top-level predicate $p$, then correctness proof is of the form
\[ \forall \bar{x} \ (p(\bar{x}) \leftrightarrow \text{Def}_p) \]

where \( \forall \bar{x} \ (p(\bar{x}) \rightarrow \exists \bar{y} \ (F_1 \lor \ldots \lor F_k)) \) and \( F_1, \ldots, F_k \) are first-order formulas. It is assumed that the arguments \( \bar{x} \) are identical in each of the above formulas. In this case, perform the following steps, which are sufficient to establish the correctness result.

1. Try to reformulate \( \text{Def}_p \) as an equivalent formula of the form \( \exists \bar{z} \ (G_1 \lor \ldots \lor G_k) \) where \( \bar{z} \) is of the same type as \( \bar{y} \) in \( \exists \bar{y} \ (F_1 \lor \ldots \lor F_k) \).
2. Try to prove \( \forall \bar{x}, \bar{y} \ (F_1 \rightarrow G_1[\bar{x}/\bar{y}]), \ldots, \forall \bar{x}, \bar{y} \ (F_k \rightarrow G_k[\bar{x}/\bar{y}]) \).

### 5.5.3 Proof Scheme for Incremental and Divide-and-conquer Schemata

If the Incremental or Divide-and-conquer schema has been applied to construct \( p(\bar{x}) \) then we apply structural induction on a well-founded set.

**Definition 5.5.1:** Let \( \prec \) be a binary relation. A partially-ordered set \( (S, \prec) \) in which every nonempty subset has a minimal element is called a well-founded set [55], [58].

Let \( (S, \prec) \) be a well-founded set and \( P \) be a proposition over \( S \). If the following two conditions hold,

1. **Base case:** \( P(x) \) is true for each minimal element of \( S \).
2. **Induction step:** \( \forall x \in S \ (\forall y \in S \ (y \prec x \rightarrow P(y)) \rightarrow P(x)) \)

then infer that \( \forall z \in S \ P(z) \). This is the *principle of structural induction*.

Note that the principle of structural induction is sometimes expressed only as the condition of the induction step because the condition of the base case is redundant. That is, if \( x \) is a minimal element of \( S \) then there is no element in \( S \) less than \( x \). So it must be shown that \( P(x) \) is true. The principle of structural induction is expressed in this thesis by using two condition statements for reasons of clarity, i.e. this pattern has to be followed in proofs.

The *principle of structural induction* is a generalization of mathematical induction for domains which consist of well-founded sets. Burstall first applied structural induction for proving properties of programs [10].

Let \( S_1 \) be a set, and let \( \mu \) be a mapping from \( S_1 \) to some well-founded set \( (S_2, \prec) \). \( \mu \) induces a well-founded ordering on \( S_1 \), i.e. for all \( u_1, u_2 \in S_1 \) \( u_1 \prec u_2 \) iff \( \mu(u_1) \prec \mu(u_2) \).
Let \( \forall \bar{u} (p(u, \bar{v}) \iff \text{Def} p) \) be the correctness theorem to be proved, abbreviated by \( \Phi(u) \). 

\( P(x) \) in the structural induction schema can now be instantiated by \( \Phi(u) \) where \( u \) is of type \( \tau \), and a well-founded ordering \( \prec \) has been defined on \( \tau \) by mapping \( \tau \) to some well-founded set.

The above induction scheme is used as the correctness proof method for the Incremental and Divide_and_conquer schemata. The following proof schemes for the base case and induction step of the above induction schema show how the proof can proceed.

In order to prove \( \forall u, \bar{v} (p(u, \bar{v}) \iff \text{Def} p) \) where \( p(u, \bar{v}) \) has been defined by an Incremental or Divide_and_conquer schema, perform the following steps. In the following, \( \bot \) represents a minimal element of the well-founded set \( (S, \prec) \).

**Base case:** \( u = \bot \)
\[
\forall u, \bar{v} (u = \bot \rightarrow (p(u, \bar{v}) \iff \text{Def} p))
\]

**Induction hypothesis:**
\[
\forall u \text{ Ind}(u) \equiv \forall u, u', \bar{v}' (u' \prec u \rightarrow (p(u', \bar{v}') \iff \text{Def} p))
\]

**Induction step:** \( u \neq \bot \)
\[
\forall u, \bar{v} (u \neq \bot \land \text{Ind}(u) \rightarrow (p(u, \bar{v}) \iff \text{Def} p))
\]

### 5.5.4 Proof Scheme for Search Schema

If the Search schema has been applied to construct \( p(x) \) its completion has the following form.

\[
\forall \bar{x} (p(\bar{x}) \iff \exists \bar{y}, w(F_1 \land F_2(w)))
\]

where \( F_2(w) \) is an atom containing a recursive predicate (the instantiation of the predicate variable Search1) and \( w \) is the argument obtained from the schema argument variable \( v_1 \) in the first clause of the search schema, which will be the search stack (see Section 3.4).

This has the same form as the completed definition of an instance of Subgoal, and the same proof scheme can be applied. However, the proof of correctness of the definition of \( F_2(w) \) has to be considered as well, since the Search schema provides a definition of \( F_2(w) \).

Suppose the specification of \( p \) is
∀\overline{x}(p(\overline{x}) \rightarrow Def_{p})

where the variables \overline{z} are identical to \overline{x} in the formula defining \(p(\overline{x})\) given above. Then try the following steps.

1. Try to transform \(Def_{p}\) into an equivalent formula \(\exists \overline{x}, w'(G_1 \land G_2(w'))\) where \(w'\) is of the same type as \(w\).

2. Try to prove \(\forall \overline{x}, \bar{y}(F_1 \rightarrow G_1[\overline{x}/\overline{y}])\)

3. Try to prove \(\forall \overline{x}, \bar{y}, w(F_2(w) \rightarrow G_2(w)[\overline{z}/\overline{y}])\)

These steps correspond to the Subgoal proof scheme. Now, to carry out step 3, a structural induction proof is appropriate. The induction argument is \(w\), which will normally be a search stack.

The well-founded ordering on the search stack will typically be a lexicographical ordering on the sequence of elements in the stack (starting from the bottom of the stack). The lexicographical ordering is induced by the well-founded ordering on the stack elements, which corresponds to the order in which the nodes of the search space will be visited, with the greatest nodes visited first. This is well-founded provided that the search space is finite.

Forward moves reduce the stack with respect to the lexicographical ordering since the top element is replaced by smaller elements. Backward moves reduce the stack with respect to the ordering since the top element is either removed or replaced by a smaller element. Hence the instance of the \textit{Search1} literal schema, namely \(F_2(w)\), has an inductive structure based on the ordering on its stack argument \(w\).

To return to step 3, the formula to be proved is

\[\forall \overline{x}, \bar{y}, w (F_2(w) \rightarrow G_2(w)[\overline{z}/\overline{y}])\]

Following the induction proof scheme, with induction parameter \(w\), the steps of the proof are as follows.

1. \textit{Base case}: Show that \(\forall (w = \bot \rightarrow (F_2(w) \rightarrow G_2(w)[\overline{z}/\overline{y}]))\)

2. \textit{Induction hypothesis}:
   \[\forall w \text{ Ind}(w) = \forall \bar{y}, w, w' (w' < w \rightarrow (F_2(w') \rightarrow G_2(w')[\overline{z}/\overline{y}]))\]

3. \textit{Induction step}: Show that \(\forall (w \neq \bot \rightarrow (\text{Ind}(w) \rightarrow (F_2(w) \rightarrow G_2(w)[\overline{z}/\overline{y}]))\)

117
Example 1: The predicate $fib(n, f)$ is true iff $f$ is the $n^{th}$ Fibonacci number. The SI-program for this predicate has been constructed using the Search schema. $fib(n, f)$ could also be implemented using Divide-and-conquer, but the implementation using Search is convenient here to illustrate the proof scheme. Here we will discuss the main points of the correctness proof. Note that stacks are implemented by sequences in this example, i.e. $stack(N) = seq(N)$.

The instance of the Search schema in the Fibonacci program is as follows. The modes of the predicates are omitted because they are not needed in this discussion.

Signatures

$Type(fib) = N \times N$
$Type(p) = N \times seq(N) \times N$
$Type(fib1) = N \times seq(N) \times N \times N$
$Type(s) = seq(N) \times N$
$Type(t) = N \times N$
$Type(q) = N \times seq(N) \times N \times seq(N) \times N$
$Type(r) = seq(N) \times N \times seq(N) \times N$

Clauses

$fib(n, f) \leftarrow p(n, s, f0) \land fib1(n, s, f0, f)$
$fib1(n, s, f0, f) \leftarrow s(s, f0) \land t(f0, f)$
$fib1(n, s, f0, f) \leftarrow \neg s(s, f0) \land q(n, s, f0, s1, f1) \land$
$\quad fib1(n, s1, f1, f)$
$fib1(n, s, f0, f) \leftarrow \neg s(s, f0) \land \neg q(n, s, f0, s1, f1) \land$
$\quad r(s, f0, s2, f2) \land fib1(n, s2, f2, f)$

The forward and backward moves are performed by the predicates $q/5$ and $r/4$ respectively.

The search space for this problem is a binary tree. The binary tree rooted at node $n$ represents the search space for the Fibonacci number of $n$. Its left and right children have nodes $n - 1$ and $n - 2$ respectively. They represent the search space for the Fibonacci numbers of $n - 1$ and $n - 2$ respectively. The leaves of this binary tree consist of the basic Fibonacci numbers, that is either 0 or 1. Figure 5.1 shows the search space for computing the call $fib(5, f)$. The set of values of $seq(N)$ is totally ordered by the lexicographical ordering on sequences (where the elements of a sequence are taken in reverse order). The sequence of stacks that occur for this problem if $n = 5$ and the order in which they occur is shown below. An index $f$ or $b$ in each stack except the first one indicates whether that stack has been created by a forward or by a backward move respectively.
Figure 5.1: Search space for constructing the Fibonacci number for $n = 5$.

\[ \langle 5 >, < 4, 3 >, < 3, 2, 3 >, < 2, 1, 2, 3 >, < 1, 0, 1, 2, 3 >, < 0, 1, 2, 3 >, b, < 1, 2, 3 >, < 2, 3 >, b, < 1, 0, 3 >, < 0, 3 >, b, f, < 2, 1 >, < 1, 0, 1, 2, 3 >, < 2, 1 >, b, < 1 > > \]

For example $\langle 1, 0, 1, 2, 3 \rangle < \langle 2, 1, 2, 3 \rangle$ since the sequence 32101 precedes 3212 in the lexicographical order.

The nodes on the stack at any stage represent these nodes in the search space which have not yet been visited and hence whose successors still need to be visited. For example, the stack $\langle 5 \rangle$ represents the whole binary tree for constructing the Fibonacci number $f$ for $n = 5$.

Note that the logic specification which has been used for this example is as follows.

\[
\forall n,f/N \ (\text{fib} s(n,f) \iff \exists f0/N,s/seq(N) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (n \geq 0 \land f0 = 0 \land s = n::<> \land \text{fib} s(s,f0,f)))
\]

\[
\forall s/seq(N),f0,f/N \ (\text{fib} s(s,f0,f) \iff (s = <> \land f0 = f) \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\exists t/seq(N),h,f1/N (s = h::t \land ((h = 0 \lor h = 1) \land f1 = h + f0 \land \text{fib} s(t,f1,f)) \lor \exists q,t1/seq(N),h1,h2/N (h > 1 \land q = h1::t1 \land t1 = h2::t \land h1 = h - 1 \land h2 = h - 2 \land \text{fib} s(q,f0,f)))))
\]
5.6 Application of the Proof Schemes

5.6.1 Structured Form of Specifications

A logic specification for predicate $p^S(x)$ can be expressed in several different ways. Which specification is appropriate for proving the correctness of the constructed SI-program? It is well-known that correctness proofs even for small programs are complex and long. A possible way to overcome this problem is by expressing logic specifications in a structured form suitable for applying the proof scheme. A form which often facilitates the correctness proof of an SI-program is the structured form of $p^S(x)$.

The structured form of a specification is built around the structural cases of a parameter as follows. Let $\tau$ be a type and let $x/\tau$. Suppose that there exists a finite set of constructors $f_1/n_1, \ldots, f_k/n_k$ such that for all $x/\tau$, $x = f_1(y_1, \ldots, y_{n_1}) \vee \ldots \vee x = f_k(z_1, \ldots, z_{n_k})$. The specification of a predicate $p$ consists of a disjunction of subformulas $F_1(x) \vee \ldots \vee F_d(x)$. Each subformula $F_i(x)$ ($1 \leq i \leq d$) has the form $S \wedge G$ where $S$ is of the form $x = f_j(y_1, \ldots, y_{n_j})$ ($1 \leq j \leq k$) and $G$ is either a conjunction of literals or a disjunction of conjunctions of literals. In addition, $x$ can be used in the correctness proofs as the induction parameter.

The structured forms of logic specifications in the next examples correspond to logic specifications shown in Section 5.2.

Example 1: Specification in structured form for predicate $\text{sum}^S(q, s)$:

$$\forall q/\text{seq}(Z), s/Z \ (\text{sum}^S(q, s) \leftarrow (q = <> \wedge s = 0) \vee \\
\exists s1, h/Z, t/\text{seq}(Z) \ (q = h :: t \wedge s = h + s1 \wedge \text{sum}^S(t, s1)))$$

Example 2: Logic specification in structured form for predicate $\text{fac}^S(n, f)$:

$$\forall n, f/N \ (\text{fac}^S(n, f) \leftarrow (n = 0 \wedge f = 1) \vee \\
\exists n1, f1/N \ (n = \text{succ}(n1) \wedge f = n \times f1 \wedge \text{fac}^S(n1, f1)))$$

Example 3: Logic specification in structured form for predicate $\text{incrOrd}^S(q)$:

$$\forall s/\text{seq}(\alpha) \ (\text{incrOrd}^S(s))$$
The logic specifications and their corresponding structured forms are the same for the next examples.

**Example 4:** The predicate \( \text{fib}(n, f) \) where \( \text{Type}(\text{fib}) = N \times N \) is true iff \( f \) is the \( n \)-th Fibonacci number.

\[
\forall n, f/N (\text{fib}(n, f) \leftarrow (n = 0 \land f = 0) \lor \\
\exists f_1, f_2, k/N (n = \text{succ}(k) \land ((k = 0 \land f = n) \lor \exists m/N (k = \text{succ}(m) \land \\
\text{fib}(k, f_1) \land \text{fib}(m, f_2) \land f = f_1 + f_2)))
\]

**Example 5:** The predicate \( \text{isEven}(s) \) where \( \text{Type}(\text{isEven}) = \text{seq}(\alpha) \) is true iff the length of the sequence \( s \) is even.

\[
\forall s/\text{seq}(\alpha) (\text{isEven}^S(s) \leftarrow \exists n/N (\text{length} \cdot \text{seq}(s, n) \land \text{even}^S(n))) \\
\forall n/N (\text{even}^S(n) \leftarrow (n = 0 \lor \exists m/N (n = \text{succ}(m) \land \\
((m = 0 \land \text{false}) \lor \exists n_1/N (m = \text{succ}(n_1) \land \text{even}^S(n_1))))))
\]

The disjunction \( \exists m/N n = \text{succ}(m) \land (m = 0 \land \text{false}) \) in the definition of the relation \( \text{even}^S(n) \) is omitted because it is \( \text{false} \). That is,

\[
\forall s/\text{seq}(\alpha) (\text{isEven}^S(s) \leftarrow \exists n/N (\text{length} \cdot \text{seq}(s, n) \land \text{even}^S(n))) \\
\forall n/N (\text{even}^S(n) \leftarrow (n = 0 \lor \exists m/N, n_1/N (n = \text{succ}(m) \land m = \text{succ}(n_1) \land \\
\text{even}^S(n_1))))
\]

The first stage in a correctness proof is often to transform a specification into a structured form. The structured forms in the examples above are derived from the corresponding initial logic specifications. These derivations are shown in Section 4.4 and in Appendix C.

### 5.6.2 Equivalence Preserving Transformations

In the proof schemes, no suggestions were made above about proving the individual steps of the schemes. In the examples, the correctness theorem is proved by performing equivalence preserving transformations. Let us assume that the correctness theorem has the following form at some point during the transformation.
Let us assume that equivalence preserving transformations have been performed on either or on both of the formulas \( F_1 \) and \( G_1 \), i.e. \( F_1 \rightarrow F_2 \) and \( G_1 \rightarrow G_2 \). The correctness theorem is transformed by these transformations into an equivalent form. That is,

\[
\forall x \ (F_1 \equiv G_1) \equiv \forall x \ (F_2 \equiv G_2)
\]

The process is continued until a formula \( \forall x \ (F \equiv F) \) is reached.

Two examples of correctness proofs are discussed in this section. The logic specification \( \text{Spec} \) of the first example is first transformed into its structured form \( \text{Spec}' \) then its correctness theorem is proved with respect to \( \text{Spec}' \). The logic specification of the second example is already in structured form. The Subgoal proof scheme is immediately applicable to the correctness theorem.

**Example 1:** The logic specification \( \text{Spec} \) for the predicate \( \text{sum}(q,s) \) where \( \text{Type}(\text{sum}) = \text{seq}(Z) \times Z \) is as follows:

\[
\forall q/\text{seq}(Z), s/Z (\text{sum}(q,s) \leftrightarrow s = \sum_{i=1}^{q} q_i)
\]

The logic specification \( \text{Spec} \) is transformed into its structured form \( \text{Spec}' \). That is,

\[
\forall q/\text{seq}(Z), s/Z (\text{sum}(q,s) \leftrightarrow (q =<> s = 0) \\
\quad \exists s1, h/Z, t/\text{seq}(Z) (q = h :: t \land s = h + s1 \land \text{sum}(t, s1))
\]

Next, the \( \text{Spec} \) theory is replaced by theory \( \text{Spec}' \). The Incremental proof scheme is followed in order to prove its correctness theorem. That is,

\[
\forall q/\text{seq}(Z), s/Z (\text{sum}(q,s) \leftrightarrow \text{sum}(q,s))
\]

**Example 2:** The logic specification of the predicate \( \text{isEven}(s) \) where \( \text{Type}(\text{isEven}) = \text{seq}(\alpha) \) is as follows:

\[
\forall s/\text{seq}(\alpha) (\text{isEven}(s) \leftrightarrow \exists n/N (\text{length}_\text{seq}(s, n) \land \text{even}(n))) \\
\forall n/N (\text{even}(n) \leftrightarrow (n = 0 \lor \exists m/N, n1/N (n = \text{succ}(m) \land m = \text{succ}(n1) \land \text{even}(n1))))
\]
Its correctness theorem is proved by following the Subgoal proof scheme. That is,

$$\forall s/seq(\alpha) (isEven(s) \leftrightarrow isEven^S(s))$$

This correctness theorem is transformed into the following form at some stage during the proof.

$$\forall s/seq(\alpha) (\exists n/N (length\_seq(s, n) \land p2(s, n)) \leftrightarrow \exists n/N (length\_seq(s, n) \land even^S(n)))$$

In order to establish the above equivalence the next correctness theorem has to be proved. That is,

$$\forall s/seq(\alpha), n/N (p2(s, n) \leftrightarrow even^S(n))$$

This correctness theorem is used as lemma in the previous proof. Incremental proof scheme is applied to this correctness theorem because the clauses of the predicate $p2(s, n)$ where $Type(p2) = seq(\alpha) \times N$ are an instance of Incremental schema. The well-ordering is based on $n/N$.

### 5.7 Examples of Correctness Proofs

The theory of the example of this section is presented in Section 5.3.

**Example 1:** The predicate $sum(q, s)$ where $Type(sum) = seq(Z) \times Z$ is true iff $s$ is the sum of the sequence of integers $q$.

1. Correctness theorem:

$$\forall q/seq(Z), s/Z (sum(q, s) \leftrightarrow sum^S(q, s))$$

2. Correctness proof: Transform logic specification into structured form.

$$\forall q/seq(Z), s/Z (sum^S(q, s) \leftrightarrow s = \sum_{i=1}^{\#q} q_i)$$
by axiom \textbf{A1} and by FOL law (if $Q \rightarrow true$ then $P \land Q \rightarrow P$)

$$
(q = <> \lor \exists h/Z, t/seq(Z) q = h :: t) \land s = \sum_{i=1}^{#q} q_i)
$$

by FOL ($\land$ distribution)

$$
(q = <> \land s = \sum_{i=1}^{#q} q_i) \lor \exists h/Z, t/seq(Z)(q = h :: t \land s = \sum_{i=1}^{#q} q_i))
$$

by axiom \textbf{A3}

$$
(q = <> \land s = 0) \lor \exists h/Z, t/seq(Z)(q = h :: t \land s = q_1 + \sum_{i=2}^{#q} q_i))
$$

by introduction of new variable $s1/Z$ and by FOL law

$$
(\forall x (P(x) \rightarrow \exists y (x = y \land P(y))))
$$

$$
(q = <> \land s = 0) \lor 
\exists s1, h/Z, t/seq(Z) (q = h :: t \land s = q_1 + s1 \land s1 = \sum_{i=2}^{#q} q_i))
$$

by lemmas \textbf{L2} and \textbf{L3}

$$
(q = <> \land s = 0) \lor 
\exists s1, h/Z, t/seq(Z) (q = h :: t \land s = q_1 + s1 \land s1 = \sum_{i=1}^{#t} t_i))
$$

by folding

$$
(q = <> \land s = 0) \lor 
\exists s1, h/Z, t/seq(Z) (q = h :: t \land s = q_1 + s1 \land \text{sum}^S(t, s1))
$$

by lemma \textbf{L4}

$$
(q = <> \land s = 0) \lor 
\exists s1, h/Z, t/seq(Z) (q = h :: t \land s = h + s1 \land \text{sum}^S(t, s1))
$$

3. Proof: \textbf{Incremental proof scheme}

Structural induction is applied to the correctness theorem. That is, \(\forall q/\text{seq}(\alpha) \exists \#q/N\) such that \(\mu(q) = \#q\). \(\forall q_1, q_2/\text{seq}(\alpha) q_1 < q_2\) if \(\#q_1 < \#q_2\). The set \(\text{seq}(\alpha), <\) is well-founded.

\textit{Induction base:} \(q = <>\)

$$
\forall q/\text{seq}(Z) (q = <> \rightarrow \forall s/Z (\text{sum}(q, s) \rightarrow \text{sum}^S(q, s)))
$$

$$
\forall q/\text{seq}(Z), s/Z (\text{sum}(q, s))
$$

$$
(q = <> \land s = 0) \lor 
\exists s1, h/Z, t/seq(Z) (q = h :: t \land s = h + s1 \land \text{sum}^S(t, s1))
$$
because of the induction base \( q = <> \) and by axiom A2.i
\[ (q = <> \land s = 0) \lor 
\exists s_1, h/Z, t/seq(Z) (false \land s = h + s_1 \land sum^S(t, s_1)) \]

by FOL laws \( (P \land false \iff false) \)
\[ (q = <> \land s = 0) \lor false) \]

by FOL laws \( (P \lor false \iff P) \)
\[ q = <> \land s = 0) \]

because of the logic specifications of DT operations empty_seq/1 and neutral_add_subtr_int/1
\[ empty_seq(q) \land neutral_add_subtr_int(s) \]

because of the completion of p1/1 and p2/2
\[ p1(q) \land p2(q, s) \]

because of the completion of sum/2
\[ sum(q,s) \]

*Induction step: \( q \neq <> \)*
\[ \forall q/seq(Z) (q \neq <> \rightarrow \forall q1/seq(Z) (q1 < q \rightarrow
\forall s_1/Z (sum(q1,s1) \rightarrow sum^S(q1,s1)))
\rightarrow \forall s/Z (sum(q,s) \rightarrow sum^S(q,s))) \]

\[ \forall q/seq(Z), s/Z (sum(q,s)) \rightarrow (q = <> \land s = 0) \lor
\exists s_1, h/Z, t/seq(Z) (q = h \land s = h + s_1 \land sum^S(t, s_1)) \]

because of the induction step \( q \neq <> \), by axiom A2.i
and by lemma L1
\[ (false \land s = 0) \lor
\exists s_1, h/Z, t/seq(Z) (q = h \land s = h + s_1 \land sum^S(t, s_1)) \]

by FOL laws \( (P \land false \iff false) \)
\[ (false \lor \exists s_1, h/Z, t/seq(Z) (q = h \land s = h + s_1 \land sum^S(t, s_1))) \]

by FOL laws \( (P \lor false \iff P) \)
\[
\exists s_1, h/Z, t/seq(Z) \ (q = h :: t \ \land \ s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

by lemma L3 \( t < q \) and by induction hypothesis

\[
\exists s_1, h/Z, t/seq(Z) \ (q = h :: t \ \land \ s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

by lemma L1 and by FOL laws (if \( P \rightarrow Q \) then \( P \land Q \rightarrow P \))

\[
\exists s_1, h/Z, t/seq(Z) \ (q \neq <> \ \land \ q = h :: t \ \land \ s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

by FOL laws (\( P \land P \rightarrow P \))

\[
\exists s_1, h/Z, t/seq(Z) \ (q \neq <> \ \land \ q = h :: t \ \land \ s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

because of the logic specification of the DT operation \( \text{head}/2 \)

\[
\exists s_1, h/Z, t/seq(Z) \ (q \neq <> \ \land \ q = h :: t \ \land \ \text{head}(q, h) \land \\
\quad s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

by FOL laws (\( P \land P \rightarrow P \))

\[
\exists s_1, h/Z, t/seq(Z) \ (q = h :: t \ \land \ q \neq <> \ \land \ s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

because of the logic specification of the DT operation \( \text{tail}/2 \)

\[
\exists s_1, h/Z, t/seq(Z) \ (q \neq <> \ \land \ \text{tail}(q, t) \ \land \ \text{head}(q, h) \land \\
\quad s = h + s_1 \ \land \ \text{sum}(t, s_1))
\]

by FOL laws (\( \land \ \text{commutativity} \))

\[
\exists s_1/Z, t/seq(Z) \ (q \neq <> \ \land \ \text{head}(q, h) \land \text{tail}(q, t) \land \\
\quad \text{sum}(t, s_1) \land s = h + s_1)
\]

because of the logic specification of the DT operation \( \text{plus_int}/3 \)

\[
\exists s_1/Z, t/seq(Z) \ (q \neq <> \ \land \ \text{head}(q, h) \land \text{tail}(q, t) \land \\
\quad \text{sum}(t, s_1) \land \text{plus_int}(s_1, h, s))
\]

by FOL laws (\( \forall x, y \ (x \neq y \rightarrow \neg x = y) \))

\[
\exists s_1/Z, t/seq(Z) \ (-(q = <>) \ \land \ \text{head}(q, h) \land \\
\quad \text{tail}(q, t) \land \text{sum}(t, s_1) \land \text{plus_int}(s_1, h, s))
\]

because of the logic specifications of DT operations \( \text{empty_seq}/1 \)

\[
\exists s_1/Z, t/seq(Z) \ (-(\text{empty_seq}(q) \ \land \ \text{head}(q, h) \land \text{tail}(q, t) \land \\
\quad \text{sum}(t, s_1) \land \text{plus_int}(s_1, h, s))
\]

126
because of the completions of $p1/1, p4/4, p5/3$ and $p6/3$

$\rightarrow \exists s1/Z, t/seq(Z), h/Z (\neg p1(q) \land p5(q, h, t) \land p6(q, h, t) \land sum(t, s1) \land p4(q, h, s1, s))$

because of the completion of $p3/3$

$\rightarrow \exists s1/Z, t/seq(Z), h/Z (\neg p1(q) \land p3(q, h, t) \land sum(t, s1) \land p4(q, h, s1, s))$

because of the completion of $sum/2$

$\rightarrow sum(q, s)$
Chapter 6

System Implementation

6.1 Introduction

In this chapter the implementation of a program development system called SIDS (SI-Program Development System) is described. The SIDS system allows a user to construct an SI program using schema refinements and data type refinements. This chapter also contains a discussion of the use of partial evaluation in order to improve the efficiency of the program following construction.

As discussed in Chapter 3, a refinement tree represents refinements to undefined predicates for the complete or partial construction of an SI-program. The first refinement is applied to an initial "top" undefined predicate. The purpose of the implementation is to convert the representation of a refinement tree into (the representation of) an SI-program, by carrying out the refinements specified in the refinement tree. The input to SIDS is thus the representation of a refinement tree supplied by the user, and the output is an SI-program.

The usage of the system assumes that the user provides for either partial or complete details of the refinements and expresses them as a sequential representation of a refinement tree. We have introduced the idea of sequential construction of a refinement tree in order to reduce interaction for the more experienced users. The sequential representation of a refinement tree allows the construction of an SI-program either in one attempt or interactively. At one
extreme a refinement tree can be created with just the refinement of the top node and then further refinements can be added incrementally. At the other end, a complete refinement tree can be defined thus creating the whole SI-program. The more experienced users can create larger refinement trees. Once a sequential (partial or complete) refinement tree is constructed it can be replayed several times with the benefit of minimal interaction. Tests with alternative individual refinements and subsequences of refinements can be done with less interaction by using the sequential representation of refinement trees rather than using a system where the user selects one-by-one the refinements and the system performs them. It should be noted that a partial SI-program is constructed from a partial refinement tree whose any subtrees can be unrefined.

The main components of SIDS and a schematic view of the construction process are presented in Section 6.2. In Section 6.3 we discuss the representation of the design knowledge. In Section 6.4 we discuss the representation and the interpretation of refinement trees. In Section 6.5 the representation of SI-programs is presented. Finally, the efficiency of SI-programs is discussed in Section 6.6.

6.2 System Components

The aim of this section is to present the components of SIDS and to show a schematic view of the construction process.

The input to SIDS is a refinement tree and its output is an SI-program. The functional description of each component is as follows.

1. **Schema refinements**: This component performs the schema refinements. An interpreter for refinement trees successively interprets the representations of the nodes of a refinement tree. This component performs the tupling operation, the schema instantiation and the flattening operation for each schema refinement. In addition, it derives the expected modes of the new undefined predicates in the instances of the schemata. It also verifies that the types and modes of the DT operations and equality predicate of the refinement tree are consistent with the declared ones. This module also constructs the representation of the SI-program.

2. **Argument matching and DT refinements**: In this component the user interactively matches the arguments of the DT operations or equality with the ones of the predicate they refine. DT refinements are also performed by this component. That is, the basic clauses are constructed and the types of the matched arguments are checked.
3. **Type propagation**: This component propagates types from the predicates refined in the leaves of the refinement tree up to the top-level predicate. That is, the type substitution of each basic clause is applied to the type substitutions of the predicates which are represented in the other leaf and non-leaf nodes of the refinement tree.

4. **Mode analysis**: This component performs mode analysis on a partial or on a complete SI-program. These modes support the argument matching process. In addition, the modes of the predicates in the SI-program are also derived by this component.

5. **Program schemata knowledge base (KB)**: Knowledge base which consists of the program schemata.

6. **DT operations and equality KB**: Knowledge base which consists of DT operations and the equality predicate.

The main components of the system and a schematic view of the construction process are shown in figure 6.1. Dashed arrows show access to a KB. Arrows without label stand for the flow of the representation of the SI-program. Sample sessions of the system are shown in Appendix E.
6.3 Representation of Design Knowledge

Two linguistic levels are seen in this description framework. They are depicted in Figure 6.2. The object level consists of logic programs. The meta level contains representations of program schemata, DT operations and refinement trees.

This section discusses the representation of design knowledge. That is, program schemata and DT operations. A ground representation [35] is used for representing it. The interpretation of a refinement tree uses this representation of design knowledge.

6.3.1 Representation of Program Schemata

The program schemata knowledge base consists of ground positive unit clauses of the following form.

```
schema(Schema_name, [Index1, Index2, ...]).
  :
schema_clause(Index1, [Head_pred1, Body_literal1a, Body_literal1b, ...]).
schema_clause(Index2, [Head_pred2, Body_literal2a, Body_literal2b, ...]).
  :
```

There are three kinds of variables, predicate variables, schema argument variables and parameter variables. Predicate variables, schema argument variables and parameter variables are represented by ground terms of the form $pvar(Id)$, $var(Id)$ and $tvar(Id)$ respectively.
where \( \text{Id} \) is a positive integer. Each schema argument variable with the corresponding type and expected mode are represented by a ground term of the following form.

\[
\text{arg(var}(\text{Id}_1), \text{Mode}, \text{tvar}(\text{Id}_2))
\]

where \( \text{Mode} \) is the expected mode of the argument, \( \text{Id}_1 \) and \( \text{Id}_2 \) stand for positive integers.

**Example 1:** The *Incremental* schema and its ground representation are as follows.

Incremental schema.

*Type Schemata*

- \( \text{Type(Incr)}: \alpha_1 \times \alpha_2 \)
- \( \text{Type(Terminating)}: \alpha_1 \)
- \( \text{Type(Initial.result)}: \alpha_1 \times \alpha_2 \)
- \( \text{Type(Deconstruction)}: \alpha_1 \times \alpha_3 \times \alpha_1 \)
- \( \text{Type(Non.initial.result)}: \alpha_1 \times \alpha_3 \times \alpha_2 \times \alpha_3 \)

*Mode Schemata*

- \( \text{Mode(Incr)}: i, d \)
- \( \text{Mode(Terminating)}: i \)
- \( \text{Mode(Initial.result)}: i, d \)
- \( \text{Mode(Deconstruction)}: i, d, d \)
- \( \text{Mode(Non.initial.result)}: i, i, i, d \)

*Schema Clauses*

\[
\begin{align*}
\text{Incr}(u_1, u_2) & \quad \text{Terminating}(u_1) \land \\
& \quad \text{Initial.result}(u_1, u_2) \\
\text{Incr}(u_1, u_2) & \quad \neg\text{Terminating}(u_1) \land \\
& \quad \text{Deconstruction}(u_1, v_1, v_2) \land \\
& \quad \text{Incr}(v_2, v_3) \land \\
& \quad \text{Non.initial.result}(u_1, v_1, v_3, u_2)
\end{align*}
\]

Ground representation of Incremental schema.

\[
\text{schema(incr, [1,2])}.
\]
6.3.2 Representation of Data Types

The knowledge base about the DT operations contains for each DT operation a representation of its declared type and mode. For each DT operation there is a ground positive unit clause of the following form:

\[ p(\text{arg}(x_1, m_1, \tau_1), \ldots, \text{arg}(x_n, m_n, \tau_n)) \]

where \( p \) stands for the predicate symbol of the DT operation. The declared type and mode of the predicate \( p/n \) are \( Type(p) = \tau_1 \times \ldots \times \tau_n \) and \( Mode(p) = m_1, \ldots, m_n \) respectively. The ground representation of each \( x_i \) \( (1 \leq i \leq n) \) is \( \text{var}(N_i) \) where \( N_i \) is a positive integer.

**Examples:** Let us assume the following DT operations.

1. Set union: DT operation \( \text{union}/3 \) with type \( Type(\text{union}) = \text{set}(\tau) \times \text{set}(\tau) \times \text{set}(\tau) \) and mode \( Mode(\text{union}) = (i, i, d). \)

2. Head of a sequence: DT operation \( \text{head}/2 \) with type \( Type(\text{head}) = \text{seq}(\tau) \times \tau \) and mode \( Mode(\text{head}) = (i, d). \)

3. Domain of a relation: DT operation \( \text{dom}/3 \) with type \( Type(\text{dom}) = \text{set}(\text{tuple}(\tau_1, \tau_2)) \times \text{set}(\tau_1) \) and mode \( Mode(\text{dom}) = (i, d). \)
They are represented by the following ground clauses respectively.

1. `union(arg(var(1),i,set(tvar(1))), arg(var(2),i,set(tvar(1))), arg(var(3),d,set(tvar(1))))`
2. `head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1)))`
3. `dom(arg(var(1),i,set(tuple(tvar(1),tvar(2)))), arg(var(2),d,set(tvar(1))))`

### 6.3.3 Representation of Equality Predicate

The knowledge base about equality contains a representation of its declared type and mode. That is, `Type(eq) = α × α` and `Mode(eq) = (d, d)`. This is represented by a ground clause of the following form.

```prolog
eq(arg(var(1),d,tvar(1)), arg(var(2),d,tvar(1)))
```

### 6.4 Representation and Interpretation of Refinement Trees

A refinement tree is represented by a sequence, each element of which is a node and the links to its children. This sequence corresponds to the inorder left-right traversal of the refinement tree. This representation of refinement trees is called *sequential representation of refinement trees*.

A meta-language is a language which describes another language. A refinement tree is expressed as a sentence in the refinement meta-language.

**Definition 6.4.1:** The *refinement meta-language* is a language in which refinement trees are represented.

The syntax of the refinement meta-language in BNF production rules is shown in Appendix D.

An interpreter for refinement trees successively interprets the node representations of a refinement tree. The traversal of the refinement tree corresponds to the order of interpreting its nodes. The interpreter for SIDS performs inorder depth-first left-right traversal of the refinement tree.
The constructed SI-programs are intended to be evaluated by the computation rule of Prolog. We believe that refinement order inorder depth-first left-right traversal of the refinement tree facilitates the refinement activity. That because programmers think procedurally and in terms of the intended computation rule. Of course, other orders for performing refinements are possible. Any valid traversal should be inorder. Variations can be breadth-first and depth-first right-to-left.

6.4.1 Representation of Nodes of Refinement Trees

A node is represented as a unit clause of the form node(A, B) or of the form node(A).

1. A stands for the representation of the refinement that is performed on this node of the refinement tree.

2. B represents a sequence of links to the children of the node in left-right order. Nodes whose A part represents a schema refinement have also B part.

Nodes of the form node(A) represent leaves of the refinement tree. It is worth noting that each node is uniquely identified by the predicate symbol of the predicate in A.

The possible forms of the representation of a refinement, i.e. A part in the representation of the node, are as follows.

\[
\text{refine}(q(\text{Arg}_1, \ldots, \text{Arg}_n), \text{Schema})
\]
\[
\text{refine}(q(\text{Arg}_1, \ldots, \text{Arg}_n), \text{dt_eq}, \text{DT-operation})
\]
\[
\text{refine}(p, \text{Schema})
\]
\[
\text{refine}(p, \text{dt_eq}, \text{DT-operation})
\]

Schema stands for the name of a program schema. \(q(\text{Arg}_1, \ldots, \text{Arg}_n)\) stands for the representation of a typed cga of the top-level undefined predicate including the type of its terms in the atom and the mode of its arguments. This predicate is given from the user. \(p\) stands for the predicate symbol of a new undefined predicate created by the system. \(\text{DT-operation}\) stands for the ground representation of a literal of a DT operation or equality predicate which refines the corresponding undefined predicate. The possible forms of the atom of the literal \(\text{DT-operation}\) are the following.

1. \(p(\text{arg}(x_1, m_1, r_1), \ldots, \text{arg}(x_n, m_n, r_n))\) where \(p(x_1, \ldots, x_n)\) is the typed cga, \(r_1, \ldots, r_n\) are the types in the atom \(p(x_1, \ldots, x_n)\) of the arguments \(x_1, \ldots, x_n\) respectively.
m_1, \ldots, m_n are the modes of the arguments x_1, \ldots, x_n respectively.

2. Typed atoms of equality predicate whose one argument is a constant are represented as follows.

\[
eq(\text{arg(var(1),d,Type)},\text{arg(Const,d,Type)})
\]

where Const is a constant of type Type, var(1) is a variable whose type in atom \(\text{eq(var(1), Const)}\) is Type and \(d, d\) are the modes of var(1) and Const respectively.

It is worth noting that the refinement in the root node of a refinement tree has either the form \(\text{refine(q(Arg_1, \ldots, Arg_n), Schema)}\) or the form \(\text{refine(q(Arg_1, \ldots, Arg_n), dt_eq, DT-operation)}\). That is, this refinement representation requires the typed cga of the undefined predicate, the type of its arguments in the atom and their modes to be specified explicitly.

Nodes of the form node(\(A, B\)) represent schema refinements. \(B\) represents links to the children of the node node(\(A, B\)). The links have the form, \(<\text{Link}_1, \ldots, \text{Link}_n>\). \(\text{Link}_i\) (\(1 \leq i \leq n\)) is either a predicate symbol or a tuple of the form (\(\text{predicate.symbol, unrefined}\)).

6.4.2 Interpretation of Nodes of Refinement Trees

The representations of refinements have the following informal interpretation.

1. \(\text{refine(p(Arg_1, \ldots, Arg_n), Schema)}\): Apply the schema Schema to the undefined predicate \(p/n\). The typed cga \(p(Arg_1, \ldots, Arg_n)\) is provided by the programmer.

2. \(\text{refine(p, Schema)}\): Apply the schema Schema to the undefined predicate with predicate symbol \(p\). Its typed cga including the type of its terms in the atom and the expected mode of its arguments are provided by the system.

3. \(\text{refine(p(Arg_1, \ldots, Arg_n), dt_eq, DT.operation)}\): Refine the undefined predicate \(p/n\) by a literal of a DT operation or equality represented by DT.operation. If the atom of the literal represented by DT.operation is \(q(Arg_1^2, \ldots, Arg_n^2)\) then \(m \leq n\). In addition, the matching arguments of predicates \(p/n\) and \(q/m\) are specified by the programmer. The typed cga \(p(Arg_1^1, \ldots, Arg_n^1)\) is provided by the programmer.

4. \(\text{refine(p, dt_eq, DT.operation)}\): Refine the undefined predicate with predicate symbol \(p\) by a literal of a DT operation or equality represented by DT.operation. If the atom of the literal represented by DT.operation is \(q(Arg_1, \ldots, Arg_m)\) and the arity of the predicate \(p\) is \(n\) then \(m \leq n\). The matching arguments of predicates \(p/n\) and \(q/m\)
are specified by the programmer. The typed cga of the undefined predicate including the type of its terms in the atom and the expected mode of its arguments are provided by the system.

The order for refining the children of a node is same as their order in the sequence of links, i.e. \(< Link_1, ..., Link_n >\).

**Example 1:** The predicate \(fac(n, f)\) is true iff \(f\) is the factorial of \(n\).

A typed cga for \(fac/2\) is \(fac(n, f)\) whose both arguments \(n\) and \(f\) have type \(nat\). The ground representation of this typed cga including the mode of its arguments is \(fac(arg(var(1), i, nat), arg(var(2), d, nat))\). The refinement tree for factorial is as follows.

**Schema refinements**

\[
\text{node(refine(fac(arg(var(11), i, nat), arg(var(12), d, nat)), incr)}, \[p_0, p_1, p_2, p_3]\).
\]

**DT refinements**

\[
\text{node(refine(p_0, dt_eq, eq(arg(var(1), d, nat), arg(0, d, nat)))),}
\]
\[
\text{node(refine(p_1, dt_eq, neutral_mult_div_nat(arg(var(1), d, nat)))),}
\]
\[
\text{node(refine(p_2, dt_eq, pred_nat(arg(var(1), i, nat), arg(var(2), d, nat)))),}
\]
\[
\text{node(refine(p_3, dt_eq, }
\]
\[
\text{mult_nat(arg(var(1), i, nat), arg(var(2), i, nat), arg(var(3), d, nat))}).
\]

The informal interpretation of the first node of this refinement tree is as follows. The predicate \(fac/2\) should be refined by Incremental schema. This schema refinement creates new undefined predicates. The links to the nodes representing these new undefined predicates in left-right order are \([p_0, p_1, p_2, p_3]\). The interpretation of the second node is as follows. The predicate with predicate symbol \(p_0\) should be refined by equality predicate whose typed atom is \(eq(var(1), 0)\). \(0\) is a constant of type \(nat\). \(var(1)\) is a variable with type \(nat\) in atom \(eq(var(1), 0)\). \(d, d\) are the modes of \(0\) and \(var(1)\) respectively. The other nodes have similar informal interpretations.

The SI-program which has been constructed by this refinement tree is following. This SI-program does not include the definitions of DT operations.

**Signatures**

\[
\text{neutral_mult_div_nat: nat}
\]
\[
\text{fac: nat} \times \text{nat}
\]
\[
\text{mult_nat: nat} \times \text{nat} \times \text{nat}
\]
Figure 6.3: Refinement tree for predicate \textit{fac}\textsubscript{2}.

\begin{center}
\begin{tabular}{c|c|c|c|c}
\textbf{pred\textsubscript{nat}: nat $\times$ nat} & \textbf{p0: nat} & \textbf{p1: nat $\times$ nat} & \textbf{p2: nat $\times$ al $\times$ nat} & \textbf{p3: nat $\times$ al $\times$ nat $\times$ nat} \\
\hline
\textbf{Modes} & neutral\textsubscript{mult}\textunderscore div\textunderscore nat: d & fac: i, d & mult\textsubscript{nat}: i, i, d & pred\textsubscript{nat}: i, d \\
\hline
\textbf{Refinements} & \textbf{fac(x1,x2) $\leftarrow$ p0(x1) $\Lambda$ p1(x1,x2)} & \textbf{fac(x1,x2) $\leftarrow$ $\neg$ p0(x1) $\Lambda$ p2(x1,x3,x4) $\Lambda$ fac(x4,x5) $\Lambda$ p3(x1,x3,x5,x2)} & \textbf{p0(x1) $\leftarrow$ eq(x1,0)} & \textbf{p1(x1,x2) $\leftarrow$ neutral\textsubscript{mult}\textunderscore div\textunderscore nat(x2)} \\
& \textbf{p2(x1,x3,x4) $\leftarrow$ pred\textsubscript{nat}(x1,x4)} & \textbf{p3(x1,x3,x5,x2) $\leftarrow$ mult\textsubscript{nat}(x1,x5,x2)} & & \\
\end{tabular}
\end{center}

The schematic representation of the refinement tree for the factorial example is shown in Figure 6.3.
6.4.3 Examples

The first example of this section illustrates a partial refinement tree and the derived SI-program from its interpretation. The second example which is more complex than the example for factorial of the previous section illustrates a complete refinement tree and the derived SI-program from its interpretation. In addition, it shows the clauses which are derived by each node of the refinement tree.

Example 1: The following partial refinement tree is converted into a partial SI-program for insertion sort.

Schema refinements

\[
\text{node(refine(insSort(arg(var(11),i,seq(tvar(1)))), arg(var(12),d,seq(tvar(2))), incr), [p0,p1,p2,p3]).}
\]
\[
\text{node(refine(p2,subgoal_B), [p4,p5]).}
\]
\[
\text{node(refine(p3,incr), [(p6,unrefined), (p7,unrefined), (p8,unrefined)].)
\]

DT refinements

\[
\text{node(refine(p0.dt.eq, empty_seq(arg(var(l),d,seq(tvar(l)))))).}
\]
\[
\text{node(refine(p1,dt.eq, empty_seq(arg(var(l),d,seq(tvar(l)))))).}
\]
\[
\text{node(refine(p4,dt.eq, head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1)))).}
\]
\[
\text{node(refine(p5,dt.eq, tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1)))))),)
\]

The corresponding schematic representation of the refinement tree is shown in Figure 6.4. The corresponding partial SI-program without definitions of DT operations is shown in Appendix A.1.

Example 2: The predicate incrOrd(s) is true iff the sequence of integers s is ordered in increasing order. The refinement tree for incrOrd(s) is as follows.

Schema refinements

\[
\text{node(refine(incrementOrd(arg(var(11),i,seq(tvar(1)))), incr), [p0,p1,p2,p3]).}
\]
\[
\text{node(refine(p0,case), [p4,p5]).}
\]
\[
\text{node(refine(p2,subgoal_3_B), [p10,p11,p12]).}
\]
\[
\text{node(refine(p5,subgoal_A), [p6,p7]).}
\]
\[
\text{node(refine(p6,subgoal_B), [p8,p9]).}
\]

139
The schematic representation of this refinement tree is shown in Figure 6.5. The corresponding SI-program without definitions of DT operations is following. The definition of each predicate and the corresponding refinement which constructed that predicate definition are shown as well.
Signatures

tail: seq(a1) × seq(a1)
empty_seq: seq(a1)
head: seq(a1) × a1
le_int: int × int
incrOrd: seq(int)
P4: seq(int)
P0: seq(int)
P1: seq(int)
P10: seq(int) × int × seq(int)
P11: seq(int) × int × seq(int)
P13: seq(int) × int × int × seq(int)
P12: seq(int) × int × seq(int)
P14: seq(int) × int × int × seq(int)
P2: seq(int) × int × seq(int)
P3: seq(int) × int
P5: seq(int)
P6: seq(int) × seq(int)
P7: seq(int) × seq(int)
P8: seq(int) × seq(int)
P9: seq(int) × seq(int)
Modes

tail: i, d
empty_seq: d
head: i, d
le_int: i, i
incrOrd: i
p4: i
p0: i
p1: i
p10: i, d, d
p11: i, i, d
p13: i, d, i, i
p12: i, i, i
p14: i, i, i, i
p2: i, d, d
p3: i, i
p5: i
p6: i, d
p7: i, i
p8: i, d
p9: i, d

Refinements.

Refinement: Incremental
    incrOrd(x1) ← p0(x1) ∧ p1(x1)
    incrOrd(x1) ← ¬p0(x1) ∧ p2(x1,x2,x3) ∧ incrOrd(x3) ∧ p3(x1,x2)

Refinement: Case
    p0(x1) ← p4(x1)
    p0(x1) ← p5(x1)

Refinement: Subgoal3.B
    p2(x1,x2,x3) ← p10(x1,x2,x3) ∧ p11(x1,x2,x3) ∧ p12(x1,x2,x3)

Refinement: Subgoal.B
    p6(x1,x5) ← p8(x1,x5) ∧ p9(x1,x5)

Refinement: Subgoal.A
    p5(x1) ← p6(x1,x5) ∧ p7(x1,x5)
    p12(x1,x2,x3) ← p13(x1,x4,x2,x3) ∧ p14(x1,x4,x2,x3)

Refinement: DT.eq
    p1(x1) ← true
p3(x1,x2) ← true
p4(x1) ← empty_seq(x1)
p10(x1,x2,x3) ← head(x1,x2)
p11(x1,x2,x3) ← tail(x1,x3)
p7(x1,x5) ← empty_seq(x5)
p13(x1,x4,x2,x3) ← head(x3,x4)
p14(x1,x4,x2,x3) ← le_int(x2,x4)
p8(x1,x5) ← - empty_seq(x1)
p9(x1,x5) ← tail(x1,x5)

The interpretation of the first node in inorder, depth-first, left-right traversal of the refinement tree defines the predicate incrOrd/1. That is, it creates the following program clauses

\[
\text{incrOrd}(x_1) \leftarrow p_0(x_1) \land p_1(x_1)
\]

\[
\text{incrOrd}(x_1) \leftarrow \neg p_0(x_1) \land p_2(x_1,x_2,x_3) \land \text{incrOrd}(x_3) \land p_3(x_1,x_2)
\]

The interpretation of the second node in inorder, depth-first, left-right traversal of the refinement tree defines the predicate p0/1, i.e.

\[
p_0(x_1) \leftarrow p_4(x_1)
\]

\[
p_0(x_1) \leftarrow p_5(x_1)
\]

and so on.

### 6.5 Representation of SI-Programs

An SI-program is represented as a sorted binary tree. Each node of the sorted binary tree has the representation of a predicate of the SI-program. The nodes of the sorted binary tree have the following form.

\[
\text{bt}(\text{Predicate.name}, \text{Instance}, \text{Parent}, \text{Children}, \text{Refinement.name}, \text{Inferred.mode},
\text{Expected.mode}, \text{Type}, \text{Flag}, \text{Left.child}, \text{Right.child})
\]
1. **Predicate name**: The name of the predicate which is represented in this node. This name is used as key for sorting the binary tree.

2. **Instance**: The clauses defining this predicate. The type and the expected mode of each predicate derived by the instantiation operation.

3. **Parent**: This slot has a link, i.e. a predicate name, to the predicate of its parent node in the refinement tree.

4. **Children**: This slot has a list of links, i.e. a list of predicate names, to its children nodes in the refinement tree.

5. **Refinement name**: This slot has the name of the refinement which has been applied for the definition of the predicate represented in this node.

6. **Inferred mode**: This slot has the form \([\text{Call\_pattern}, \text{Success\_pattern}]\). It has the call and success modes of this predicate derived by the mode inference procedure.

7. **Expected mode**: The expected mode of this predicate.

8. **Type**: The type of this predicate.

9. **Flag**: This slot has a flag which is used for predicates which are refined by DT refinements. It has value on if the arguments of the undefined predicate have been matched with the ones of the DT operation or equality, otherwise it has value off. Predicates refined by schema refinements have always this flag on.

10. **Left\_child, Right\_child** are the left and right children of this node.

The representation of the predicates \(\text{fac}/2\), \(\text{p1}/2\) and \(\text{neutral\_mult\_div\_nat}/1\) after the construction of program for the factorial example are as follows. Note that the constructed program and the refinement tree for this example are shown in Section 6.4.

\[
bt(\text{fac},
[[[\text{fac}, \text{arg}(\text{var}(11), i, \text{nat}), \text{arg}(\text{var}(12), d, \text{nat})]],
[\text{p0}, \text{arg}(\text{var}(11), i, \text{nat})]],
[\text{p1}, \text{arg}(\text{var}(11), i, \text{nat}), \text{arg}(\text{var}(12), d, \text{nat})]],
[[\text{fac}, \text{arg}(\text{var}(11), i, \text{nat}), \text{arg}(\text{var}(12), d, \text{nat})]],
[\text{not p0}, \text{arg}(\text{var}(11), i, \text{nat})],
[\text{p2}, \text{arg}(\text{var}(11), i, \text{nat}), \text{arg}(\text{var}(34), d, \text{tvar}(35)), \text{arg}(\text{var}(38), d, \text{nat})]],
[\text{fac}, \text{arg}(\text{var}(38), i, \text{nat}), \text{arg}(\text{var}(39), d, \text{nat})],
[\text{p3}, \text{arg}(\text{var}(11), i, \text{nat}), \text{arg}(\text{var}(34), i, \text{tvar}(35)), \text{arg}(\text{var}(39), i, \text{nat}),
\text{arg}(\text{var}(12), d, \text{nat})]],)
\]
Partial evaluation in logic programming is described in the following way in [48] and in [57]. “Given a program \( \text{Prog} \) and a goal \( G \), partial evaluation produces a new program \( \text{Prog}' \) which is \( \text{Prog} \) specialized to goal \( G \). \( \text{Prog}' \) should have the same semantics as \( \text{Prog} \) with respect to \( G \), that is, the correct and computed answers for \( G \) in \( \text{Prog} \) should be equal to answers for \( G \) in \( \text{Prog} \). It is also expected that \( G \) should be executed more efficiently in \( \text{Prog}' \) than in \( \text{Prog} \).”

Partial evaluation, in general, is applied to a program and some (partial) data. Let \( PE \) be a partial evaluator, let \( \text{Prog} \) be either an object program or meta-program which \( PE \) is applied to and let \( D \) be some (partial) data of \( \text{Prog} \).

1. If \( \text{Prog} \) is a logic program and \( D \) is a goal, then \( PE \) evaluates \( \text{Prog} \) with respect to \( D \) generating \( \text{Prog}' \). \( \text{Prog}' \) is expected to run more efficiently than \( \text{Prog} \) for all instances \( D \theta \) of \( D \).
2. If \( \text{Prog} \) is an interpreter, and \( D \) is an object program, then \( PE \) performs “compiling” of \( D \).
3. If \( \text{Prog} \) is a partial evaluator (self-application) and \( D \) is an interpreter, then \( PE \) performs
compiler-generation. That is, the output is a compiler corresponding to the interpreter \( D \).

Partial evaluators for logic programs is a class of meta-programs which have mainly been used for optimizing logic programs \([27], [92]\). They have been used as well in the construction of logic programs \([48], [49], [52]\). Partial evaluation supports program development methodologies based on data abstraction and on procedural abstraction by removing layers of procedure calls and by propagating data structures \([48]\). Programming techniques like accumulators are incorporated into programs by partial evaluation \([52]\). A technique interpreter schema representing a technique is partially evaluated with respect to a program to incorporate the technique into it. Whenever something is general there is potential for partial evaluation \([88]\). The generality of meta-level description is appropriate for applying partial evaluation. The abstraction-based paradigm is a domain where partial evaluation can be applied due to its generality.

In this thesis, 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.

### 6.6.2 Partial Evaluation of Constructed Programs

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

The partial evaluators that we use for the needs of this thesis are Mixtus \([81]\) and SP \([28]\). Mixtus is a partial evaluator for full Prolog including its non-logical features. SP is designed to specialize declarative normal logic programs.

We partially evaluated using Mixtus and 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. Note that the program names Fibonacci 1 and Fibonacci 2 stand for two different implementations of Fibonacci numbers. Fibonacci 1 has been implemented using the Divide and conquer schema. Fibonacci 2 has been implemented using the Search schema. Four tables of results are following. The first one shows the number of clauses of the SI-programs and the ones generated by partial evaluation. We notice that the residual programs of SP have the least clauses. The improvement in size of the SI-programs by SP is significant. SI-programs have from 3.33 (factorial) to 8.86 (Fibonacci 2) times more clauses than the corresponding programs generated by SP. SI-programs have from 1.57 (spanning tree) to 6.25 (max of a sequence) times more clauses than the corresponding programs generated by Mixtus. The second table illustrates for the same set of programs the time and space requirements of the SI-programs. The third and fourth tables illustrate the time and space requirements of the residual programs of Mixtus and SP respectively. The runtime results have been measured for 100 runs. The measurements of global stack, trail size and number of garbage collections are for one run. We notice that in some cases the residual programs of Mixtus show the best efficiency while in some others the residual programs of SP show the best efficiency. The residual programs generated by SP are from 2.45 (Fibonacci 1) to 29.33 (sum of elements) times more efficient than the corresponding SI-programs. The residual programs generated by Mixtus are from 1.12 (Fibonacci 1) to 8.63 (search sequence) times more efficient than the corresponding SI-programs.

The most likely reasons for these optimisations 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.

**Program size results:**

<table>
<thead>
<tr>
<th>Program name</th>
<th>SI-programs</th>
<th>Mixtus programs</th>
<th>SP programs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no. of clauses</td>
<td>no. of clauses</td>
<td>no. of clauses</td>
</tr>
<tr>
<td>Factorial:</td>
<td>10</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Fibonacci 1:</td>
<td>14</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Fibonacci 2:</td>
<td>62</td>
<td>25</td>
<td>7</td>
</tr>
</tbody>
</table>
Ordered sequence: 24
Search sequence: 13
Max of a sequence: 25
Sum of elements: 13
Insertion sort: 27
Selection sort: 40
Merge sort: 64
Quick sort: 53
Spanning tree: 105

Time and space results for SI-programs:

<table>
<thead>
<tr>
<th>Program name</th>
<th>Runtime (100 runs)</th>
<th>Global stack size (1 run)</th>
<th>Trail size (1 run)</th>
<th>No. of GC (1 run)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial:</td>
<td>0.61</td>
<td>4196</td>
<td>132</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci 1:</td>
<td>163.84</td>
<td>113888</td>
<td>4440</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci 2:</td>
<td>879.30</td>
<td>219592</td>
<td>1276</td>
<td>7</td>
</tr>
<tr>
<td>Ordered sequence:</td>
<td>1.71</td>
<td>6240</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>Search sequence:</td>
<td>0.69</td>
<td>4168</td>
<td>124</td>
<td>0</td>
</tr>
<tr>
<td>Max element:</td>
<td>1.46</td>
<td>7160</td>
<td>164</td>
<td>0</td>
</tr>
<tr>
<td>Sum of elements:</td>
<td>0.88</td>
<td>6012</td>
<td>168</td>
<td>0</td>
</tr>
<tr>
<td>Insertion sort:</td>
<td>5.53</td>
<td>20848</td>
<td>236</td>
<td>0</td>
</tr>
<tr>
<td>Selection sort:</td>
<td>14.17</td>
<td>74732</td>
<td>1076</td>
<td>0</td>
</tr>
<tr>
<td>Merge sort:</td>
<td>13.88</td>
<td>58420</td>
<td>492</td>
<td>36</td>
</tr>
<tr>
<td>Quick sort:</td>
<td>10.62</td>
<td>60448</td>
<td>568</td>
<td>36</td>
</tr>
<tr>
<td>Spanning tree:</td>
<td>27.56</td>
<td>62744</td>
<td>504</td>
<td>0</td>
</tr>
</tbody>
</table>

Time and space results for residual programs by Mixtus:

<table>
<thead>
<tr>
<th>Program name</th>
<th>Runtime (100 runs)</th>
<th>Global stack size (1 run)</th>
<th>Trail size (1 run)</th>
<th>No. of GC (1 run)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial:</td>
<td>0.18</td>
<td>2232</td>
<td>132</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci 1:</td>
<td>146.26</td>
<td>611388</td>
<td>1604</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci 2:</td>
<td>151.64</td>
<td>611388</td>
<td>1604</td>
<td>1</td>
</tr>
<tr>
<td>Ordered sequence:</td>
<td>0.22</td>
<td>2192</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>Search sequence:</td>
<td>0.08</td>
<td>2020</td>
<td>124</td>
<td>0</td>
</tr>
<tr>
<td>Max element:</td>
<td>0.27</td>
<td>3056</td>
<td>164</td>
<td>0</td>
</tr>
</tbody>
</table>
# Time and space results for residual programs by SP:

<table>
<thead>
<tr>
<th>Program name</th>
<th>Runtime (100 runs)</th>
<th>Global stack size (1 run)</th>
<th>Trail size (1 run)</th>
<th>No. of GC (1 run)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial:</td>
<td>0.22</td>
<td>2480</td>
<td>132</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci 1:</td>
<td>66.79</td>
<td>233880</td>
<td>7980</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci 2:</td>
<td>143.84</td>
<td>403748</td>
<td>92</td>
<td>0</td>
</tr>
<tr>
<td>Ordered sequence:</td>
<td>0.19</td>
<td>2312</td>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>Search sequence:</td>
<td>0.12</td>
<td>2200</td>
<td>124</td>
<td>3</td>
</tr>
<tr>
<td>Max element:</td>
<td>0.32</td>
<td>3092</td>
<td>164</td>
<td>3</td>
</tr>
<tr>
<td>Sum of elements:</td>
<td>0.03</td>
<td>2748</td>
<td>112</td>
<td>3</td>
</tr>
<tr>
<td>Insertion sort:</td>
<td>1.07</td>
<td>5564</td>
<td>188</td>
<td>3</td>
</tr>
<tr>
<td>Selection sort:</td>
<td>3.51</td>
<td>18612</td>
<td>696</td>
<td>3</td>
</tr>
<tr>
<td>Merge sort:</td>
<td>5.09</td>
<td>18844</td>
<td>456</td>
<td>3</td>
</tr>
<tr>
<td>Quick sort:</td>
<td>2.00</td>
<td>13572</td>
<td>568</td>
<td>3</td>
</tr>
<tr>
<td>Spanning tree:</td>
<td>13.18</td>
<td>25128</td>
<td>440</td>
<td>3</td>
</tr>
</tbody>
</table>

We show for the insertion sort in Section 3.7 and for goal \( G \leftarrow \text{insSort}(S, Q) \) the programs generated by Mixtus and SP.

The residual program, \( Prog' \), for insertion sort returned by Mixtus is following.

\[
\text{insSort}(A, B) :- \\
\quad \text{insSort1}(A, B).
\]

\[
\text{insSort1}(\text{[]}, \text{[]}). \\
\text{insSort1}(\text{[]}, _):- !, \\
\text{fail}.  \\
\text{insSort1}([A|B], D) :- \\
\quad \text{insSort1}(B, C), \\
\quad 'p3.1' (A, B, C, D).
\]
The residual program, Prog', for insertion sort returned by SP is following.

\[
\text{insSort}([\ ], [\ ]):- true.
\]
\[
\text{insSort}([X_1|X_2], X_3):- \\
\text{\textbackslash + fail,} \\
\text{insSort}(X_2, X_4), \\
p3_1([X_1|X_2], X_1, X_4, X_3).
\]
\[
p3_1(X_1, X_2, X_3, [X_2|X_3]):- \\
p6_1(X_1, X_2, X_3).
\]
\[
p8_1(X_1, X_2, [X_3|X_4], [X_3|X_5]):- \\
\text{\textbackslash + p6_1(X_1, X_2, [X_3|X_4]),} \\
p3_1(X_6, X_2, X_4, X_5).
\]
\[
p6_1(X_1, X_2, [\ ]):- true.
\]
p6_1(X1, X2, [X3|X4]) :-
    X2=<X3.
Chapter 7

Conclusions

7.1 Comparison of this Program Development Method with Related Work

In this section, the method of this thesis is compared with other schema-based program development methods. Initially, the schemata are compared as theses are the most important components of a schema-based method. Next, the methods are compared on the following features.

1. Construction of non-trivial programs.
2. Combination of instances of schemata.
3. Size of constructed programs and refinement trees.
4. Modes (use of modes, mode precision and mode analysis of partial programs).
5. Approaches to proving correctness.

Next, it is shown that program reasoning by this method should be performed at the level of refinement trees. Finally, the conclusions from this comparison are presented.
7.1.1 Comparison of Schemata

A comparison of schema-based methods should start from the basic and most important components, i.e. the program schemata. The sets of schemata that a method proposes and their computational power are important for the judgement of a schema-based method. We distinguish methods which provide large sets of schemata and the ones which provide small sets of schemata. Large sets of schemata are not easily manageable by humans. Their design depends on structural similarities or on the processing of particular data types. Methods which use such schemata do not usually propose fixed sets of schemata. If fixed sets of such schemata are proposed their computational power is restricted to the construction of a limited class of programs. On the other hand, small sets of schemata are more manageable by humans. Such schemata are usually independent of the underlying data types. That is, the arguments of the literals of these schemata are variables. Methods with small sets of schemata usually propose a fixed set of schemata for the construction of programs. Such schemata are computationally powerful in the sense that very large classes of programs can be constructed by using them.

Methods with large sets of schemata: Initially, we discuss the methods with large sets of schemata [5], [30], [40], [52], [53], [80], [86], [90]. The schemata of these methods are specific to particular data types. For a common data type these schemata are instances of the schemata of this thesis.

The definition of schemata in [40] is based on the inductive definition of the underlying data types. Different schemata have to be used for each data type and for each inductive definition. The clauses of these schemata consist of an inductive argument, the arguments which are changed and the ones which are not changed by the recursion and the recursive subgoals. These schemata support the construction of partial instances. That is, partial instances are derived by instantiation in [40]. These partial instances have to be refined by formal transformations to be completed.

Abstraction in [5], [52], [53], [80], [86] is based on control flow similarities between logic programs. These tend to be constructed from commonly occurring program idioms such as processing all elements of a list. These skeletons and clichés tend to be closely related to certain data structures. Experience seems to be the main requirement needed to select such schemata. A library of such schemata may become quite large because schemata should be available for each data structure and method of processing its elements.

Similarly, schemata derived as the most specific generalization of sets of programs result in large hierarchies of schemata [30], [90]. Abstraction based on most specific generalization may result in schemata with very little structure in them to support the design process. For
example, an atom consisting only of a predicate variable and arguments individual variables is a generalization of very large set of programs. These methods depend very much on the set of programs that are used for finding the most specific generalization. The schemata that can be constructed by these approaches are as many as there are programs, that is, infinite. Other criteria are considered in conjunction with the most specific generalization in order to provide manageable sets of schemata [30]. Such a criterion which has been used in [30] is the representation of a general programming technique by each schema.

Methods with small sets of schemata: Next, we will compare the method of this thesis and the ones in [20], [21], [25], [26], [31], [73], [74], [97] which support small sets of schemata.

Four logic description schemata are proposed in [20], [21]. They are based on structural and computational generalization strategies. These schemata represent different problem solving strategies than the ones represented by the schemata of this thesis. The logic description schema which applies when generalization is not followed is analogous to the Incremental schema of this thesis. A detailed comparison of these schemata is presented in Section 3.4. The tupling logic description schema in [20], [21] has fixed components, i.e. it is defined in terms of the predicate append/3. As a consequence, this schema depends upon the list data type.

One schema, namely the Divide-and-Conquer is used in [25], [26] for constructing logic programs. This Divide-and-Conquer schema is analogous to the Incremental and Divide-and-Conquer schemata of this thesis. The representation and use of the Divide-and-Conquer schema in [25], [26] is different from the approach of this thesis. This Divide-and-Conquer schema is hardwired into the synthesis mechanism while in our method the schemata are input into the system. That is, the schema in [25], [26] is a program which performs the divide-and-conquer strategy while in this thesis schemata are represented as schemata and their use involves the instantiation operation. A detailed comparison of the Divide-and-Conquer schema in [25], [26] with the Incremental and Divide-and-Conquer of this thesis is presented in Section 3.4.

The schemata in [31], [73], [74], [97] are higher-order expressions. They accept predicate variables as arguments. The constructed programs have as their clauses the clauses of the schemata which have been used for their construction. A predicate defined by a higher-order schema involves a clause whose head is the predicate itself and its body has a call to the higher-order schema. This is because the same high-order language is used for both programs and schemata. There is no distinction between programs and schemata in [31], [73], [74], [97]. Note that negation is not studied in [73], [74]. On the other hand, schemata and programs in this thesis are expressions of a meta-language and of first-order logic respectively. If we disregard that the Divide-and-conquer schema in [73] has as arguments
predicate variables then this schema is analogous to the Divide-and-conquer schema of this thesis.

It should be mentioned that the schemata of this thesis included a type schema and a mode schema for each predicate variable. In addition, each literal schema is associated with informal guidance which assists programmers to make appropriate design decisions for the refinement of the instances of this literal schema. None of the aforementioned methods supports analogous features.

7.1.2 Comparison of Methods

Construction of non-trivial programs: We argue using examples that the method of this thesis supports the construction of non-trivial programs. Most previous methods have been demonstrated for small examples only. For this reason it is hard to make direct comparisons about the suitability of this method for developing larger pieces of software. Larger examples cannot be shown within the limitations of published papers, and it may be that other methods have been applied to non-trivial examples. However, in the comparisons below, we draw attention to features of other methods which may hinder their application to the development of larger programs. We argue that this method is designed to overcome some of these limitations. Two examples of non-trivial programs that have been constructed by this method are shown in Appendix B. The first example constructs the depth-first search spanning tree of a strongly connected directed graph. The second example is the implementation of a unification algorithm.

Methods which are based on inductive synthesis [25], [26], [86], [90] support mainly the construction of trivial programs. The set of examples which are used for specification of predicates is the main bottleneck of these approaches. Enhancing this approach by properties [25], [26] or skeletons [86] does not improve much the complexity of derived programs.

Methods which perform synthesis based on formal transformations of a specification [40] have not shown success in constructing non-trivial programs. The main bottleneck of these approaches are the small transformation steps which result in long derivation sequences.

Construction of non-trivial programs has not been shown by methods which start program development based on a formal specification. Structured formal specifications is a good approach to meet this goal. Even though, it is not shown in [20], [21] how this method can be scaled up to non-trivial programs.

The methods in [31], [97] do not support negation. The lack of support for negation makes
the task of developing non-trivial programs much more difficult. The methods in [5], [80] need to be extended with larger clichés and skeletons respectively in order to show their capability for the construction of non-trivial programs. The method in [30] supports the construction of small programs. The design of schemata in [30] which is based on syntactic similarities is the main reason for its limitation.

The method in [73], [74] illustrates the construction of complex programs such as parsers, interpreters and meta-interpreters. The method of skeletons and techniques in [52] does support the construction of non-trivial programs. However, the identification of the initial skeleton for large programs and for new application domains does not seem to be less complex than writing the target program. The idea of large skeletons [52] does not seem practical for programmers unless they are experienced in the particular application domain. A programmer should have understood an abstraction of the whole program in order to be able to specify its skeleton. In addition, the concept of skeletons is not "well-defined". That is, two programmers may understand different skeletons for the same problem domain. On the other hand, the construction method of this thesis provides informal guidance to the next refinements. That is, refining an undefined predicate by one of the five schemata puts forward new subtasks that have to be met by subsequent refinements. Each literal schema informally defines a subtask that has to be completed by its instances.

Combination of instances of schemata: It is clear that schema instances which have undefined components can be nested. The method of this thesis provides support, i.e. types and modes, to ensure that pieces of several schema instances fit together correctly. The definition of an undefined predicate by a schema is performed automatically based on the expected modes and types of the arguments of the predicate and the ones of the schema. We also believe that top-down refinement is the typical and natural approach that logic programmers follow. The methods in [5], [20], [21], [97] do not discuss program construction by applying more than one schemata, or by applying a particular schema more than once. The composition of programs which are extensions of the same skeletons is discussed in [52], [86]. The method of this thesis does not have any feature which corresponds to composition of skeletons. It is shown in [26] that during synthesis new sets of examples and properties can be inferred. The synthesis method is applied on these sets of examples and properties for the synthesis of a procedure which is called by the initial one. It is shown in [40] that the synthesis of a new predicate can start during the derivation of a program clause. If the form $U$ of the relation $R$ in a partially instantiated schema clause does not have the desired form, i.e. it is not a conjunction of literals nor a disjunction of conjunction of literals, then a procedure for the relation $R$ has to be derived. A new schema is selected and the formula $\forall(R \rightarrow U)$ is taken as the specification of the relation $R$ for the new synthesis. The combination of different schemata is illustrated in [31], [73], [74]. Procedures for different subgoals of a clause can be constructed by defining the corresponding predicates in terms
of a schema.

Size of constructed programs and refinement trees: The method of this thesis constructs programs which have larger size than the corresponding ones of the other methods. Their large size is due to their structured form. The benefit of the structured form is that construction and maintenance become easier by performing them at the design level. Actually, this method constructs executable program designs. We can get shorter and faster programs by partially evaluating these structured programs. In addition, the predicates of these structured logic programs apart from the top-level one and the DT operations can have redundant arguments. These arguments are carried down during development due to the top-down refinement method. That is, at any point during development it is not known which arguments are needed by the DT refinements of that refinement subtree. These redundant arguments are removed by partial evaluation from the program which will be used for running.

The size of refinement trees for non-trivial programs is large. Consequently, such refinement trees cannot be manipulated easily by programmers. Programmers can reason effectively on such refinement trees if support is provided by the system. A menu-drive interface extended with graphical presentation of refinement trees and automatic guidance for the refinement process will make large refinement trees amenable to human manipulation and reasoning.

Modes: The method of this thesis constructs moded programs. Mode declarations are supplied as meta-knowledge to the system in [86] in order to determine which variables will be unified. One of the uses of modes in this thesis is for the unification of arguments of the DT operations with the ones of the predicate that they refine. That is, each unified argument of the unrefined predicate must have a mode which is subsumed by the mode of the corresponding argument of the DT operation. Modes in and out which correspond to call and success modes of this thesis are used in the construction of logic descriptions in [20], [21]. The following differences are noted from the modes used in this approach. First, valid modes in [20], [21] are {ground, variable, ngv} where ngv stands for neither ground nor variable. While the ones in this method are {ground, don't know}. That is, the method in [20], [21] derives more precise modes. Second, the constructed predicate can satisfy more than one pairs of in and out modes which do not have to be subsumed. The method of this thesis constructs predicates which satisfy one pair of call and success modes. Third, mode analysis of partial programs is performed in this thesis. Mode analysis of partial programs is not discussed in [20], [21]. Note that the other methods do not use modes. Finally, it should be mentioned that modes in this thesis support schema refinements.
7.1.3 Comparison of Correctness Methods

The main features of the correctness method of this thesis are the following.

1. Specifications are transformed into a structured form which facilitates correctness proofs.

2. This correctness method provides guidance to proofs through the proof schemes which correspond directly to design schemata.

3. The structure of an SI-program is reflected in the structure of its correctness proof.

4. The correctness of the DT operations is assumed. This results in correctness proofs shorter than proofs which have to show the correctness of DT operations.

The different approaches to proving correctness in related work are classified as follows. First, specifications in [25], [26], [90] are expressed by examples of behaviour of the constructed procedure. Programs in [90] are specified by sets of positive and negative examples. Verification is discussed with respect to these sets of examples. In a similar direction, programs in [25], [26] are specified by sets of examples and properties. Verification is discussed with respect to these sets of examples and properties. Second, specifications in [20], [40] are expressed as formulas in first-order logic and correctness is ensured by construction. It is shown in [20] that the logic descriptions are correct by construction. Logic descriptions in [20] have a structured form similar to the one of this thesis. The development method in [40] produces programs which are partially correct with respect to a first-order logic specification. It is also shown that there exists a specification called the implemented specification which is stronger that the actual specification. The completed definition of the implemented relation is complete with respect to the implemented specification. Program clauses in [40] are derived from the corresponding partially instantiated schema clauses by performing equivalence rewriting of formulas. Correctness theorems in this thesis are proved by equivalence preserving transformations. Finally, the remaining schema-based methods do not discuss correctness with respect to specifications.

7.1.4 Reasoning about Programs in the Meta-Level

A set of examples shown in the bibliography of the other schema-based methods are discussed in this section. These trivial examples have also been implemented in the method of this thesis. Their refinement trees are shown in Appendix F because the refinement trees are only needed in the next discussion. In addition, this section shows that program reasoning in this method should be performed at the level of refinement trees. 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.

Two refinement trees \( T_1 \) and \( T_2 \) are called similar if their non-leaf nodes contain exactly the same schema refinements. Their leaves may consist of different DT refinements. Note that \( \text{Subgoal}_A \) and \( \text{Subgoal}_B \) are assumed to be the same refinement, i.e. \( \text{Subgoal} \).

The methods in [5], [30], [31], [40], [52], [53], [80], [86], [90] use the same schema for the construction of both predicates \( \text{subset}/2 \) and \( \text{prefix}/2 \). That is, they use a schema which complete traverses the underlying data structure. The refinement trees of the predicates \( \text{subset}/2 \) and \( \text{prefix}/2 \) are similar. That is, they have the same control. The DT refinements of these refinement trees are different. One program can be derived from the other by changing just the DT operations in the refinement tree. This method uses similar refinement trees to construct programs for these predicates. The refinement trees are shown in Examples 1 and 2 of the Appendix F.

The implementation of the predicates \( \text{lengthAcc}/2 \) and \( \text{sumAcc}/2 \) in Examples 3 and 4 of Appendix F involve the use of the accumulator technique [5], [20], [52]. The refinement trees of these predicates are similar. That is, the accumulator technique is encoded by a particular set of schema refinements and by a particular arrangement of these schema refinements.

The predicates \( \text{append}/3 \) and \( \text{efface}/3 \) in the Examples 5 and 6 of Appendix F require complete and partial traversal of the underlying data structure. The methods in [5], [31], [40], [52], [53], [80], [86] use different schemata for deriving programs for each one of these predicates. The refinement tree of these predicates are similar. That is, they have been constructed by the same pattern of schemata. One program can be derived from the other by changing just the DT operations in the refinement tree.

Let \( T_1 \) and \( T_2 \) be two refinement trees. \( T_2 \) is an extension of \( T_1 \) if \( T_2 \) can be derived from \( T_1 \) by replacing one or more leaves by non-leaf refinement trees. Each non-leaf refinement tree is rooted in the corresponding leaf node.

The predicates \( \text{connected}/2 \) and \( \text{path}/3 \) whose refinement trees are shown in Examples 7 and 8 of Appendix F correspond to examples from [52]. The program for \( \text{path}/3 \) in [52] is derived from the program for \( \text{connected}/2 \) by extending it. \( \text{connected}/2 \) involves the implementation of the transitivity relation using graphs. Graphs in this example are represented by the
relation edge/2 as a set of unit clauses. The refinement tree of path/3 is an extension of the refinement tree for connected/2. The leaf node pt in the refinement tree for connected/2 has been replaced by a refinement tree rooted in that node.

7.1.5 Conclusions from the Comparison

As with other schema-based approaches, the method of this thesis is designed to provide a structured, systematic way of developing logic programs. We build on previous work and make some improvements over other methods, emphasising features supporting scalability. On the other hand, our development of schema-based methods has highlighted certain difficulties that apply to any structured, formal approach.

On the positive side, this method provides a small set of schemata, including type and mode information, which are independent of particular data types. This independence provides greater abstraction than other schema-based approaches, and encourages the designer to focus on problem-solving rather than other characteristics of programs. It also facilitates the combination of schemata into larger programs since the type and mode information provides useful constraints. Data independence also encourages a more structured approach to verification, mirroring the schema structure of the program.

On the other hand, it is clear that this approach requires the programmer to write very lengthy refinement trees, in an initially unfamiliar style, which are translated into even lengthier programs. This is a consequence of the increased structure and abstraction of this approach. For a human programmer, this is a definite disadvantage and suggests that this method, like other highly formal and structured methods, will require sophisticated support tools to be used effectively. The top-down tree-structured approach in this method suggests the use of graphical interactive tools providing rich feedback to the designer after each design decision.

In summary, the contribution of this thesis is to extend and improve schema-based development methods, to a degree where their real advantages and disadvantages can be seen more clearly. The fact that a small set of typed, moded schemata and data types provides a rich and highly structured design language is striking, and can surely be exploited further. Strong connections have also been shown between design structure and proof structure. However, it is equally clear that appropriate tools and environments are needed to realise the advantages. The potential payoff of such a structured approach cannot yet be measured, though it has been shown, to a degree not shown for other methods, that this method provides practical features of developing non-trivial programs.
7.2 Limitations of this Method

The limitations of this method are discussed in this section. In addition, features of this method are discussed that may appear as limitations to a reader.

New function symbols on user-defined DTs are not introduced by this method. There is a methodological and a technical reason for this. The methodological reason stems from that aims of this method. That is, this method aims to define new data types by building them on top of pre-defined ones. This method provides a rich set of built-in DTs which we believe is powerful enough to define other DTs. The DT \texttt{f\textunderscore struct} is such an example. The technical reason is the following. In order to allow new function symbols arguments in the head of clauses of SI-programs would have to contain function symbols and equality atoms would have to contain compound terms.

The programmer is encouraged to follow left-right refinement order. Refinement order same as the intended computation rule allows the derivation by the mode analysis procedure more precise modes. Deriving precise modes facilitates the argument matching in DT refinements. Note that this method allows refinement order other than left-right.

SI-programs have large size due to their highly structured form. The size of SI-programs is larger than the corresponding ones of the other schema-based methods. Their size can be reduced by partial evaluation. SI-programs have redundant arguments due to the top-down refinement process. These redundant arguments can be removed by partial evaluation as well.

The size of refinement trees for non-trivial programs become large. Human reasoning on such refinement trees is difficult. A menu-driven interface extended with graphical presentations of refinement trees will be more friendly to programmers than the sequential form of refinement trees. Using a menu-driven interface a programmer can select successively a predicate and either a schema refinement or DT refinement that will refine it. Then both the sequential and graphical forms of the refinement tree can be constructed automatically.

7.3 The Practical Use of the Method

The size of refinement trees is large and so programmers will not like to write refinement trees in the refinement meta-language. SIDS need to be extended with a friendly and
intelligent interface to make the method appealing to programmers.

The interface which a programmer has to have in order to start program development is as follows.

1. A menu with the available set of schemata.

2. A menu with the available data types.

3. A menu with the undefined predicates. Initially, only the top-level predicate will be in the menu. New undefined predicates will be added after each schema refinement.

4. A graphical form of the constructed refinement tree. The programmer should be able to see part of the refinement tree, e.g. the path from the root upto a specific node. Each node of the refinement tree should have annotations with details about the task that the corresponding refinement is intended to solve and possibly other details which will guide the programmer into his/her design decisions.

Initially, the programmer will select an undefined predicate for refinement. Next, he will fire either a schema refinement or a DT refinement based upon the task that he/she wants to be completed by the selected undefined predicate. If a schema refinement is fired then the programmer has to state explicitly his intentions. That is, he/she has to particularize the informal meaning of the corresponding atom schema to his problem. If a DT refinement is fired then the system will perform mode analysis. The programmer can match the corresponding arguments by using types, modes and the informal meaning of the atom schemata from the root of the refinement tree upto the current node as particularized to the specific problem.

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.

The long term goals of this approach are as follows. The programmer will supply as input into the system a specification of his problem, i.e. either in first-order logic or an algorithm in pseudo-code, or in some other form that looks appealing to programmers. The system will be extended with a mapping tool. The aim of the mapping tool is to map the specification of his problem to an expression of the refinement meta-language. A such tool will be interactive. Such mapping tools have been developed in other frameworks in [11], [91] for mapping requirements to designs. The system incrementally will increase its programming knowledge. Design decisions will be shifted from the programmer to the system.
7.4 Contributions

This thesis presents a new schema-based method of programming in logic which advances previous work in this area. The contributions of this thesis are in three areas:

1. A method for systematic development of logic programs.
2. A method which enhances the schema-based paradigm.
3. A systematic approach to verification of programs developed from schemata.

7.4.1 Method of Programming

The basic language constructs of this programming method are the five program schemata and the DT operations. The programming style suggested by this method is top-down. The contributions of this method in the field of systematic development of logic programs are the following:

1. This method shows that the schema-based paradigm can be used for development of non-trivial pieces of software.
2. This method shows that constructing logic programs is an engineering activity. Construction of a logic program involves selection and application of appropriate construction parts, i.e. program schemata and DT operations.
3. The constructed SI-programs are well engineered. Their structure reflects the design decisions which have been applied to undefined predicates during program development. These programs can be maintained easily by changing design decisions.
4. Methods like static analysis which have been developed with other problem domains in mind are used during program development. Static analysis is performed to partial programs as well.
5. The programs are amenable to partial evaluation and other source-to-source optimisations to improve performance and reduce program size.
7.4.2 Schema-based Program Development

This method has enhanced the schema-based paradigm with several new aspects. Its novel features are the following:

1. The domain knowledge, i.e. the data types, are separated from the programming knowledge, i.e. the program schemata. The structural details of the domain knowledge are confined into data type operations.

2. The set of program schemata consists of only five schemata, making them manageable by humans. No analogous schemata have appeared in the literature for three of them. These schemata are flexible and expressive enough in order to be applied to large classes of problems.

3. The schemata are problem independent algorithm design strategies. Their selection is based on program design criteria that programmers follow.

4. The 5 schemata capture the expected mode and type of the predicates. That is, type and mode schemata for predicate variables are introduced by this method.

5. Design decisions are guided. That is, each schema literal is associated with informal guidance which supports the refinement of its instances.

6. The application of refinements is directed by modes and types. That is, the application of a program schema to an undefined predicate is directed by types and expected modes. In addition, types and modes direct the argument matching in DT refinements apart the guidance which is offered by the top-down refinement process.

7. The instantiation operation is simple and the instances of schemata are formulas in the target language.

8. This method provides the necessary support, i.e. types and modes, to ensure that the instances of the 5 schemata are combined correctly.

9. The constructed SI-programs are type correct.

10. The constructed SI-programs are well-moded.

7.4.3 Correctness of Logic Programs

The main contribution of this thesis on correctness is the guidance which is provided to the correctness proofs. Correctness proofs are guided by the constructed SI-programs. That is,
1. Each schema is associated with a correctness proof scheme.

2. The structure of an SI-program is reflected in the structure of its correctness proof.

The correctness proof scheme that corresponds to the schema that has been applied for the construction of the top-level predicate is initially followed. Next, correctness proof schemes for schemata that have been applied for the construction of predicates in lower levels are followed. Correctness proofs for predicates in lower levels of an SI-program are used as lemmas for correctness proofs of predicates in higher levels of the program. Finally, correctness proofs are performed in a top-down manner, as the refinements in the program development process.

7.5 Further Research

It is not known at each refinement step which arguments will be used in the next refinements. During program development some arguments of schema predicates are never used. We call them redundant arguments. A next research step is to apply transformations such as fold, unfold and definition in order to remove redundant arguments [76].

The development of a semi-automatic verifier for programs that are constructed by this method is another direction for research. The input to a such verifier would be the following:

1. A specification S expressed into structured form.

2. The completion comp(P) of a logic program P.

3. A correctness theorem.

4. A knowledge base of axioms and lemmata for the DTs. Specifications of DT operations.

The verifier would be directed by the programmer. That is, the programmer would select first-order logic inferences and the verifier would carry them out. We see a such verifier to become more automatic in a stepwise manner by shifting proof decisions from the programmer to the system. A such verifier directed by the user should be able to prove lemmata which would be used in subsequent proofs. The prover apart from first-order inferences should be able to perform inductive proofs. That is, either structural induction or computational induction.
We expect this method to be extended by modules. Modules will extend the power of this logic program construction framework. They will facilitate also the organization of large pieces of software. Such modules may import system-defined DTs, user-defined DTs and other modules. A programmer may define a module which will consist of the definition of a DT and its operations. A module may be exported to other modules.

The study of more powerful computation rules is another research direction. Programs will be constructed with such computation rules in mind. That is,

1. Dynamic selection rules defined by means of delay declarations [36]. Calls can be delayed until their arguments satisfy a delay condition specified by the programmer. In this case, this method will incorporate into the constructed programs control declarations.

2. Coroutining: Dynamic selection of calls based on the data flow through their variables [70]. Call selection is assumed to be performed automatically by the system. Control annotations could be included in the programs.

Another direction for further research is static analysis of the constructed programs in other more complex abstract domains like \( \{ \text{ground, free, partially ground} \} \). Aliasing should be studied in such domains as well.

An interesting topic for research is to offer guidance to the programmer during the refinement process. Guidance can be given with respect to a logic specification expressed in structured form. Suggestions can be given to the programmer as to what schemata and/or DT operations he/she may use during development. The suggestions can be extracted from the form of the logic specification and the data types involved.
Bibliography


Appendix A

Partial SI-Programs and Implementations of Data Type Operations

This appendix illustrates the construction of a partial SI-program and the implementation by this method of a complex DT operation for user-defined DTs.

A.1 Partial SI-Program for Insertion Sort

A partial SI-program for insertion sort without the definitions of DT operations is following. It has been constructed by the conversion of the refinement tree in Section 6.4.

Signatures
- empty: seq(a2)
- head: seq(a2) × a2
- insSort: seq(a1) × seq(a3)
- tail: seq(a2) × seq(a2)
- p0: seq(a1)
- p1: seq(a1) × seq(a3)
- p2: seq(a1) × a1 × seq(a1)
p3: seq(a1) \times a1 \times seq(a3) \times seq(a3)
p4: seq(a1) \times a1 \times seq(a1)
p5: seq(a1) \times a1 \times seq(a1)
p6: seq(a1) \times a1 \times seq(a3)
p7: seq(a1) \times a1 \times seq(a3) \times seq(a3)
p8: seq(a1) \times a1 \times seq(a3) \times a4 \times seq(a1) \times a1 \times seq(a3)
p9: seq(a1) \times a1 \times seq(a3) \times a4 \times seq(a3) \times seq(a3)

Modes
empty_seq: d
head: i, d
insSort: i, d
tail: i, d
p0: i
p1: i, d
p2: i, d, d
p3: i, i, i, d
p4: i, d, d
p5: i, i, d
p6: i, i, i
p7: undefined, undefined, undefined, undefined, undefined
p8: i, i, i, d, d, d, d
p9: undefined, undefined, undefined, undefined, undefined, undefined, undefined

Refinements
insSort(x1,x2) \leftarrow p0(x1) \land p1(x1,x2)
insSort(x1,x2) \leftarrow \neg p0(x1) \land p2(x1,x3,x4) \land insSort(x4,x5) \land p3(x1,x3,x5,x2)
p0(x1) \leftarrow empty_seq(x1)
p1(x1,x2) \leftarrow empty_seq(x2)
p2(x1,x3,x4) \leftarrow p4(x1,x3,x4) \land p5(x1,x3,x4)
p3(x1,x3,x5,x2) \leftarrow p6(x1,x3,x5) \land p7(x1,x3,x5,x2)
p3(x1,x3,x5,x2) \leftarrow \neg p6(x1,x3,x5) \land p8(x1,x3,x5,x6,x7,x8,x9) \land
   p3(x7,x8,x9,x10) \land p9(x1,x3,x5,x6,x10,x2)
p4(x1,x3,x4) \leftarrow head(x1,x3)
p5(x1,x3,x4) \leftarrow tail(x1,x4)
A.2 Implementation of a Data Type Operation for a User-defined Data Type

A refinement tree and the constructed SI-program for the DT operation string_to_f_struct/2 are shown in this appendix. The predicate string_to_f_struct(s,f) is true iff f is the representation of a functional structure whose string representation is s.

Refinement tree.

Schema refinements

- node(refine(string_to_f_struct(arg(var(11),i,str),
arg(var(12),d,seq(str))), subgoal_B), [p0,p1]).
- node(refine(p0, subgoal_A), [p2,p3]).
- node(refine(p3, incr), [p4,p5,p6,p7]).
- node(refine(p6, case4), [p8,p9,p10,p11]).
- node(refine(p8, subgoal3_A), [p12,p13,p14]).
- node(refine(p9, subgoal3_A), [p15,p16,p17]).
- node(refine(p10, subgoal3_A), [p18,p19,p20]).
- node(refine(p11, subgoal_A), [p21,p22]).
- node(refine(p21, incr), [p23,p24,p25,p26]).
- node(refine(p23, subgoal_A), [p27,p28]).
- node(refine(p24, subgoal_B), [p32,p33]).
- node(refine(p26, subgoal_B), [p34,p35]).
- node(refine(p28, case3), [p29,p30,p31]).

DT refinements

- node(refine(p1, dt_eq, parse_f_struct(arg(var(1),i,seq(str))))).
- node(refine(p2, dt_eq,
  string_to_ascii(arg(var(1),i,str),arg(var(2),d,seq(nat)))).
- node(refine(p4, dt_eq, empty_seq(arg(var(1),d,seq(tvar(l))))).
- node(refine(p5, dt_eq, empty_seq(arg(var(1),d,seq(tvar(l))))).
- node(refine(p7, dt_eq, seq_cons(arg(var(1),i,seq(tvar(l))),
  arg(var(2),i,tvar(1)),arg(var(3),d,seq(tvar(1))))).
- node(refine(p12, dt_eq, head_tail(arg(var(1),i,seq(tvar(1))),
  arg(var(2),d,tvar(1)),arg(var(3),d,seq(tvar(1))))).
- node(refine(p13, dt_eq, eq(arg(var(1),d,nat),arg(40,d,nat))))).
- node(refine(p14, dt_eq, eq(arg(var(1),d,str),arg('d',d,str))))).
- node(refine(p15, dt_eq, head_tail(arg(var(1),i,seq(tvar(1)))),
arg(var(2),d,tvar(1)),arg(var(3),d,seq(tvar(1))))).
node(refine(p16, dt_eq, eq(arg(var(1),d,nat),arg(41,d,nat)))).
node(refine(p17, dt_eq, eq(arg(var(1),d,str),arg(')',d,str)))).
node(refine(p18, dt_eq, head_tail(arg(var(1),i,seq(tvar(1))),
    arg(var(2),d,tvar(1)),arg(var(3),d,seq(tvar(1)))))).
node(refine(p19, dt_eq, eq(arg(var(1),d,nat),arg(44,d,nat)))).
node(refine(p20, dt_eq, eq(arg(var(1),d,str),arg(')',d,str)))).
node(refine(p22, dt_eq,
    ascii_to_string(arg(var(1),i,seq(nat)),arg(var(2),d,str)))).
node(refine(p25, dt_eq, head_tail(arg(var(1),i,seq(tvar(1))),
    arg(var(2),d,tvar(1)),arg(var(3),d,seq(tvar(1)))))).
node(refine(p27, dt_eq,
    head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p29, dt_eq, eq(arg(var(1),d,nat),arg(40,d,nat)))).
node(refine(p30, dt_eq, eq(arg(var(1),d,nat),arg(41,d,nat)))).
node(refine(p32, dt_eq, empty_seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p33, dt_eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p34, dt_eq, seq_cons(arg(var(1),i,seq(tvar(1))),
    arg(var(2),i,tvar(1)),arg(var(3),d,seq(tvar(1)))))).
node(refine(p35, dt_eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).

SI-Program Without Definitions of DT Operations

Signatures

string_to_ascii: str × seq(nat)
seq_cons: seq(al) × al × seq(al)
head_tail: seq(al) × al × seq(al)
empty_seq: seq(al)
ascii_to_string: seq(nat) × str
head: seq(al) × al
parse_f_struct: seq(str)
string_to_f_struct: str × seq(str)
p4: str × seq(nat)
p32: str × seq(nat) × seq(nat) × str × str × seq(nat)
p3: str × seq(nat) × seq(str)
p2: str × seq(nat) × seq(str)
p0: str × seq(str)
p10: str × seq(nat) × str × str × seq(nat)
p1: str × seq(str)
p12: str × seq(nat) × nat × str × str × seq(nat)
p11: str × seq(nat) × str × str × seq(nat)
p13: str × seq(nat) × nat × str × str × seq(nat)
p14: str × seq(nat) × nat × str × str × seq(nat)
p15: str × seq(nat) × nat × str × str × seq(nat)
p16: str × seq(nat) × nat × str × str × seq(nat)
p17: str × seq(nat) × nat × str × str × seq(nat)
p18: str × seq(nat) × nat × str × str × seq(nat)
p19: str × seq(nat) × nat × str × str × seq(nat)
p20: str × seq(nat) × nat × str × str × seq(nat)
p21: str × seq(nat) × seq(nat) × str × str × seq(nat)
p22: str × seq(nat) × seq(nat) × str × str × seq(nat)
p23: str × seq(nat)
p24: str × seq(nat) × seq(nat) × str × str × seq(nat)
p25: str × seq(nat) × nat × str × seq(nat)
p26: str × seq(nat) × nat × seq(nat) × str × str × seq(nat) ×
  seq(nat) × str × str × seq(nat)
p27: str × seq(nat) × nat
p28: str × seq(nat) × nat
p29: str × seq(nat) × nat
p30: str × seq(nat) × nat
p31: str × seq(nat) × nat
p32: str × seq(nat) × seq(nat) × str × str × seq(nat)
p33: str × seq(nat) × seq(nat) × str × str × seq(nat)
p34: str × seq(nat) × nat × seq(nat) × str × str × seq(nat) ×
  seq(nat) × str × str × seq(nat)
p35: str × seq(nat) × nat × seq(nat) × str × str × seq(nat) ×
  seq(nat) × str × str × seq(nat)
p36: str × seq(nat) × str × str × seq(nat)
p37: str × seq(nat) × str × str × seq(nat)
p38: str × seq(nat) × str × str × seq(nat)
p39: str × seq(nat) × str × str × seq(nat)

Modes
  string_to_ascii: i, d
  seq_cons: i, i, d
  head_tail: i, d, d
  empty_seq: d
ascii.to.string: i, d
head: i, d
parse.f.struct: i
string.to.f.struct: i, d
p4: d, i
p32: d, i, d, d, d, d
p3: d, i, d
p2: i, d, d
p0: i, d
p10: d, i, d, d, d
p1: i, i
p12: d, i, d, d, d, d
p11: d, i, d, d
p13: d, i, i, d, d, i
p14: d, i, i, d, d, i
p15: d, i, d, d, d, d
p16: d, i, i, d, d, i
p17: d, i, i, d, d, i
p18: d, i, d, d, d, d
p19: d, i, i, d, d, i
p20: d, i, i, d, d, i
p21: d, i, d, d, d, d
p23: d, i
p22: d, i, i, d, d, i
p24: d, i, d, d, d, d
p25: d, i, d, d, d
p27: d, i, d
p26: d, i, i, i, d, d, d, d, d, d
p28: d, i, i
p29: d, i, i
p30: d, i, i
p31: d, i, i
p33: d, i, i, d, d, d
p34: d, i, i, i, d, d, d, d, d, d
p35: d, i, i, i, d, d, i, i, d, d, d, d
p5: d, i, d
p6: d, i, d, d, d
p8: d, i, d, d, d
p7: d, i, i, i, d
p9: d, i, d, d, d
Program clauses

string_to_f_struct(x1,x7) ← p0(x1,x7) ∧ p1(x1,x7)

p0(x1,x7) ← p2(x1,x2,x7) ∧ p3(x1,x2,x7)
p1(x1,x7) ← parse_f_struct(x7)
p2(x1,x2,x7) ← string_to_ascii(x1,x2)
p3(x1,x2,x7) ← p4(x1,x2) ∧ p5(x1,x2,x7)
p3(x1,x2,x7) ← p4(x1,x2) ∧ p6(x1,x2,x4,x5,x6) ∧ p7(x1,x2,x4,x8,x7)
p4(x1,x2) ← empty_seq(x2)
p5(x1,x2,x7) ← empty_seq(x7)
p6(x1,x2,x4,x5,x6) ← p8(x1,x2,x4,x5,x6)
p6(x1,x2,x4,x5,x6) ← p9(x1,x2,x4,x5,x6)
p6(x1,x2,x4,x5,x6) ← p10(x1,x2,x4,x5,x6)
p6(x1,x2,x4,x5,x6) ← p11(x1,x2,x4,x5,x6)
p7(x1,x2,x4,x8,x7) ← seq_cons(x8,x4,x7)
p8(x1,x2,x4,x5,x6) ← p12(x1,x2,x10,x4,x5,x6) ∧ p13(x1,x2,x10,x4,x5,x6) ∧ p14(x1,x2,x10,x4,x5,x6)
p9(x1,x2,x4,x5,x6) ← p15(x1,x2,x11,x4,x5,x6) ∧ p16(x1,x2,x11,x4,x5,x6) ∧ p17(x1,x2,x11,x4,x5,x6)
p10(x1,x2,x4,x5,x6) ← p18(x1,x2,x9,x4,x5,x6) ∧ p19(x1,x2,x9,x4,x5,x6) ∧ p20(x1,x2,x9,x4,x5,x6)
p11(x1,x2,x4,x5,x6) ← p21(x1,x2,x3,x4,x5,x6) ∧ p22(x1,x2,x3,x4,x5,x6)
p12(x1,x2,x10,x4,x5,x6) ← head_tail(x2,x10,x6)
p13(x1,x2,x10,x4,x5,x6) ← eq(x10,40)
p14(x1,x2,x10,x4,x5,x6) ← eq(x4,')')
p15(x1,x2,x11,x4,x5,x6) ← head_tail(x2,x11,x6)
p16(x1,x2,x11,x4,x5,x6) ← eq(x11,41)
p17(x1,x2,x11,x4,x5,x6) ← eq(x4,')')
p18(x1,x2,x9,x4,x5,x6) ← head_tail(x2,x9,x6)
p19(x1,x2,x9,x4,x5,x6) ← eq(x9,44)
p20(x1,x2,x9,x4,x5,x6) ← eq(x4,')')
p21(x1,x2,x3,x4,x5,x6) ← p23(x1,x2) ∧ p24(x1,x2,x3,x4,x5,x6)
p21(x1,x2,x3,x4,x5,x6) ← p23(x1,x2) ∧ p25(x1,x2,x12,x13,x14) ∧ p26(x1,x2,x12,x15,x16,x17,x18,x3,x4,x5,x6)
p22(x1,x2,x3,x4,x5,x6) ← ascii_to_string(x3,x4)
p23(x1,x2) ← p27(x1,x2,x19) ∧ p28(x1,x2,x19)
p24(x1,x2,x3,x4,x5,x6) ← p23(x1,x2,x3,x4,x5,x6) ∧ p33(x1,x2,x3,x4,x5,x6)
p25(x1,x2,x12,x13,x14) ← head_tail(x2,x12,x14)
p26(x1,x2,x12,x15,x16,x17,x18,x3,x4,x5,x6) ← p34(x1,x2,x12,x15,x16,x17,x18,x3,x4,x5,x6) ∧ p35(x1,x2,x12,x15,x16,x17,x18,x3,x4,x5,x6)
The DT operations for sequences are empty_seq/2, head_tail/2, seq_cons/3, tail/2 and head/2. The DT operations for strings are ascii_to_string/2 and string_to_ascii/2. parse_f_struct/1 is a user-defined DT operation implemented by this method. Any predicate that is implemented by this method can be used in subsequent programs as user-defined DT operation.
Appendix B

Non-trivial Programs

This appendix illustrates the construction by this method of two non-trivial logic programs. The first example finds the depth-first spanning tree of a digraph and the second one is an implementation of a unification algorithm.

B.1 Depth-First Search Spanning Tree

Let $G = (V, A)$ be a digraph and let $u \in V$. Then there is a subdigraph $T_u = (V_u, A_u)$ of $G$ that is a rooted tree with root $u$ such that the vertex set of $T_u$ is the set of vertices of $G$ that are reachable from $u$. One method of finding such a rooted tree is a depth-first search, and a rooted tree so found is called a depth-first search tree of $G$. If every vertex of $G$ is reachable from $u$, in particular, if $G$ is strongly connected, then $T_u$ is a spanning subdigraph of $G$, called a depth-first search spanning tree of $G$.

Depth-first search spanning tree. search_tree($rt, gr, dfTr$) is true iff $dfTr$ is a depth-first search spanning tree of $gr$ rooted at $rt$.

The type of predicate search_tree/3 is Type(search_tree) = $\alpha \times$ tuple(set($\alpha$), set(tuple($\alpha$, $\alpha$))) $\times$ tuple(set($\alpha$), set(tuple($\alpha$, $\alpha$))). Its expected mode is Mode(search_tree) = (i, i, d).

A refinement tree for depth-first search spanning tree and the constructed SI-program are as follows.
B.1.1 Refinement tree

Schema refinements

node(refine(search_tree(arg(var(11),i,tvar(1))),
    arg(var(12),i,tuple(set(tvar(1)),set(tuple(tvar(1),tvar(1))))),
    arg(var(13),d,tuple(set(tvar(1)),set(tuple(tvar(1),tvar(1)))) ),
    search), [p0,p2,p3,p4,p5]).

node(refine(p0, subgoal_B), [p6,p7]).
node(refine(p4, subgoal_A), [p18,p19]).
node(refine(p5, subgoal3_A), [p48,p49,p50]).
node(refine(p6, subgoal_A), [p8,p9]).
node(refine(p9, subgoal_A), [p10,p11]).
node(refine(p10, subgoal_A), [p12,p13]).
node(refine(p11, subgoal_A), [p14,p15]).
node(refine(p14, subgoal_A), [p16,p17]).
node(refine(p18, subgoal3_A), [p20,p21,p22]).
node(refine(p19, subgoal_B), [p23,p24]).
node(refine(p23, subgoal3_A), [p25,p26,p27]).
node(refine(p24, subgoal_B), [p32,p33]).
node(refine(p27, subgoal_A), [p28,p29]).
node(refine(p28, subgoal_A), [p30,p31]).
node(refine(p32, subgoal_A), [p34,p35]).
node(refine(p33, subgoal_A), [p38,p39]).
node(refine(p35, subgoal_A), [p36,p37]).
node(refine(p38, subgoal_A), [p40,p41]).
node(refine(p40, subgoal_A), [p42,p43]).
node(refine(p41, subgoal_A), [p46,p47]).
node(refine(p43, subgoal_A), [p44,p45]).
node(refine(p48, subgoal_A), [p51,p52]).
node(refine(p52, case), [p53,p54]).
node(refine(p53, subgoal_B), [p55,p56]).
node(refine(p49, case), [p57,p58]).
node(refine(p57, subgoal_B), [p59,p60]).
node(refine(p58, subgoal_A), [p61,p62]).
node(refine(p61, subgoal_A), [p63,p64]).
node(refine(p62, subgoal_A), [p67,p68]).
node(refine(p64, subgoal_A), [p65,p66]).
DT refinements

node(refine(p2, dt_eq, empty_stack(arg(var(1),d,seq(tvar(1)))))).
node(refine(p3, dt_eq, eq(arg(var(1),d,tvar(1)), arg(var(2),d,tvar(1))))).
node(refine(p7, dt_eq, empty_graph(arg(var(1),d, tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))))).
node(refine(p8, dt_eq, empty_stack(arg(var(1),d,seq(tvar(1)))))).
node(refine(p12, dt_eq, empty_set(arg(var(1),d,set(tvar(1)))))).
node(refine(p13, dt_eq, set_elem(arg(var(1),i,tvar(D), arg(var(2),i,set(tvar(1))), arg(var(3),d,set(tvar(1))))))).
node(refine(p15, dt_eq, push(arg(var(1),i,seq(tvar(1))),arg(var(2),i,tvar(1)),
arg(var(3),d,seq(tvar(1)))))).
node(refine(p16, dt_eq, edges(arg(var(1),i,tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))),
arg(var(2),d,set(tuple(tvar(1),tvar(1))))))).
node(refine(p17, dt_eq, dom_restr(arg(var(1),i, set(tuple(tvar(1),tvar(2)))), arg(var(2),i,set(tvar(1))),
arg(var(3),d,set(tuple(tvar(1),tvar(2))))))).
node(refine(p20, dt_eq, top(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p21, dt_eq, not empty_set(arg(var(1),d,set(tvar(1)))))).
node(refine(p22, dt_eq, member_set(arg(var(1),d,tvar(1)),arg(var(2),i,set(tvar(1)))))).
node(refine(p25, dt_eq, second(arg(var(1),i,tuple(tvar(1),tvar(2))), arg(var(2),d,tvar(2))))).
node(refine(p26, dt_eq, not eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p29, dt_eq, not member_set(arg(var(1),d,tvar(1)), arg(var(2),i,set(tvar(1)))))).
node(refine(p30, dt_eq, edges(arg(var(1),i,tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))),
arg(var(2),d,set(tuple(tvar(1),tvar(1))))))).
node(refine(p31, dt_eq, ran(arg(var(1),i,set(tuple(tvar(1),tvar(2)))), arg(var(2),d,set(tvar(2)))))).
node(refine(p34, dt_eq, edges(arg(var(1),i,tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))),
arg(var(2),d,set(tuple(tvar(1),tvar(1))))))).
node(refine(p36, dt_eq, set_elem(arg(var(1),i,tvar(1)),arg(var(2),i,set(tvar(1))),
arg(var(3),d,set(tvar(1))))).
node(refine(p37, dt.eq,
    make_graph(arg(var(1),i,set(tvar(1),tvar(1)))),
    arg(var(2),d,tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))))).
node(refine(p39, dt.eq,
    push(arg(var(1),i,seq(tvar(1))),
    arg(var(2),i,tvar(1)),arg(var(3),d,seq(tvar(1))))).
node(refine(p42, dt.eq,
    second(arg(var(1),i,tuple(tvar(1),tvar(2))), arg(var(2),d,tvar(2))))).
node(refine(p44, dt.eq, empty_set(arg(var(1),d,set(tvar(1)))))).
node(refine(p45, dt.eq,
    set_elem(arg(var(1),i,tvar(1)),arg(var(2),i,set(tvar(1))),
    arg(var(3),d,set(tvar(1))))).
node(refine(p46, dt.eq,
    edges(arg(var(1),i,tuple(set(tvar(1)), set(tuple(tvar(1),tvar(1))))),
    arg(var(2),d,set(tuple(tvar(1),tvar(1)))))).
node(refine(p47, dt.eq, dom_restr(arg(var(1),i,
    set(tuple(tvar(1),tvar(2)))),arg(var(2),i,set(tvar(1))),
    arg(var(3),d,set(tuple(tvar(1),tvar(2)))))).
node(refine(p50, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p51, dt.eq,
    top(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p54, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p55, dt.eq, empty_set(arg(var(1),d,set(tvar(1)))))).
node(refine(p56, dt.eq,
    pop(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1))))).
node(refine(p59, dt.eq, empty_stack(arg(var(1),d,seq(tvar(1)))))).
node(refine(p60, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p63, dt.eq,
    top(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p65, dt.eq,
    member_set(arg(var(1),d,tvar(1)),arg(var(2),i,set(tvar(1))))).
node(refine(p66, dt.eq,
    delete(arg(var(1),i,tvar(1)),arg(var(2),i,set(tvar(1))),
    arg(var(3),d,set(tvar(1))))).
node(refine(p67, dt.eq,
    pop(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1))))).
node(refine(p68, dt.eq,
    push(arg(var(1),i,seq(tvar(1))),arg(var(2),i,tvar(1)),
    arg(var(3),d,seq(tvar(1))))).
B.1.2 SI-Program Without Definitions of DT Operations

Signatures

set_elem: al × set(al) × set(al)
second: tuple(al,a2) × a2
push: seq(al) × al × seq(al)
pop: seq(al) × seq(al)
member_set: al × set(al)
empty_stack: seq(al)
empty_set: set(al)
edges: tuple(set(al),set(tuple(al,a1))) × set(tuple(al,a1))
dom_restr: set(tuple(al,a2)) × set(al) × set(tuple(al,a2))
delete: al × set(al) × set(al)
empty_graph: tuple(set(al),set(tuple(al,a1)))
make_graph: set(tuple(al,a1)) × tuple(set(al),set(tuple(al,a1)))
ran: set(tuple(al,a2)) × set(al)
search_tree: a3 × tuple(set(a3),set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
top: seq(al) × al
p7: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p67: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) × set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p6: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p57: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p5: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p48: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p46: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) × set(a3) ×
set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p40: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) × set(a3) ×
set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p32: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p30: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p20: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) ×
set(a3) × seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p18: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(a3,a3) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p0: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p10: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p1: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p11: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
p12: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(a3) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p13: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(a3) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p14: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p15: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p16: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))
p17: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
set(a3) × set(tuple(a3,a3)) × set(tuple(a3,a3)) ×
seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3)))
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × seq(set(set(a3),set(tupl(a3,a3))))

p35: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(tupl(a3,a3)) × seq(set(set(a3,a3))) × tuple(set(a3), set(tupl(a3,a3)))

p36: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(tupl(a3,a3)) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p37: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(tupl(a3,a3)) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p38: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(tupl(a3,a3)) ×
seq(set(set(a3,a3))) × tuple(set(a3), set(tupl(a3,a3)))

p39: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(tupl(a3,a3)) ×
seq(set(set(a3,a3))) × tuple(set(a3), set(tupl(a3,a3)))

p4: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × a3 × set(a3) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p42: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × a3 × set(a3) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p41: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × set(a3) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p43: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × a3 × set(a3) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p44: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × a3 × set(a3) ×
set(tupl(a3,a3)) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3)))

p45: a3 × tuple(set(a3), set(tupl(a3,a3))) × seq(set(set(a3,a3))) ×
tuple(set(a3), set(tupl(a3,a3))) × tuple(a3,a3) × a3 × set(a3) ×
set(a3) × set(tuple(a3,a3)) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p47: a3 × tuple(set(a3),set(tuple(a3,a3))) × seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p49: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p50: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p51: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p52: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p53: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p54: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p55: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p56: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p58: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p59: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p60: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
seq(set(tuple(a3,a3))) ×
tuple(set(a3),set(tuple(a3,a3)))

p61: seq(set(tuple(a3,a3))) × tuple(set(a3),set(tuple(a3,a3))) ×
$$seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p62: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p63: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p64: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p65: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p66: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p68: $$seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times set(tuple(a3,a3)) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p8: $$a3 \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

p9: $$a3 \times tuple(set(a3),set(tuple(a3,a3))) \times seq(set(tuple(a3,a3))) \times tuple(set(a3),set(tuple(a3,a3)))$$

Modes
set.elem: i, i, d
second: i, d
push: i, i, d
pop: i, d
member_set: d, i
empty_stack: d
empty_set: d
edges: i, d
dom.restr: i, i, d
delete: i, i, d
empty_graph: d
make_graph: i, d
ran: i, d
search_tree: i, i, d
top: i, d
p7: i, i, i, d
p67: i, i, i, d, d, d
p6: i, i, d, d
p57: i, i, i, d, d
p5: i, i, d, d
p48: i, i, d, d, d
p46: i, i, i, i, i, d, d, d, i
p40: i, i, i, i, i, d, d, d, i
p32: i, i, i, i, i, d, d
p30: i, i, i, i, i, d, d, d, d, d
p20: i, i, i, i, d, d, d
p18: i, i, i, i, d, d, d
p0: i, i, d, d
p10: i, i, i, d, d, d
p1: i, i, i, i, d
p11: i, i, i, i, d, d
p12: i, i, i, i, d, d, d
p13: i, i, i, i, d, d, d
p14: i, i, i, i, d, d
p15: i, i, i, i, i, d, d
p16: i, i, i, i, d, d, d
p17: i, i, i, i, i, d, d, d
p19: i, i, i, i, i, d, d
p2: i, i
p21: i, i, i, i, i, d, d, d
p22: i, i, i, i, i, d, d, d
p23: i, i, i, i, i, d, d
p25: i, i, i, i, i, d, d, d
p24: i, i, i, i, i, d, d
p26: i, i, i, i, i, i, d, d
p27: i, i, i, i, i, i, d, d
p28: i, i, i, i, i, i, d, d, d
p29: i, i, i, i, i, i, i, d, d
p3: i, d
p31: i, i, i, i, i, i, i, d, d, d
p34: i, i, i, i, i, d, d, d
Program clauses
search_tree(x1,x2,x26) ← p0(x1,x2,x3,x4) ∧ p1(x1,x2,x3,x4,x26)
p0(x1,x2,x3,x4) ← p6(x1,x2,x3,x4) ∧ p7(x1,x2,x3,x4)
p1(x1,x2,x3,x4,x26) ← p2(x3,x4) ∧ p3(x4,x26)
p1(x1,x2,x3,x4,x26) ← ¬ p2(x3,x4) ∧ p4(x1,x2,x3,x4,x16,x17) ∧ p1(x1,x2,x16,x17,x26)
\[
p1(x_1, x_2, x_3, x_4, x_{26}) \rightarrow p2(x_3, x_4) \land p4(x_1, x_2, x_3, x_4, x_{16}, x_{17}) \land p5(x_3, x_4, x_8, x_9) \land p1(x_1, x_2, x_8, x_9, x_{26})
\]
\[
p6(x_1, x_2, x_3, x_4) \rightarrow p8(x_1, x_2, x_{10}, x_3, x_4) \land p9(x_1, x_2, x_{10}, x_3, x_4)
\]
\[
p7(x_1, x_2, x_3, x_4) \rightarrow \text{empty graph}(x_4)
\]
\[
p5(x_3, x_4, x_8, x_9) \rightarrow p48(x_3, x_4, x_5, x_8, x_9) \land p49(x_3, x_4, x_5, x_8, x_9) \land p50(x_3, x_4, x_5, x_8, x_9)
\]
\[
p4(x_1, x_2, x_3, x_4, x_{16}, x_{17}) \rightarrow p18(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \land p19(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17})
\]
\[
p2(x_3, x_4) \rightarrow \text{empty stack}(x_3)
\]
\[
p3(x_4, x_{26}) \rightarrow \text{eq}(x_4, x_{26})
\]
\[
p8(x_1, x_2, x_{10}, x_3, x_4) \rightarrow \text{empty stack}(x_{10})
\]
\[
p9(x_1, x_2, x_{10}, x_3, x_4) \rightarrow p10(x_1, x_2, x_{10}, x_{24}, x_3, x_4) \land p11(x_1, x_2, x_{10}, x_{24}, x_3, x_4)
\]
\[
p48(x_3, x_4, x_5, x_8, x_9) \rightarrow p51(x_3, x_4, x_{11}, x_5, x_8, x_9) \land p52(x_3, x_4, x_{11}, x_5, x_8, x_9)
\]
\[
p49(x_3, x_4, x_5, x_8, x_9) \rightarrow p57(x_3, x_4, x_5, x_8, x_9)
\]
\[
p49(x_3, x_4, x_5, x_8, x_9) \rightarrow p58(x_3, x_4, x_5, x_8, x_9)
\]
\[
p50(x_3, x_4, x_5, x_8, x_9) \rightarrow \text{eq}(x_4, x_9)
\]
\[
p18(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \rightarrow p20(x_1, x_2, x_3, x_4, x_{23}, x_{12}, x_{16}, x_{17}) \land p21(x_1, x_2, x_3, x_4, x_{23}, x_{12}, x_{16}, x_{17}) \land p22(x_1, x_2, x_3, x_4, x_{23}, x_{12}, x_{16}, x_{17})
\]
\[
p19(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \rightarrow p23(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \land p24(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17})
\]
\[
p10(x_1, x_2, x_{10}, x_{24}, x_3, x_4) \rightarrow p12(x_1, x_2, x_{10}, x_{25}, x_3, x_4) \land p13(x_1, x_2, x_{10}, x_{25}, x_3, x_4)
\]
\[
p11(x_1, x_2, x_{10}, x_{24}, x_3, x_4) \rightarrow p14(x_1, x_2, x_{10}, x_{24}, x_{27}, x_3, x_4) \land p15(x_1, x_2, x_{10}, x_{24}, x_{27}, x_3, x_4)
\]
\[
p51(x_3, x_4, x_{11}, x_5, x_8, x_9) \rightarrow \text{top}(x_3, x_{11})
\]
\[
p52(x_3, x_4, x_{11}, x_5, x_8, x_9) \rightarrow p53(x_3, x_4, x_{11}, x_5, x_8, x_9)
\]
\[
p52(x_3, x_4, x_{11}, x_5, x_8, x_9) \rightarrow p54(x_3, x_4, x_{11}, x_5, x_8, x_9)
\]
\[
p57(x_3, x_4, x_5, x_8, x_9) \rightarrow p59(x_3, x_4, x_5, x_8, x_9) \land p60(x_3, x_4, x_5, x_8, x_9)
\]
\[
p58(x_3, x_4, x_5, x_8, x_9) \rightarrow p61(x_3, x_4, x_5, x_6, x_8, x_9) \land p62(x_3, x_4, x_5, x_6, x_8, x_9)
\]
\[
p20(x_1, x_2, x_3, x_4, x_{23}, x_{12}, x_{16}, x_{17}) \rightarrow \text{top}(x_3, x_{23})
\]
\[
p21(x_1, x_2, x_3, x_4, x_{23}, x_{12}, x_{16}, x_{17}) \rightarrow \text{empty set}(x_{23})
\]
\[
p16(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \rightarrow p25(x_1, x_2, x_3, x_4, x_{12}, x_{20}, x_{16}, x_{17}) \land p26(x_1, x_2, x_3, x_4, x_{12}, x_{20}, x_{16}, x_{17}) \land p27(x_1, x_2, x_3, x_4, x_{12}, x_{20}, x_{16}, x_{17})
\]
\[
p28(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \rightarrow p32(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17}) \land p33(x_1, x_2, x_3, x_4, x_{12}, x_{16}, x_{17})
\]
\[
p12(x_1, x_2, x_{10}, x_{25}, x_3, x_4) \rightarrow \text{empty set}(x_{25})
\]
\[
p13(x_1, x_2, x_{10}, x_{25}, x_3, x_4) \rightarrow \text{set elem}(x_1, x_{25}, x_24)
\]
\[
p14(x_1, x_2, x_{10}, x_{24}, x_{27}, x_3, x_4) \rightarrow p16(x_1, x_2, x_{10}, x_{24}, x_{28}, x_{27}, x_3, x_4) \land p17(x_1, x_2, x_{10}, x_{24}, x_{28}, x_{27}, x_3, x_4)
\]
\[
p15(x_1, x_2, x_{10}, x_{24}, x_{27}, x_3, x_4) \rightarrow \text{push}(x_{10}, x_{27}, x_3)
\]
\[
p53(x_3, x_4, x_{11}, x_5, x_8, x_9) \rightarrow p55(x_3, x_4, x_{11}, x_5, x_8, x_9) \land p56(x_3, x_4, x_{11}, x_5, x_8, x_9)
\]
\[
p54(x_3, x_4, x_{11}, x_5, x_8, x_9) \rightarrow \text{eq}(x_3, x_5)
\]
\[
p59(x_3, x_4, x_5, x_8, x_9) \rightarrow \text{empty stack}(x_5)
\]
p60(x3,x4,x5,x8,x9) ← eq(x5,x8)

p61(x3,x4,x5,x6,x8,x9) ← p63(x3,x4,x5,x31,x6,x8,x9) ∧ p64(x3,x4,x5,x31,x6,x8,x9)

p62(x3,x4,x5,x6,x8,x9) ← p67(x3,x4,x5,x6,x7,x8,x9) ∧ p68(x3,x4,x5,x6,x7,x8,x9)

p25(x1,x2,x3,x4,x12,x20,x16,x17) ← second(x12,x20)

p26(x1,x2,x3,x4,x12,x20,x16,x17) ← ¬ eq(x20,x1)

p27(x1,x2,x3,x4,x12,x20,x16,x17) ← p28(x1,x2,x3,x4,x12,x20,x22,x16,x17) ∧

p29(x1,x2,x3,x4,x12,x20,x22,x16,x17)

p32(x1,x2,x3,x4,x12,x16,x17) ← p34(x1,x2,x3,x4,x12,x19,x16,x17) ∧

p35(x1,x2,x3,x4,x12,x19,x16,x17)

p33(x1,x2,x3,x4,x12,x16,x17) ← p38(x1,x2,x3,x4,x12,x15,x16,x17) ∧

p39(x1,x2,x3,x4,x12,x15,x16,x17)

p16(x1,x2,x10,x24,x28,x27,x3,x4) ← edges(x2,x28)

p17(x1,x2,x10,x24,x28,x27,x3,x4) ← dom_restr(x28,x24,x27)

p55(x3,x4,x11,x5,x8,x9) ← empty_set(x11)

p56(x3,x4,x11,x5,x8,x9) ← pop(x3,x5)

p63(x3,x4,x5,x31,x6,x8,x9) ← top(x5,x31)

p64(x3,x4,x5,x31,x6,x8,x9) ← p65(x3,x4,x5,x31,x32,x6,x8,x9) ∧

p66(x3,x4,x5,x31,x32,x6,x8,x9)

p67(x3,x4,x5,x6,x7,x8,x9) ← pop(x5,x7)

p68(x3,x4,x5,x6,x7,x8,x9) ← push(x7,x6,x8)

p28(x1,x2,x3,x4,x12,x20,x22,x16,x17) ← p30(x1,x2,x3,x4,x12,x20,x21,x22,x16,x17) ∧

p31(x1,x2,x3,x4,x12,x20,x21,x22,x16,x17)

p29(x1,x2,x3,x4,x12,x20,x22,x16,x17) ← ¬ member_set(x20,x22)

p34(x1,x2,x3,x4,x12,x19,x16,x17) ← edges(x4,x19)

p35(x1,x2,x3,x4,x12,x19,x16,x17) ← p36(x1,x2,x3,x4,x12,x19,x29,x16,x17) ∧

p37(x1,x2,x3,x4,x12,x19,x29,x16,x17)

p38(x1,x2,x3,x4,x12,x15,x16,x17) ← p40(x1,x2,x3,x4,x12,x13,x15,x16,x17) ∧

p41(x1,x2,x3,x4,x12,x13,x15,x16,x17)

p39(x1,x2,x3,x4,x12,x15,x16,x17) ← push(x3,x15,x16)

p65(x3,x4,x5,x31,x32,x6,x8,x9) ← member_set(x32,x31)

p66(x3,x4,x5,x31,x32,x6,x8,x9) ← delete(x32,x31)

p30(x1,x2,x3,x4,x12,x20,x21,x22,x16,x17) ← edges(x4,x21)

p31(x1,x2,x3,x4,x12,x20,x21,x22,x16,x17) ← ran(x21,x22)

p36(x1,x2,x3,x4,x12,x19,x29,x16,x17) ← set_elem(x12,x19,x29)

p37(x1,x2,x3,x4,x12,x19,x29,x16,x17) ← make_graph(x29,x17)

p40(x1,x2,x3,x4,x12,x13,x15,x16,x17) ← p42(x1,x2,x3,x4,x12,x18,x13,x15,x16,x17) ∧

p43(x1,x2,x3,x4,x12,x18,x13,x15,x16,x17)

p41(x1,x2,x3,x4,x12,x13,x15,x16,x17) ← p46(x1,x2,x3,x4,x12,x13,x14,x15,x16,x17) ∧

p47(x1,x2,x3,x4,x12,x13,x14,x15,x16,x17)

p42(x1,x2,x3,x4,x12,x18,x13,x15,x16,x17) ← second(x12,x18)
B.2 Unification

The SI-program for the predicate unify/3 is presented in this section. This predicate is an implementation of the unification algorithm in [87] with a small change to handle variant terms. Note that the algorithm in [87] returns the empty substitution for variants terms.

Initially, the unification algorithm is presented. Next, the refinement tree and the constructed SI-program are presented.

B.2.1 Unification Algorithm

The aim is to present the unification algorithm in a form which helps the reader to understand the mapping of the algorithm into the refinement tree and consequently into the structure of the SI-program.

Algorithm:

Input: Two terms \( T_1 \) and \( T_2 \) to be unified

Output: \( \theta \), the mgu of \( T_1 \) and \( T_2 \), or failure

\[
S := \{T_1 = T_2\};
\]
\[
\theta := \emptyset;
\]
while \( S \neq \emptyset \) and no failure do

new set of terms is \( S \setminus \{T'_1 = T'_2\} \);

case

\( T'_1 \) and \( T'_2 \) are different variables:

new substitution is \( \theta \circ \{T'_1/T'_2\} \);

replace each occurrence of \( T'_1 \) in \( S \) by \( T'_2 \);

\( T'_1 \) is a variable and \( T'_2 \) is a non-variable term and \( T'_1 \) does not occur in \( T'_2 \):

new substitution is \( \theta \circ \{T'_1/T'_2\} \);
replace each occurrence of $T'_1$ in $S$ by $T'_2$;

*T* is a non-variable term and $T'_2$ is a variable and $T'_2$ does not occur in $T'_1$:
new substitution is $\theta \circ \{ T'_2 / T'_1 \}$;

*replace* each occurrence of $T'_2$ in $S$ by $T'_1$;

$T'_1$ and $T'_2$ are identical terms:
continue;

$T'_1$ and $T'_2$ are compound terms with same functors and arities:
new set of terms is $S \cup \{ t_1 = s_1, \ldots, t_n = s_n \}$
where $T'_1 = f(t_1, \ldots, t_n)$ and $T'_2 = f(s_1, \ldots, s_n)$;

otherwise failure

end_case

end_while

It is worth noting the following. The set of terms $S$ is represented as a stack whose elements are pairs. Each pair is a sequence of strings. That is, the type of $S$ is $\text{seq}(\text{tuple}(\text{seq}(\text{str}), \text{seq}(\text{str})))$. The substitution $\theta$ is represented as a sequence whose elements are pairs. The first element of each pair is a string and the second one is a sequence of strings. That is, the type of $\theta$ is $\text{seq}(\text{tuple}(\text{str}, \text{seq}(\text{str})))$. An equation $T_1 = T_2$ is a pair of the form $(T_1, T_2)$.

**Unification Algorithm and Refinement Tree**

The following part of the unification algorithm is mapped into the top part of the refinement tree shown in Figure B.1.

\[
S := \{ T_1 = T_2 \};
\]

$\theta := \emptyset$;

while $S \neq \emptyset$ and no failure do

new set of terms is $S \setminus \{ T'_1 = T'_2 \}$;

end_while

The first case of the algorithm corresponds to the part of the refinement tree with link number 1 which is shown in Figure B.2. The second case of the algorithm corresponds to the part of the refinement tree with link number 2 which is shown in Figure B.2 and so on.
B.2.2 Refinement tree

The schematic form of the refinement tree for unify/3 is presented in a slightly different form than the one of the refinement trees which has been presented in this thesis. This form facilitates the readability of large refinement trees. The changes in the schematic form of large refinement trees are as follows.

1. Arguments are omitted from all predicates except the one in the root node of the refinement tree and the ones of the DT operations.

2. Link nodes are introduced which point to parts of the refinement tree. Each such link node is denoted by a number in circle. This circled number also appears as the root of the corresponding part of the refinement tree.

The predicate unify(T1, T2, Subst) is true iff Subst is the substitution from the unification of terms T1 and T2. 

Type(unify) = seq(str) × seq(str) × seq(tuple(str, seq(str))).

Mode(unify) = (i, i, d).

The definition of the following user-defined DT operations are needed in order to make readable the refinement tree and the constructed SI-program.

1. The predicate replace_occurrences_in_stack(Subst, St, NewSt) is true iff NewSt is St after replacing each occurrence of V in St by T where Subst = (V, T).

Type(replace_occurrences_in_stack) = tuple(str, seq(str)) × seq(tuple(seq(str), seq(str))) × seq(tuple(seq(str), seq(str))).

Mode(replace_occurrences_in_stack) = (i, i, d).

2. The predicate make.idempotent_subst1(Subst, S1, S2) is true iff the substitution S2 is the substitution S1 after replacing each occurrence of V in the range of the elements of S1 by T where Subst = (V, T).

Type(make.idempotent_subst1) = tuple(str, seq(str)) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))).

Mode(make.idempotent_subst1) = (i, i, d).

3. The predicate occurs1(V, T) is true iff the variable V occurs in the term T.

Type(occurs1) = str × seq(str).

Mode(occurs1) = (i, i).

4. The predicate push_subterms(T1, T2, St, NewSt) is true iff NewSt is the stack St after pushing the corresponding terms of the sequences T1 and T2 in reverse order.

Type(push_subterms) = seq(seq(str)) × seq(seq(str)) × seq(tuple(seq(str), seq(str))) × seq(tuple(seq(str), seq(str))).

Mode(push_subterms) = (i, i, i, d).

Schema refinements
node(refine(unify(arg(var(11),i,seq(str)), arg(var(12),i,seq(str)),
    arg(var(13),d,seq(tuple(str,seq(str))))), subgoal.A), [p0,p1]).
node(refine(p0, subgoal.A), [p2,p3]).
node(refine(p1, subgoal.A), [p6,p7]).
node(refine(p3, subgoal.A), [p4,p5]).
node(refine(p7, incr), [p8,p9,p10,p11]).
node(refine(p10, subgoal.A), [p12,p13]).
node(refine(p12, subgoal.B), [p14,p15]).
node(refine(p13, subgoal.A), [p16,p17]).
node(refine(p17, subgoal.A), [p18,p19]).
node(refine(p19, case5), [p20,p21,p22,p23,p24]).
node(refine(p20, subgoal.A), [p25,p26,p27]).
node(refine(p21, subgoal.A), [p37,p38,p39]).
node(refine(p22, subgoal.A), [p49,p50,p51]).
node(refine(p23, subgoal.B), [p61,p62,p63]).
node(refine(p24, subgoal.A), [p64,p65,p66,p67]).
node(refine(p25, subgoal.B), [p28,p29]).
node(refine(p27, subgoal.A), [p35,p36]).
node(refine(p28, subgoal.B), [p30,p31,p32]).
node(refine(p29, subgoal.A), [p33,p34]).
node(refine(p37, subgoal.B), [p40,p41]).
node(refine(p39, subgoal.A), [p47,p48]).
node(refine(p40, subgoal.B), [p42,p43]).
node(refine(p41, subgoal.A), [p44,p45,p46]).
node(refine(p49, subgoal.B), [p52,p53]).
node(refine(p51, subgoal.A), [p59,p60]).
node(refine(p52, subgoal.B), [p54,p55]).
node(refine(p53, subgoal.A), [p56,p57,p58]).
node(refine(p54, subgoal.B), [p68,p69]).
node(refine(p55, subgoal.A), [p76,p77]).
node(refine(p66, subgoal.A), [p80,p81]).
node(refine(p68, subgoal.B), [p70,p71]).
node(refine(p69, subgoal.A), [p72,p73]).
node(refine(p73, subgoal.A), [p74,p75]).
node(refine(p77, subgoal.A), [p78,p79]).
node(refine(p81, subgoal.A), [p82,p83]).

DT refinements
node(refine(p2, dt_eq,
    make_pair(arg(var(1),i,tvar(1)), arg(var(2),i,tvar(2)),
arg(var(3), d, tuple(tvar(1), tvar(2))))).
node(refine(p4, dt.eq, empty_stack(arg(var(1), d, seq(tvar(1)))))**).
node(refine(p5, dt.eq, push(arg(var(1), i, seq(tvar(1))),
    arg(var(2), i, tvar(1)), arg(var(3), d, seq(tvar(1))))))**).
node(refine(p6, dt.eq, empty_seq(arg(var(1), d, seq(tvar(1)))))**).
node(refine(p8, dt.eq, empty_stack(arg(var(1), d, seq(tvar(1)))))**).
node(refine(p9, dt.eq, eq(arg(var(1), d, tvar(1)), arg(var(2), d, tvar(1))))**).
node(refine(p11, dt.eq, eq(arg(var(1), d, tvar(1)), arg(var(2), d, tvar(1))))**).
node(refine(p12, dt.eq,
    top(arg(var(1), i, seq(tvar(1))), arg(var(2), d, tvar(1))))).
node(refine(p15, dt.eq,
    pop(arg(var(1), i, seq(tvar(1))), arg(var(2), d, seq(tvar(1))))).
node(refine(p16, dt.eq,
    first(arg(var(1), i, tuple(tvar(1), tvar(2))), arg(var(2), d, tvar(1))))**).
node(refine(p18, dt.eq,
    second(arg(var(1), i, tuple(tvar(1), tvar(2))), arg(var(2), d, tvar(2))))).
node(refine(p26, dt.eq,
    replace_occurrences_in_stack(arg(var(1), i, tuple(seq(str), seq(str))),
    arg(var(2), i, seq(tuple(seq(str), seq(str)))),
    arg(var(3), d, seq(tuple(seq(str), seq(str))))))**).
node(refine(p30, dt.eq, var.f_struct(arg(var(1), i, seq(str))))**).
node(refine(p31, dt.eq, var.f_struct(arg(var(1), i, seq(str))))**).
node(refine(p32, dt.eq,
    not eq(arg(var(1), d, tvar(1)), arg(var(2), d, tvar(1))))).
node(refine(p33, dt.eq,
    head(arg(var(1), i, seq(tvar(1))), arg(var(2), d, tvar(1))))).
node(refine(p34, dt.eq,
    make_pair(arg(var(1), i, tvar(1)), arg(var(2), i, tvar(2)),
    arg(var(3), d, tuple(tvar(1), tvar(2))));)
node(refine(p35, dt.eq,
    make_idempotent_subst1(arg(var(1), i, tuple(str, seq(str))),
    arg(var(2), i, seq(tuple(str, seq(str)))),
    arg(var(3), d, seq(tuple(str, seq(str))))))**).
node(refine(p36, dt.eq,
    add_last_elem(arg(var(1), i, seq(tvar(1))), arg(var(2), i, tvar(1)),
    arg(var(3), d, seq(tvar(1))))).
node(refine(p38, dt.eq,
    replace_occurrences_in_stack(arg(var(1), i, tuple(seq(str), seq(str)));
    arg(var(2), i, seq(tuple(seq(str), seq(str)))),
    arg(var(3), d, seq(tuple(seq(str), seq(str))))))**).
B.2.3 SI-Program Without Definitions of DT Operations

Signatures

\[\begin{align*}
\text{top: } & \text{seq(a1) } \times \text{a1} \\
\text{second: } & \text{tuple(a1,a2) } \times \text{a2} \\
\text{replace_occurrences_in_stack: } & \text{tuple(str,seq(str)) } \times \text{seq(tuple(seq(str),seq(str))) } \times \\
& \text{seq(tuple(seq(str),seq(str)))} \\
\text{push: } & \text{seq(a1) } \times \text{a1 } \times \text{seq(a1)} \\
\text{pop: } & \text{seq(a1) } \times \text{seq(a1)} \\
\text{occursl: } & \text{str } \times \text{seq(str)} \\
\text{make.pair: } & \text{a1 } \times \text{a2 } \times \text{tuple(a1,a2)} \\
\text{first: } & \text{tuple(a1,a2) } \times \text{a1} \\
\text{f_struct.f_name: } & \text{seq(str) } \times \text{str} \\
\text{empty_stack: } & \text{seq(a1)} \\
\text{empty_seq: } & \text{seq(a1)} \\
\text{comp.f_struct: } & \text{seq(str)} \\
\text{add_last_elem: } & \text{seq(a1) } \times \text{a1 } \times \text{seq(a1)} \\
\text{arity: } & \text{seq(str) } \times \text{nat} \\
\text{f_struct.args: } & \text{seq(str) } \times \text{seq(seq(str))} \\
\end{align*}\]
Figure B.1: The top part of the refinement tree for unify/3.
Figure B.2: Parts 1 and 2 of the refinement tree for unify/3.
Figure B.3: Parts 3 and 4 of the refinement tree for unify/3.
Figure B.4: Part 5 of the refinement tree for unify/3.
head: seq(a1) × a1
make_idempotent_subst1: tuple(str, seq(str)) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str)))
push_subterms: seq(seq(str)) × seq(seq(str)) × seq(tuple(seq(str), seq(str))) × seq(tuple(seq(str), seq(str)))
var_f_strct: seq(str)
unify: seq(str) × seq(str) × seq(tuple(seq(str), seq(str)))

p9: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str)))
p8: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p76: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p7: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str)))
p61: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p6: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str)))
p59: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × tuple(seq(str), seq(str)) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p5: seq(str) × seq(str) × tuple(seq(str), seq(str)) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p49: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × tuple(seq(str), seq(str)) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p47: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str))) × seq(str) × seq(str) × tuple(seq(str), seq(str)) × seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
p4: seq(str) × seq(str) × tuple(seq(str), seq(str)) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str)))

p37: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(seq(str), seq(str))) × seq(str) ×
    seq(str) × tuple(str, seq(str)) × tuple(seq(str), seq(str)) ×
    seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p35: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(seq(str), seq(str))) × seq(str) ×
    seq(str) × tuple(str, seq(str)) × tuple(seq(str), seq(str)) ×
    seq(str) × seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p3: seq(str) × seq(str) × tuple(seq(str), seq(str)) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p2: seq(str) × seq(str) × tuple(seq(str), seq(str)) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p0: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str)))

p1: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str)))

p10: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) × seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p12: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) ×
    tuple(seq(str), seq(str)) × seq(str) ×
    seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p11: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) ×
    seq(tuple(str, seq(str))) ×
    seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p13: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) ×
    seq(tuple(seq(str), seq(str))) ×
    tuple(seq(str), seq(str)) × seq(str) ×
    seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p14: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) ×
    seq(tuple(seq(str), seq(str))) ×
    tuple(seq(str), seq(str)) × seq(str) ×
    seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))

p15: seq(str) × seq(str) × seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) ×
    seq(tuple(seq(str), seq(str))) ×
    seq(tuple(str, seq(str))) × seq(tuple(str, seq(str))) ×
    seq(str) × seq(str) × seq(str) ×
    seq(tuple(seq(str), seq(str))) × seq(tuple(str, seq(str)))
tuple(seq(str),seq(str)) × seq(str) × seq(str) × seq(tupl(seq(str),seq(str))) × seq(tuple(seq(str),seq(str)))

p16: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) ×

p17: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) ×

p18: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p19: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p20: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p21: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p22: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p23: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p24: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

p25: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) ×

seq(tupl(seq(str),seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) × seq(str) × seq(str) × seq(str) × seq(tupl(seq(str),seq(str))) × seq(tuple(str,seq(str))) × seq(str) × seq(str) × seq(str) × seq(str) × seq(tupl(seq(str),seq(str))) × seq(tuple(str,seq(str)))
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tup(te(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p26: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p27: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p28: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p29: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p30: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p31: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p32: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p33: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p34: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tup(le(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(str,seq(str)) × tuple(seq(str),seq(str)) × seq(str) ×
seq(str) × seq(tup(le(seq(str),seq(str))) × seq(tuple(str,seq(str))))
p36: \( \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \\
p38: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \\
p39: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p40: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p41: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p42: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p43: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p44: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p45: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
p46: \text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \\
\text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str}))) \times \text{seq}(\text{tuple}(\text{str},\text{seq}(\text{str})))) \times \text{seq}(\text{str}) \times \\
\text{seq}(\text{str}) \times \text{str} \times \text{tuple}(\text{str},\text{seq}(\text{str})) \times \text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})) \times \\
\text{seq}(\text{str}) \times \text{seq}(\text{str}) \times \text{seq}(\text{tuple}(\text{seq}(\text{str}),\text{seq}(\text{str})))) \times \

212
seq(str) × str × tuple(str,seq(str)) × tuple(seq(str),seq(str)) ×
seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str)))
p58: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × str × tuple(str,seq(str)) × tuple(seq(str),seq(str)) ×
seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str)))
p60: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuplc(str,seq(str))) × seq(tuple(str,seq(str)))
p62: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p63: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p64: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p65: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p66: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p67: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
p68: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuplc(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
tuple(seq(str),seq(str)) × seq(tuplc(str,seq(str)))
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p69: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p70: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p71: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p72: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p73: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p74: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p75: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p76: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p77: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p78: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))

p79: seq(str) × seq(str) × seq(tuple(seq(str),seq(str))) ×
seq(tuple(str,seq(str))) × seq(tuple(seq(str),seq(str))) × seq(str) ×
seq(str) × tuple(seq(str),seq(str)) × seq(str) × seq(str) ×
seq(tuple(seq(str),seq(str))) × seq(tuple(str,seq(str)))
seq(tuple(seq(str),seq(str))) \times seq(tuple(seq(str),seq(str)))

p80: seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(tuple(str,seq(str))) \times seq(tuple(seq(str),seq(str))) \times seq(str) \times

seq(str) \times seq(seq(str)) \times tuple(seq(str),seq(str)) \times

seq(str) \times seq(tuple(seq(str),seq(str))) \times seq(str) \times

seq(str)

p82: seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(tuple(str,seq(str))) \times seq(seq(str)) \times tuple(seq(str),seq(str)) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(tuple(str,seq(str)))

p81: seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(seq(str)) \times tuple(seq(str),seq(str)) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(tuple(str,seq(str)))

p83: seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(seq(str)) \times tuple(seq(str),seq(str)) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

seq(str) \times seq(str) \times seq(tuple(seq(str),seq(str))) \times

Modes

top: i, d

second: i, d

replace_occurrences_in_stack: i, i, d

push: i, i, d

pop: i, d

occurs1: i, i

make_pair: i, i, d

first: i, d

f_struct.f_name: i, d

empty_stack: d

empty_seq: d

comp_f_struct: i

add_last_elem: i, i, d

arity: i, d

f_struct.args: i, d

head: i, d

make_idempotent_subst1: i, i, d

push_subterms: i, i, i, d
var f_struct: i
unify: i, i, d
p9: d, d, i, i, d
p8: d, d, i, i
p76: d, d, i, i, i, i, d, i, d, d, d
p7: d, d, i, i, d
p61: d, d, i, i, i, i, i, d, d, d
p6: i, i, i, d
p59: d, d, i, i, i, i, d, i, d, i, d
p5: i, i, i, d, d
p48: i, i, i, i, d
p47: d, d, i, i, i, i, i, d, d, d
p46: d, d, i, i, i, i, i, d, d, d
p4: i, i, i, d, d
p37: d, d, i, i, i, i, i, d, d, d
p35: d, d, i, i, i, i, i, i, d, d, d
p3: i, i, i, d, d
p2: i, i, d, d
p0: i, i, d, d
p1: i, i, i, d
p10: d, d, i, i, d, d, d, d, d, d
p12: d, d, i, i, d, d, d, d, d, d
p11: d, d, i, i, i, i, d
p13: d, d, i, i, i, i, d, d, d, d
p14: d, d, i, i, d, d, d, d, d, d
p15: d, d, i, i, d, i, d, d, d, d
p16: d, d, i, i, i, i, d, i, d, d, d, d
p17: d, d, i, i, i, i, d, i, d, d, d, d
p18: d, d, i, i, i, i, d, i, d, d, d, d
p19: d, d, i, i, i, i, i, d, d, d, d, d
p20: d, d, i, i, i, i, i, d, d, d, d, d
p21: d, d, i, i, i, i, i, d, d, d, d, d
p22: d, d, i, i, i, i, i, d, d, d, d, d
p23: d, d, i, i, i, i, i, d, d, d, d, d
p24: d, d, i, i, i, i, i, d, d, d, d, d
p25: d, d, i, i, i, i, i, d, d, d, d, d
p26: d, d, i, i, i, i, i, i, d, d, d, d, d
p27: d, d, i, i, i, i, i, i, d, d, d, d, d
p28: d, d, i, i, i, i, i, i, d, d, d, d, d
p29: d, d, i, i, i, i, i, i, d, d, d, d, d
p30: d, d, i, i, i, i, i, i, d, d, d, d, d
217
Program Clauses.

unify(x₁,x₂,x₅) ← p₀(x₁,x₂,x₃,x₅) Λ p₁(x₁,x₂,x₃,x₅)
p₀(x₁,x₂,x₃,x₅) ← p₂(x₁,x₂,x₁₈,x₃,x₅) Λ p₃(x₁,x₂,x₁₈,x₃,x₅)
p₁(x₁,x₂,x₃,x₅) ← p₆(x₁,x₂,x₃,x₄,x₅) Λ p₇(x₁,x₂,x₃,x₄,x₅)
p₂(x₁,x₂,x₁₈,x₃,x₅) ← make_pair(x₁,x₂,x₁₈)
p₃(x₁,x₂,x₁₈,x₃,x₅) ← p₄(x₁,x₂,x₁₈,x₁₉,x₃,x₅) Λ p₅(x₁,x₂,x₁₈,x₁₉,x₃,x₅)
p₆(x₁,x₂,x₃,x₄,x₅) ← empty_seq(x₄)
p₇(x₁,x₂,x₃,x₄,x₅) ← p₈(x₁,x₂,x₃,x₄) Λ p₉(x₁,x₂,x₃,x₄)
p₇(x₁,x₂,x₃,x₄,x₅) ← p₁₀(x₁,x₂,x₃,x₄,x₅,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) Λ p₁₁(x₁,x₂,x₃,x₄,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄)
p₄(x₁,x₂,x₁₈,x₁₉,x₃,x₅) ← empty_stack(x₁₉)
p₅(x₁,x₂,x₁₈,x₁₉,x₃,x₅) ← push(x₁₉,x₁₈,x₃)
p₈(x₁,x₂,x₃,x₄) ← empty_stack(x₃)
p₉(x₁,x₂,x₃,x₄,x₅) ← eq(x₄,x₅)
p₁₀(x₁,x₂,x₃,x₄,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← p₁₂(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) Λ p₁₃(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄)
p₁₁(x₁,x₂,x₃,x₄,x₁₀,x₁₅,x₅) ← eq(x₁₅,x₅)
p₁₂(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← p₁₄(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) Λ p₁₅(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄)
p₁₃(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← p₁₆(x₁,x₂,x₃,x₄,x₆,x₇,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) Λ p₁₇(x₁,x₂,x₃,x₄,x₆,x₇,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄)
p₁₄(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← top(x₃,x₁₀)
p₁₅(x₁,x₂,x₃,x₄,x₆,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← pop(x₃,x₁₀)
p₁₆(x₁,x₂,x₃,x₄,x₆,x₇,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← first(x₁₀,x₇)
p₁₇(x₁,x₂,x₃,x₄,x₆,x₇,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← p₁₈(x₁,x₂,x₃,x₄,x₆,x₇,x₈,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) Λ p₁₉(x₁,x₂,x₃,x₄,x₆,x₇,x₈,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄)
p₁₈(x₁,x₂,x₃,x₄,x₆,x₇,x₈,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ← second(x₁₀,x₈)
p₁₉(x₁,x₂,x₃,x₄,x₆,x₇,x₈,x₁₀,x₁₁,x₁₂,x₁₃,x₁₄) ←
p20(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p19(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p21(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p19(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p22(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p19(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p23(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p19(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p24(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p20(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p25(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p26(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p27(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p21(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p37(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ∧
p38(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ∧
p39(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ∧
p22(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p49(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ∧
p50(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ∧
p51(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ∧
p23(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p61(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p62(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p63(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p24(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p64(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p65(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p66(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p67(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p25(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
p28(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p29(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p26(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
replace_occurrences_in_stack(x22,x6,x13)
p27(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
p35(x1,x2,x3,x4,x6,x7,x8,x22,x23,x10,x11,x12,x13,x14) ∧
p36(x1,x2,x3,x4,x6,x7,x8,x22,x23,x10,x11,x12,x13,x14) ∧
p37(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ←
p40(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ∧
p41(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14)
p38(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ← replace_occurrences_in_stack(x20,x6,x13)
p39(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ←
p47(x1,x2,x3,x4,x6,x7,x8,x20,x21,x10,x11,x12,x13,x14) ∧
p48(x1,x2,x3,x4,x6,x7,x8,x20,x21,x10,x11,x12,x13,x14)
p49(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ←
p52(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ∧
p53(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14)
p50(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ← replace_occurrences_in_stack(x16,x6,x13)
p51(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ←
p59(x1,x2,x3,x4,x6,x7,x8,x16,x17,x10,x11,x12,x13,x14) ∧
p60(x1,x2,x3,x4,x6,x7,x8,x16,x17,x10,x11,x12,x13,x14)
p61(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ← eq(x7,x8)
p62(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ← eq(x6,x13)
p63(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ← eq(x4,x14)
p64(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p68(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p69(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p65(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p76(x1,x2,x3,x4,x6,x7,x8,x9,x10,x11,x12,x13,x14) ∧
p77(x1,x2,x3,x4,x6,x7,x8,x9,x10,x11,x12,x13,x14)
p66(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p80(x1,x2,x3,x4,x6,x7,x8,x27,x10,x11,x12,x13,x14) ∧
p81(x1,x2,x3,x4,x6,x7,x8,x27,x10,x11,x12,x13,x14)
p67(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ← eq(x4,x14)
p28(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
p30(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p31(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ∧
p32(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14)
p29(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
p33(x1,x2,x3,x4,x6,x7,x8,x24,x22,x10,x11,x12,x13,x14) ∧
p34(x1,x2,x3,x4,x6,x7,x8,x24,x22,x10,x11,x12,x13,x14)
p35(x1,x2,x3,x4,x6,x7,x8,x22,x23,x10,x11,x12,x13,x14) ←
make_idempotent_subst1(x22,x4,x23)
p36(x1,x2,x3,x4,x6,x7,x8,x22,x23,x10,x11,x12,x13,x14) ←
add_last_elem(x23,x22,x14)
p40(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ←
p42(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ∧
p43(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14)
p41(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ←
p44(x1,x2,x3,x4,x6,x7,x8,x25,x20,x10,x11,x12,x13,x14) ∧
p45(x1,x2,x3,x4,x6,x7,x8,x25,x20,x10,x11,x12,x13,x14) ∧
p46(x1,x2,x3,x4,x6,x7,x8,x25,x20,x10,x11,x12,x13,x14)
p47(x1,x2,x3,x4,x6,x7,x8,x20,x21,x10,x11,x12,x13,x14) ←
make.idempotent.subst1(x20,x4,x21)
p48(x1,x2,x3,x4,x6,x7,x8,x20,x21,x10,x11,x12,x13,x14) ←
add.last.elem(x21,x20,x14)
p52(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ←
p54(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ∧
p55(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14)
p53(x1,x2,x3,x4,x6,x7,x8,x16,x10,x11,x12,x13,x14) ←
p56(x1,x2,x3,x4,x6,x7,x8,x26,x16,x10,x11,x12,x13,x14) ∧
p57(x1,x2,x3,x4,x6,x7,x8,x26,x16,x10,x11,x12,x13,x14) ∧
p58(x1,x2,x3,x4,x6,x7,x8,x26,x16,x10,x11,x12,x13,x14)
p59(x1,x2,x3,x4,x6,x7,x8,x16,x17,x10,x11,x12,x13,x14) ←
make.idempotent.subst1(x16,x4,x17)
p60(x1,x2,x3,x4,x6,x7,x8,x16,x17,x10,x11,x12,x13,x14) ←
add.last.elem(x17,x16,x14)
p68(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p70(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ∧
p71(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14)
p69(x1,x2,x3,x4,x6,x7,x8,x10,x11,x12,x13,x14) ←
p72(x1,x2,x3,x4,x6,x7,x8,x28,x10,x11,x12,x13,x14) ∧
p73(x1,x2,x3,x4,x6,x7,x8,x28,x10,x11,x12,x13,x14)
p76(x1,x2,x3,x4,x6,x7,x8,x9,x10,x11,x12,x13,x14) ←
arity(x7,x9)
p77(x1,x2,x3,x4,x6,x7,x8,x9,x10,x11,x12,x13,x14) ←
p78(x1,x2,x3,x4,x6,x7,x8,x9,x30,x10,x11,x12,x13,x14) ∧
p79(x1,x2,x3,x4,x6,x7,x8,x9,x30,x10,x11,x12,x13,x14)
p80(x1,x2,x3,x4,x6,x7,x8,x27,x10,x11,x12,x13,x14) ←
f.struct.args(x7,x27)
p81(x1,x2,x3,x4,x6,x7,x8,x27,x10,x11,x12,x13,x14) ←
p82(x1,x2,x3,x4,x6,x7,x8,x27,x31,x10,x11,x12,x13,x14) ∧
p83(x1,x2,x3,x4,x6,x7,x8,x27,x31,x10,x11,x12,x13,x14)
p30(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
var.f.struct(x7)
p31(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
var.f.struct(x8)
p32(x1,x2,x3,x4,x6,x7,x8,x22,x10,x11,x12,x13,x14) ←
eq(x7,x8)
p33(x1,x2,x3,x4,x6,x7,x8,x24,x22,x10,x11,x12,x13,x14) ←
head(x7,x24)
p34(x1,x2,x3,x4,x6,x7,x8,x24,x22,x10,x11,x12,x13,x14) ←
make.pair(x24,x8,x22)
p42(x1,x2,x3,x4,x6,x7,x8,x20,x10,x11,x12,x13,x14) ←
var.f.struct(x7)
Examples of Sample Runs of the Predicate Unify/3.

The following examples illustrates the results from sample runs of the predicate Unify(T1, T2, Subst) for certain values of T1 and T2. Terms and substitutions are also presented in non-ground form because they are more readable by the reader than the ground one.

1. Ground representation
   
   T1 = ['p', '(', 'X', ',', 'Y', ',', 'Z', ',' ,')', ')']
   
   T2 = ['p', '(', 'b', ',', 'Z', ')']
   
   Subst = [(X,[b]),(Z,[f(a,Y)])]
   
   Non-ground representation
   
   T1 = p(X,f(a,Y)), T2 = p(b,Z) and Subst = {X/b, Z/f(a,Y)}

2. Ground representation
   
   T1 = ['p', '(', 'a', ',', 'X', ',', 'Y', ',', 'Z', ',')', ')']
   
   T2 = ['p', '(', 'Y', ',', 'Z', ',', 'h', ',', 'h', ',', 'Y', ',' ,')', ')', ')']
   
   Subst = [(Z,[a]),(X,[h,(g,(a,))]),(Y,[g,(a,)])]
Non-ground representation

\[ T_1 = p(a, X, h(g(Z))), \quad T_2 = p(Z, h(Y), h(Y)), \quad \text{Subst} = \{ Z/a, X/h(g(a)), Y/g(a) \} \]

3. Occur check.

Ground representation

\[ T_1 = ['p', '(', 'X', ',', 'Y', ',', 'X', ')'] \]
\[ T_2 = ['p', '(', 'Y', ',', 'Y', ',', 'Y', ')'] \]

failure

Non-ground representation

\[ T_1 = p(X, X), \quad T_2 = p(Y, f(Y)) \]

4. Variant terms.

Ground representation

\[ T_1 = ['p', '(', 'f', '(', 'Z', ',', 'X', ',', 'Y', ')', ',', 'g', '(', 'a', ')'] \]
\[ T_2 = ['y', '(', 'f', '(', 'Y', ',', 'X', ',', 'Y')', ',', 'g', '(', 'a')'] \]

\[ \text{Subst} = \{(X, [Y]), (Z, [U])\} \]

Non-ground representation

\[ T_1 = p(f(X, Y), g(Z), a), \quad T_2 = p(f(Y, X), g(U), a), \quad \text{Subst} = \{X/Y, Z/U\} \]
Appendix C

Examples of Correctness Proofs

The notation $x, y, z/\tau$ means that the variables $x, y, z$ have the same type $\tau$. In places where parentheses are omitted the precedence of the logical connectives is as follows (highest) $\neg, \land, \lor, \rightarrow, \leftarrow$ (lowest). The left-to-right order is assumed for logical connectives of the same precedence.

C.1 Sequence Length Even

1. Informal specification: The predicate $\text{isEven}(s)$ where $\text{Type}(\text{isEven}) = \text{seq}(a)$ is true iff the length of the sequence $s$ is even.

2. Logic specification for predicate $\text{isEven}^S(s)$ ($\text{Spec} = \{\text{Spec}_{\text{isEven}^S}, \text{Spec}_{\text{even}^S}\}$):

   $\forall s/\text{seq}(a) (\text{isEven}^S(s) \leftrightarrow \exists n/N(\text{length_seq}(s, n) \land \text{even}^S(n)))$

   $\forall n/N (\text{even}^S(n) \rightarrow (n = 0 \lor \exists m/N, n1/N (n = \text{succ}(m) \land m = \text{succ}(n1) \land \text{even}^S(n1))))$

   The logic specification is in structured form. Note that $\text{length_seq}/2$ is a DT operation.

3. Axioms - Lemmas - Specifications of DT operations (A):

   **Axioms**

   **A1** Domain closure axiom for naturals
   $\forall n/N (n = 0 \lor \exists m/N (n = \text{succ}(m)))$

   **A2** Uniqueness axioms for naturals
i. \( \forall n/N (\neg (\text{suc}(n) = 0)) \)

ii. \( \forall n_1, n_2/N (\text{suc}(n_1) = \text{suc}(n_2) \rightarrow n_1 = n_2) \)

A3 \( \forall n/N (n > 0 \rightarrow \neg n = 0) \)

A4 \( \forall n/N (n = 0 + n) \)

**Lemmas**

L1 \( \forall n/N (n \neq 0 \rightarrow \exists m/N (n = \text{suc}(m) \rightarrow m = \text{pred}(n))) \)

L2 \( \forall n/N (n \neq 0 \rightarrow \exists m/N (m = \text{pred}(n) \rightarrow n > m)) \)

L3 \( \forall n/N (\exists k, m/N (n = \text{suc}(k) \land k = \text{suc}(m)) \land m > 1) \)

L4 \( \forall n, m, k/N (n > k \land m = n - k \rightarrow m > 0) \)

L5 \( \forall n/N (n > 1 \rightarrow n > 0) \)

L6 \( \forall n/N (n \neq 0 \rightarrow \exists m/N m = \text{suc}(n)) \)

**Logic specifications of DT operations**

\( \forall q/\text{seq}(a), n/N (\text{length.seq}(q, n) \leftarrow ((q = <> \land n = 0) \lor \exists a, t/\text{seq}(\alpha), n_1/N (q = h :: t \land \text{length.seq}(t, n_1) \land n = n_1 + 1))) \)

\( \forall n_1, n_2/N (\text{succ.nat}(n_1, n_2) \leftarrow n_2 = n_1 + 1) \)

\( \forall n, n_1/N (\text{pred.nat}(n, n_1) \leftarrow n > 0 \land n_1 = n - 1) \)

\( \forall n_1, n_2/N (\text{gt.nat}(n_1, n_2) \leftarrow n_1 > n_2) \)

4. Constructed SI-program without the definitions of DT operations (Pr):

\begin{align*}
\text{isEven}(s) & \leftarrow p1(s, n) \land p2(s, n) \\
p1(s, n) & \leftarrow \text{length.seq}(s, n) \\
p2(s, n) & \leftarrow p3(s, n) \land p4(s, n) \\
p2(s, n) & \leftarrow \neg p3(s, n) \land p5(s, n, s1, n1) \land p6(s, n, v) \\
p3(s, n) & \leftarrow n = 0 \\
p4(s, n) & \leftarrow \text{true} \\
p5(s, n, v, s1, n1) & \leftarrow p7(s, n, v, s1, n1) \land p8(s, n, v, s1, n1) \\
p7(s, n, v, s1, n1) & \leftarrow p9(s, n, v, v, s1, n1) \land p10(s, n, v, v, s1, n1) \\
p9(s, n, v, v, s1, n1) & \leftarrow v1 = 0 \\
p10(s, n, v, v, s1, n1) & \leftarrow p11(s, n, v, v, v, s1, n1) \land p12(s, n, v, v, v, s1, n1) \\
p11(s, n, v, v, v, s1, n1) & \leftarrow \text{succ.nat}(v1, v2) \\
p12(s, n, v, v, v, s1, n1) & \leftarrow \text{gt.nat}(n, v2) \\
p8(s, n, v, s1, n1) & \leftarrow p13(s, n, v, v, s1, n1) \land p14(s, n, v, v, s1, n1) \\
p13(s, n, v, v, s1, n1) & \leftarrow \text{pred.nat}(v3, n) \\
p14(s, n, v, v, s1, n1) & \leftarrow \text{pred.nat}(v3, n) \\
p6(s, n, v) & \leftarrow \text{true}
\end{align*}

5. Completion of the logic program of isEven(s) (comp(Pr)):
∀s/seq(a) (isEven(s) → ∃n/N (p1(s, n) ∧ p2(s, n)))
∀seq(a), n/N (p1(s, n) → length_seq(s, n))
∀seq(a), n/N (p2(s, n) →
(p3(s, n) ∧ p4(s, n)) ∨ ∃v/N, s1/seq(a), n1/N (¬p3(s, n) ∧ p5(s, n, v, s1, n1) ∧ p2(s1, n1) ∧ p6(s, n, v))
∀s/seq(a), n/N (p3(s, n) → n = 0)
∀s/seq(a), n/N (p4(s, n) → true)
∀s/seq(a), n/N, v/N, s1/seq(a), n1/N (p5(s, n, v, s1, n1) → p7(s, n, v, s1, n1))
∀s/seq(a), n/N, v/N, s1/seq(a), n1/N (p7(s, n, v, s1, n1) →
∃v1/N (p9(s, n, v1, v1, s1, n1) ∧ p10(s, n, v1, v1, s1, n1)))
∀s/seq(a), n/N, v1/N, v/N, s1/seq(a), n1/N (p9(s, n, v1, v1, s1, n1) → v1 = 0)
∀s/seq(a), n/N, v1/N, v2/N, v/N, s1/seq(a), n1/N (p10(s, n, v1, v1, s1, n1) → ∃v2/N (p11(s, n, v1, v2, v, s1, n1) ∧ p12(s, n, v1, v2, v, s1, n1)))
∀s/seq(a), n/N, v1/N, v2/N, v/N, s1/seq(a), n1/N (p11(s, n, v1, v2, v, s1, n1) ↔ succ_nat(v1, v2))
∀s/seq(a), n/N, v1/N, v2/N, v/N, s1/seq(a), n1/N (p12(s, n, v1, v2, v, s1, n1) ↔ gt_nat(n, v2))
∀s/seq(a), n/N, v/N, s1/seq(a), n1/seq(a) (p8(s, n, v, s1, n1) ↔
∃v3/N (p13(s, n, v3, v, s1, n1) ∧ p14(s, n, v3, v, s1, n1))
∀s/seq(a), n/N, v3/N, v/N, s1/seq(a), n1/N (p13(s, n, v3, v, s1, n1) ↔ pred_nat(v3, n1))
∀s/seq(a), n/N, v3/N, v/N, s1/seq(a), n1/N (p14(s, n, v3, v, s1, n1) ↔ pred_nat(v3, n1))
∀s/seq(a), n/N, v/N (p6(s, n, v) → true)

6. Correctness theorem and theory to prove it.

comp(Pr) ∪ Spec ∪ A ⊢ ∀s/seq(a) (isEven(s) ↔ isEvenS(s))

7. Correctness theorem:

∀s/seq(a) (isEven(s) ↔ isEvenS(s))

8. Proof: Subgoal proof scheme:

∀s/seq(a) (isEven(s) ↔ isEvenS(s)

by definition of isEvenS(s)

↔ ∃n/N (length_seq(s, n) ∧ evenS(n)))
because of the completion of \( \text{isEven}/1 \)
\[
\forall s/\text{seq}(\alpha) \ (\exists n/N \ (p1(s, n) \land p2(s, n))) \quad \Leftrightarrow \quad \text{-/-}
\]

because of the completion of \( p1/2 \)
\[
\forall s/\text{seq}(\alpha) \ (\exists n/N \ (\text{length}\_\text{seq}(s, n) \land p2(s, n))) \quad \Leftrightarrow \quad \text{-/-}
\]

Therefore, in order to establish the equivalence we have to prove the following theorem,
\[
\forall s/\text{seq}(\alpha), n/N (p2(s, n) \leftrightarrow \text{even}^S(n))
\]

**Incremental proof scheme**

We apply structural induction with induction parameter \( n/N \). The set \( (N, <) \) is well-founded.

**Induction base:** \( n = 0 \)
\[
\forall n/N \ (n = 0 \rightarrow \forall s/\text{seq}(\alpha) \ (p2(s, n) \rightarrow \text{even}^S(n)))
\]
\[
\forall s/\text{seq}(\alpha), n/N (p2(s, n) \rightarrow \text{even}^S(n))
\]

by definition of \( \text{even}^S(n) \)
\[
\Leftrightarrow (n = 0 \lor \exists v3, n1/N \ (n = \text{succ}(v3) \land v3 = \text{succ}(n1) \land \text{even}^S(n1)))
\]

by axiom A2.i
\[
\Leftrightarrow (n = 0 \lor \exists v3, n1/N \ (\text{false} \land v3 = \text{succ}(n1) \land \text{even}^S(n1)))
\]

because \( n = 0 \) and by FOL laws (\( P \land \text{false} \leftrightarrow \text{false}; P \lor \text{false} \leftrightarrow P \))
\[
\Leftrightarrow (n = 0)
\]

by FOL laws (\( P \land \text{true} \leftrightarrow P \))
\[
\Leftrightarrow n = 0 \land \text{true}
\]

because of the completion of \( p3(s, n) \) and \( p4(s, n) \)
\[
\Leftrightarrow p3(s, n) \land p4(s, n)
\]

because of the completion of \( p2(s, n) \)
\[
\Leftrightarrow p2(s, n)
\]

**Induction step:** \( n \neq 0 \)
\[
\forall s/\text{seq}(\alpha), n/N (n \neq 0 \rightarrow \forall n1/N \ (n1 < n \rightarrow
\]

228
\[ \forall s/seq(\alpha) \ (p2(s, n) \leftrightarrow even^S(n)) \]

\[ \forall s/seq(\alpha), n/N \ (p2(s, n) \leftrightarrow even^S(n)) \]

by definition of \( even^S(n) \)

\[ \rightarrow (n = 0 \lor \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land even^S(n1))) \]

by axioms A2.i and by lemma L6

\[ \rightarrow (false \lor \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land even^S(n1))) \]

by FOL laws \( (P \lor false \leftrightarrow P) \)

\[ \rightarrow \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land even^S(n1))) \]

by lemma L1 and by FOL laws \( (if \ P \rightarrow Q \ then \ P \leftrightarrow P \land Q) \)

\[ \rightarrow \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land n1 = pred(v3) \land v3 = pred(n) \land even^S(n1))) \]

by lemma L2 and by transitivity of \( < \) follows \( (n1 < n) \)

and by induction hypothesis

\[ \rightarrow \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land n1 = pred(v3) \land v3 = pred(n) \land p2(s, n1))) \]

by definition of function \( pred(n) \)

\[ \rightarrow \exists n1/N, v3/N \ (n = succ(v3) \land v3 = succ(n1) \land n1 = v3 - 1 \land v3 = n - 1 \land p2(s, n1))) \]

by lemma L3

\[ \rightarrow \exists n1/N, v3/N \ (n > 1 \land n1 = v3 - 1 \land v3 = n - 1 \land p2(s, n1))) \]

by lemma L4 and by FOL laws \( (if \ P \rightarrow Q \ then \ P \leftrightarrow P \land Q) \)

\[ \rightarrow \exists n1/N, v3/N \ (n > 1 \land v3 > 0 \land n1 = v3 - 1 \land v3 = n - 1 \land p2(s, n1))) \]

by lemma L5 and by FOL laws \( (if \ P \rightarrow Q \ then \ P \leftrightarrow P \land Q) \)

\[ \rightarrow \exists n1/N, v3/N \ (n > 0 \land n > 1 \land v3 > 0 \land n1 = v3 - 1 \land v3 = n - 1 \land p2(s, n1))) \]
by FOL laws ($P \leftrightarrow P \land P; \land \text{commutativity}$)

$\exists n_1/N, v_3/N \ (n > 0 \land n > 1 \land v_3 > 0 \land n_1 = v_3 - 1 \land n > 0 \land v_3 = n - 1 \land p_2(s, n_1))$

by axiom A3

$\exists n_1/N, v_3/N \ (\neg n = 0 \land n > 1 \land v_3 > 0 \land n_1 = v_3 - 1 \land n > 0 \land v_3 = n - 1 \land p_2(s, n_1))$

by introduction of a new variable $v_2/N$ and by FOL laws

$\forall x \ (P(x) \leftrightarrow \exists y (x = y \land P(y)))); \land \text{associativity};$

$A \land \exists x P \leftrightarrow \exists x (A \land P) \text{ if } x \text{ is not free in } A$

$\exists n_1/N, v_3/N, v_2/N \ (\neg n = 0 \land n > v_2 \land v_2 = 1 \land v_3 > 0 \land n_1 = v_3 - 1 \land n > 0 \land v_3 = n - 1 \land p_2(s, n_1))$

by axiom A4

$\exists n_1/N, v_2/N, v_3/N \ (\neg n = 0 \land n > v_2 \land v_2 = 0 + 1 \land v_3 > 0 \land n_1 = v_3 - 1 \land n > 0 \land v_3 = n - 1 \land p_2(s, n_1))$

by introduction of a new variable $v_1/N$ and by FOL laws

$\forall x \ (P(x) \leftrightarrow \exists y (x = y \land P(y)))); \land \text{associativity};$

$A \land \exists x P \leftrightarrow \exists x (A \land P) \text{ if } x \text{ is not free in } A$

$\exists n_1/N, v_2/N, v_3/N, v_1/N \ (\neg n = 0 \land n > v_2 \land v_2 = v_1 + 1 \land v_1 = 0 \land v_3 > 0 \land n_1 = v_3 - 1 \land n > 0 \land v_3 = n - 1 \land p_2(s, n_1))$

because of the specifications of the DT operations

$succ.nat/2, pred.nat/2 \text{ and } gt.nat/2$

$\exists n_1/N, v_2/N, v_3/N, v_1/N \ (\neg n = 0 \land gt.nat(n, v_2) \land succ.nat(v_1, v_2) \land v_1 = 0 \land pred.nat(v_3, n_1) \land pred.nat(v_3, n_1))$

because of the completion of $p_3/2, p_9/6, p_{11}/7, p_{12}/7, p_{13}/6, p_{14}/6$

$\exists n_1/N, v_2/N, v_3/N, v_1/N, v/N, s_1/seq(\alpha)$

$(-p_3(s, n) \land p_12(s, n, v_1, v_2, v, s_1, n_1) \land p_{11}(s, n, v_1, v_2, v, s_1, n_1) \land p_9(s, n, v_1, v, s_1, n_1) \land p_{14}(s, n, v_3, v, s_1, n_1) \land p_{13}(s, n, v_3, v, s_1, n_1) \land p_2(s, n_1))$

because of the completion of $p_{10}/6$ and $p_{8}/5$ and by FOL laws

($\land \text{commutativity}$)

$\exists n_1/N, v_1/N, v/N, s_1/seq(\alpha) \ (\neg p_3(s, n) \land p_{10}(s, n, v_1, v, s_1, n_1) \land p_9(s, n, v_1, v, s_1, n_1) \land p_8(s, n, v, s_1, n_1) \land p_2(s, n_1))$
by FOL laws ($P \iff P \land \text{true}$)

\[ \iff \exists n_1/N, v_1/N, v/N, s_1/\text{seq}(\alpha) (-p_3(s, n) \land p_5(s, n, v, s_1, n_1) \land p_8(s, n, v, s_1, n_1) \land \text{true} \land p_2(s, n_1)) \]

because of the completion of $p_7/5$ and by FOL laws ($\land \text{commutativity}$)

\[ \iff \exists n_1/N, v_1/N, v/N, s_1/\text{seq}(\alpha) (-p_3(s, n) \land p_5(s, n, v, s_1, n_1) \land p_2(s, n_1) \land \text{true}) \]

because of the completion of $p_5/5$

\[ \iff \exists n_1/N, v_1/N, v/N, s_1/\text{seq}(\alpha) (-p_3(s, n) \land p_5(s, n, v, s_1, n_1) \land p_5(s, n_1) \land \text{true}) \]

because of the completion of $p_6/3$

\[ \iff \exists n_1/N, v_1/N, v/N, s_1/\text{seq}(\alpha) (-p_3(s, n) \land p_5(s, n, v, s_1, n_1) \land p_5(s, n_1) \land \text{true}) \]

because of the completion of $p_2/2$

\[ \iff p_2(s, n) \]

C.2 Ordered Sequence

1. Informal specification: The predicate $\text{incrOrd}(s)$ where $\text{Type}(s) = \text{seq}(Z)$ is true iff the elements of sequence $s$ are in increasing order.

2. Logic specification of the relation $\text{incrOrd}(s)$ ($\text{Spec} = \{\text{Spec}_{\text{incrOrd}}\}$):

\[ \forall s/\text{seq}(Z) (\text{incrOrd}(s) \iff \forall i/N_1 (1 \leq i < (\#s - 1) \rightarrow s_i \leq s_{(i+1)})) \]

Structured form of logic specification ($\text{Spec}' = \{\text{Spec}'_{\text{incrOrd}}\}$):

\[ \forall s/\text{seq}(Z) (\text{incrOrd}(s)) \]

\[ \iff \exists h, h_1/Z, t, t_1/\text{seq}(Z) (s = <> \lor (s = h :: t \land (t = <> \lor (t = h_1 :: t_1 \land h \leq h_1 \land \text{incrOrd}(t)))))) \]

3. Axioms - Lemmas - Specifications of DT operations ($A$):

\textbf{Axioms}

\textbf{A1} Domain closure axiom for sequences

\[ \forall s/\text{seq}(\alpha) (s = <> \lor \exists h/\alpha, t/\text{seq}(\alpha) s = h :: t) \]

\textbf{A2} Uniqueness axioms for sequences

\[ \text{i. } \forall h/\alpha, s/\text{seq}(\alpha) (\neg (h :: t = <>)) \]
ii. $\forall h_1, h_2 / \alpha, t_1, t_2 / \text{seq}(\alpha) (h_1 :: t_1 = h_2 :: t_2 \rightarrow (h_1 = h_2 \land t_1 = t_2))$

A3 $\forall s / \text{seq}(\alpha) (s = <> \rightarrow \#s = 0)$

**Lemmas**

L1 $\forall s / \text{seq}(\alpha) (s \neq <> \rightarrow \exists h / \alpha, t / \text{seq}(\alpha) s = h :: t)$

L2 $\forall s / \text{seq}(\alpha) (\exists t / \text{seq}(\alpha), h / \alpha (s = h :: t \land t = <>)) \rightarrow \#s = 1$

L3 $\forall s / \text{seq}(\alpha) (\exists t_1 / \text{seq}(\alpha), h_1 / \alpha (s = h :: t \land t = h_1 :: t_1)) \rightarrow \#s \geq 2$

L4 $\forall i / N_1, k / N_1 (1 \leq i \leq k \rightarrow i = 1 \lor (2 \leq i \leq k))$

L5 $\forall s / \text{seq}(\alpha) (\exists h / \alpha, t / \text{seq}(\alpha) (s = h :: t \rightarrow \forall i / N_1 (2 \leq i \leq \#s \rightarrow s_i = t_{i-1}))

L6 $\forall s / \text{seq}(\alpha) (\exists t / \text{seq}(\alpha), h / \alpha (s = h :: t \rightarrow \#s = \#t + 1))$

L7 $\forall s / \text{seq}(\alpha), i / N_1 (2 \leq i \leq \#s \rightarrow s_{i-1} \leq s_i) \rightarrow (1 \leq i \leq \#s - 1 \rightarrow s_i \leq s_{i+1})$

L8 $\forall s / \text{seq}(\alpha) (\exists h / \alpha, t / \text{seq}(\alpha) (s = h :: t \rightarrow h = s_1)$

L9 $\forall s / \text{seq}(\alpha) (\exists h_1 / \alpha, t_1 / \text{seq}(\alpha) (s = h :: t \land t = h_1 :: t_1 \rightarrow s_2 = h_1)$

**Logic specifications of DT operations**

$\forall q / \text{seq}(\alpha) (\text{empty}_seq(q) \leftarrow q = <>)$

$\forall q / \text{seq}(\alpha), h / \alpha (\text{head}(q, h) \rightarrow q \neq <> \land \exists t / \text{seq}(\alpha) q = h :: t)$

$\forall q / \text{seq}(\alpha) (\text{tail}(q, t) \leftarrow q \neq <> \land \exists h / \alpha q = h :: t)$

$\forall n_1, n_2 / Z (\text{le}_\text{int}(n_1, n_2) \leftarrow n_1 \leq n_2)$

4. Constructed SI-program without the definitions of DT operations $(Pr)$:

- $\text{incrOrd}(s) \leftarrow p1(s) \land p2(s)$
- $\text{incrOrd}(s) \leftarrow \neg p1(s) \land p3(s, v, s_1) \land \text{incrOrd}(s_1) \land p4(s, v)$
- $p1(s) \leftarrow p5(s)$
- $p1(s) \leftarrow p6(s)$
- $p5(s) \leftarrow \text{empty}_seq(s)$
- $p6(s) \leftarrow p7(s, q_1) \land p8(s, q_1)$
- $p7(s, q_1) \leftarrow p9(s, q_1) \land p10(s, q_1)$
- $p9(s, q_1) \leftarrow \neg \text{empty}_seq(s)$
- $p10(s, q_1) \leftarrow \text{tail}(s, q_1)$
- $p8(s, q_1) \leftarrow \text{empty}_seq(q_1)$
- $p2(s) \leftarrow \text{true}$
- $p3(s, v, s_1) \leftarrow p11(s, v, s_1) \land p12(s, v, s_1) \land p13(s, v, s_1)$
- $p11(s, v, s_1) \leftarrow \text{head}(s, v)$
- $p12(s, v, s_1) \leftarrow \text{tail}(s, s_1)$
- $p13(s, v, s_1) \leftarrow p14(s, v_1, v, s_1) \land p15(s, v_1, v, s_1)$
- $p14(s, v_1, v, s_1) \leftarrow \text{head}(s_1, v_1)$
- $p15(s, v_1, v, s_1) \leftarrow \text{le}_\text{int}(v, v_1)$
- $p4(s, v) \leftarrow \text{true}$
5. Completion of the logic program of \( \text{incrOrd}(s) \) (\( \text{comp}(Pr) \)):

\[
\forall s/\text{seq}(Z) \ (\text{incrOrd}(s) \rightarrow (p1(s) \land p2(s))
\]

\[
\land \exists v/Z, s1/\text{seq}(Z) \ (-p1(s) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v))
\]

\[
\forall s/\text{seq}(Z) \ (p1(s) \rightarrow (p5(s) \lor p6(s))
\]

\[
\forall s/\text{seq}(Z) \ (p5(s) \rightarrow \text{empty_seq}(s))
\]

\[
\forall s/\text{seq}(Z) \ p6(s) \rightarrow \exists q1/\text{seq}(Z) (p7(s,q1) \land p8(s,q1))
\]

\[
\forall s,q1/\text{seq}(Z) (p7(s,q1) \rightarrow p9(s,q1) \land p10(s,q1))
\]

\[
\forall s,q1/\text{seq}(Z) \ (p9(s,q1) \rightarrow \neg \text{empty_seq}(s))
\]

\[
\forall s,q1/\text{seq}(Z) \ (p10(s,q1) \rightarrow \text{tail}(s,q1))
\]

\[
\forall s,q1/\text{seq}(Z) \ (p8(s,q1) \rightarrow \text{empty_seq}(q1))
\]

\[
\forall s,sl/\text{seq}(Z) \ (p2(s) \rightarrow \text{true})
\]

\[
\forall s,sl/\text{seq}(Z), v/Z \ (p3(s,v,sl) \rightarrow p11(s,v,sl) \land p12(s,v,sl) \land p13(s,v,sl))
\]

\[
\forall s,sl/\text{seq}(Z), v/Z \ (p11(s,v,sl) \rightarrow \text{head}(s,v))
\]

\[
\forall s,sl/\text{seq}(Z), v/Z \ (p12(s,v,sl) \rightarrow \text{tail}(s,sl))
\]

\[
\forall s,v,sl \ (p13(s,v,sl) \rightarrow \exists q1/Z (p14(s,v1,v,sl) \land p15(s,v1,v,sl)))
\]

\[
\forall s,sl/\text{seq}(Z), v,v1/Z \ (p14(s,v1,v,sl) \rightarrow \text{head}(sl,v1))
\]

\[
\forall s,sl/\text{seq}(Z), v,v1/Z \ (p15(s,v1,v,sl) \rightarrow \text{le}.\text{int}(v,v1))
\]

\[
\forall s,sl/\text{seq}(Z), v/Z \ (p4(s,v) \rightarrow \text{true})
\]

6. Correctness theorem and theory to prove it.

\[
\text{comp}(Pr) \cup \text{Spec} \cup \text{A} \models \forall s/\text{seq}(\alpha) \ (\text{incrOrd}(s) \rightarrow \text{incrOrd}^S(s))
\]


\[
\forall s/\text{seq}(Z) \ (\text{incrOrd}^S(s)
\]

\[
\rightarrow \forall i/N_1 \ (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)})
\]

by axiom A1 and by FOL laws (if \( Q \rightarrow \text{true} \) then \( P \land Q \rightarrow P \))

\[
\rightarrow ((s =<> \lor \exists h/Z, t/\text{seq}(Z) s = h :: t) \land
\]

\[
\forall i/N_1 \ (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)})
\]

by FOL laws (\( \land \) distribution)

\[
\rightarrow (s =<> \land \forall i/N_1 \ (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}) \lor
\]

\[
(\exists h/Z, t/\text{seq}(Z) s = h :: t \land \forall i/N_1 \ (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))
\]

by axiom A3

\[
\rightarrow (s =<> \land \forall i/N_1 \ (1 \leq i \leq -1 \rightarrow s_i \leq s_{(i+1)}) \lor
\]

\[
(\exists h/Z, t/\text{seq}(Z) s = h :: t \land \forall i/N_1 \ (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))
\]
by definition of the relation \( \leq \)

\[ (s = \langle \rangle \land \forall i/N_1 (\text{false} \rightarrow s_i \leq s_{i+1})) \lor \exists h/Z, t/\text{seq}(Z) (s = h \cdot t \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1})) \]

by FOL laws (false \( \rightarrow P \rightarrow \text{true} \))

\[ (s = \langle \rangle \lor \exists h/Z, t/\text{seq}(Z) (s = h \cdot t \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1})) \]

by FOL laws (P \( \land \text{true} \rightarrow P \))

\[ (s = \langle \rangle \lor \exists h/Z, t/\text{seq}(Z) (s = h \cdot t \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1})) \]

by axiom A1 and by FOL laws (if Q \( \rightarrow \text{true} \) then P \( \land \) Q \( \rightarrow \) P)

\[ (s = \langle \rangle \lor \exists h/Z, t/\text{seq}(Z) (s = h \cdot t \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1}))) \]

by FOL laws (A \( \lor \exists xP \rightarrow \exists x(A \lor P) \) if x is not free in A)

\[ \exists h/Z, t/\text{seq}(Z) (s = \langle \rangle \lor (s = h \cdot t \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1}))) \]

by FOL laws (\( \land \) distribution)

\[ \exists h/Z, t/\text{seq}(Z) (s = \langle \rangle \lor (((s = h \cdot t \land t = \langle \rangle) \lor (s = h \cdot t \land \exists h1/Z, t1/\text{seq}(Z) t = h1 \cdot t1) \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1})))) \]

by FOL laws (\( \land \) distribution; A \( \land \exists xP \rightarrow \exists x(A \land P) \) if x is not free in A)

\[ \exists h/Z, t/\text{seq}(Z) (s = \langle \rangle \lor ((s = h \cdot t \land t = \langle \rangle) \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1})) \lor \exists h1/Z, t1/\text{seq}(Z) (s = h \cdot t \land h1 \cdot t1) \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1}))) \]

by FOL laws (A \( \lor \exists xP \rightarrow \exists x(A \lor P) \) if x is not free in A)

\[ \exists h, h1/Z, t, t1/\text{seq}(Z) (s = \langle \rangle \lor ((s = h \cdot t \land t = \langle \rangle) \lor \exists i/N_1 (1 \leq i < (#s - 1) \rightarrow s_i \leq s_{i+1})) \lor (s = h \cdot t \land t = h1 \cdot t1 \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{i+1}))) \]

by lemma L2
\[\exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor ((s = h :: t \land t = <>) \land
\forall i/N_1 (1 \leq i < 0 \rightarrow s_i \leq s_{(i+1)})) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by definition of relation \(\leq\)

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor ((s = h :: t \land t = <>) \land
\forall i/N_1 (false \rightarrow s_i \leq s_{(i+1)})) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by FOL laws (\(\lor\) associativity; false \(\rightarrow\) \(P \rightarrow\) true)

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <> \land true) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 (1 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by lemmas L3 and L4

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 (i = 1 \lor 2 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by FOL laws ((\(P \lor Q\) \(\rightarrow\) \(R \rightarrow (P \rightarrow R) \land (Q \rightarrow R)\))

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 ((i = 1 \rightarrow s_i \leq s_{(i+1)})) \land
(2 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by FOL laws (\(\forall x (P \land Q) \rightarrow \forall x P \land \forall x Q; \land\) associativity)

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land
t = h_1 :: t_1 \land \forall i/N_1 (i = 1 \rightarrow s_i \leq s_{(i+1)})) \land
\forall i/N_1 (2 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by FOL laws (one-point law: \(\forall x (x = E \rightarrow P) \rightarrow Px/E\))

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land
t = h_1 :: t_1 \land s_1 \leq s_2 \land
\forall i/N_1 (2 \leq i \leq (#s - 1) \rightarrow s_i \leq s_{(i+1)}))))\]

by lemmas L5 and L6

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land
t = h_1 :: t_1 \land s_1 \leq s_2 \land
\forall i/N_1 (2 \leq i \leq (#s - 1) \rightarrow t_{i-1} \leq t_{(i+1)}))))\]

by lemma L7

\[\Rightarrow \exists h, h_1/Z, t, t_1/seq(Z) (s = <> \lor (s = h :: t \land t = <>) \lor (s = h :: t \land t = h_1 :: t_1 \land s_1 \leq s_2 \land
(235)\]
\forall i/N_1 \ (1 \leq i \leq \#t - 1 \rightarrow t_i \leq t_{i+1}))}

by lemmas L8 and L9

\[ \Rightarrow \exists h, h1/Z, t, t1/seq(Z) \ (s =<> \ \vee (s = h :: t \land t =<>)) \ \vee \\
(s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \\
\forall i/N_1 \ (1 \leq i \leq \#t - 1 \rightarrow t_i \leq t_{i+1}))\]

by folding

\[ \Rightarrow \exists h, h1/Z, t, t1/seq(Z) \ (s =<> \ \vee (s = h :: t \land t =<>) \ \vee \\
(s = h :: t \land t = h1 :: t1 \land h \leq h1 \land incrOrdS(t)))\]

by FOL laws (\land distribution)

\[ \Rightarrow \exists h, h1/Z, t, t1/seq(Z) \ (s =<> \ \vee (s = h :: t \land t =<>) \ \vee \\
(t = h1 :: t1 \land h \leq h1 \land incrOrdS(t)))\]

**Incremental proof scheme:**

**Induction base:** \( s =<> \)

\[ \forall s/seq(Z) \ (incrOrd(s)) \]

\[ \Rightarrow \exists h, h1/Z, t, t1/seq(Z) \ (s =<> \ \vee (s = h :: t \land t =<>) \ \vee \\
(t = h1 :: t1 \land h \leq h1 \land incrOrdS(t)))\]

by axiom A2.i

\[ \Rightarrow \exists h, h1/Z, t, t1/seq(Z) \ (s =<> \ \vee (false \land (t =<>) \ \vee \\
t = h1 :: t1 \land h \leq h1 \land incrOrdS(t)))\]

by FOL laws (\false \land P \leftrightarrow \false)

\[ \Rightarrow (s =<> \ \vee false \ \vee false)\]

by FOL laws (\false \lor P \leftrightarrow P)

\[ \Rightarrow s =<>\]

because of the completion of incrOrd/1

\[ \forall s/seq(Z) \ ((p1(s) \land p2(s)) \lor \exists v/Z, s1/seq(Z) \ (\neg p1(s) \land \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \leftrightarrow */- \]

because of the completion of p1/1

\[ \forall s/seq(Z) \ ((p5(s) \lor p6(s)) \land p2(s)) \lor \exists v/Z, s1/seq(Z) \ (\neg (p5(s) \lor \]
\[ p6(s) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v) \]  

because of the completion of \( p5/1 \) and \( p6/1 \)

\[
\forall s/seq(Z) \ ( (empty_seq(s) \lor \exists q1/seq(Z) \\
(p7(s, q1) \land p8(s, q1)) \land p2(s)) \lor \exists v/Z, s1/seq(Z) \\
(\neg (empty_seq(s) \lor \exists q1/seq(Z) \ (p7(s, q1) \land p8(s, q1)) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v)) )
\]

by FOL laws (\( A \land \exists x P \rightarrow \exists x (A \land P) \) if \( x \) is not free in \( A \); \( A \lor \exists x P \rightarrow \exists x (A \lor P) \) if \( x \) is not free in \( A \);
\[
\exists x (P \lor Q) \rightarrow \exists x P \lor \exists x Q)
\]

\[
\forall s/seq(Z) \exists q1/seq(Z), v/Z, s1/seq(Z) \\
( (empty_seq(s) \lor (p7(s, q1) \land p8(s, q1)) \land p2(s)) \lor \\
(\neg (empty_seq(s) \lor (p7(s, q1) \land p8(s, q1))) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))
\]

because of the specifications of the DT operation \( empty_seq/1 \)

\[
\forall s/seq(Z) \exists q1/seq(Z), v/Z, s1/seq(Z) \\
((s =<> \lor (p7(s, q1) \land p8(s, q1)) \land p2(s)) \\
\lor (\neg (s =<> \lor (p7(s, q1) \land p8(s, q1))) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))
\]

because of the completion of \( p7/2 \) and by FOL laws (\( \land \) associativity)

\[
\forall s/seq(Z) \exists q1/seq(Z), v/Z, s1/seq(Z) \\
((s =<> \lor (p9(s, q1) \land p10(s, q1) \land p8(s, q1)) \land \\
p2(s)) \lor (\neg (s =<> \lor (p9(s, q1) \land p10(s, q1) \land \\
p8(s, q1))) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))
\]

because of the completion of \( p8/2, p9/2 \) and \( p10/2 \)

\[
\forall s/seq(Z) (\exists q1/seq(Z), v/Z, s1/seq(Z) \\
((s =<> \lor (\neg empty_seq(s) \land tail(s, q1) \land empty_seq(q1)) \land \\
p2(s)) \lor (\neg (s =<> \lor (\neg empty_seq(s) \land tail(s, q1) \land \\
empty_seq(q1))) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v)) \rightarrow -/-)
\]

because of the specifications of the DT operations \( empty_seq/1 \)

and \( tail/2 \)

\[
\forall s/seq(Z) (\exists q1/seq(Z), v/Z, s1/seq(Z) \\
((s =<> \lor (\neg s =<> \land s \neq <> \land \exists h/Z s = h :: q1 \land \\
q1 = <> \land p2(s)) \lor (\neg s =<> \lor (\neg s =<> \land s \neq <> \land \\
\exists h/Z s = h :: q1 \land q1 = <>)) \land p3(s, v, s1) \land
\]

237
because of the induction base \( s = <> \rightarrow \text{true} \) and \( \neg s = <> \rightarrow \text{false} \)

\[
\forall s/\text{seq}(Z) \ (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z)
\quad ((s = <> \lor (\text{false} \land s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land
p2(s)) \lor ((\neg(s = <> \lor (\text{false} \land s \neq <> \land \exists h/Z s = h :: q1 \land
q1 = <>)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))
\]

by FOL laws \( (\text{false} \land P \rightarrow \text{false}) \)

\[
\forall s/\text{seq}(Z) \ (\exists v/Z, s1/\text{seq}(Z)
\quad (s = <> \lor \text{false} \lor (\neg(s = <> \lor \text{false}) \land
p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))
\]

by FOL laws (de Morgan’s laws)

\[
\forall s/\text{seq}(Z) \ (\exists v/Z, s1/\text{seq}(Z)
\quad (s = <> \lor \text{false} \lor ((\neg(s = <> \land \neg\text{false}) \land
p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))
\]

because of the induction base \( s = <> \rightarrow \text{true} \) and \( \neg s = <> \rightarrow \text{false} \)

by FOL laws \( (\text{false} \land P \rightarrow \text{false}) \)

\[
\forall s/\text{seq}(Z) \ (\exists v/Z, s1/\text{seq}(Z)
\quad (s = <> \lor \text{false} \lor \text{false})
\]

by FOL laws \( (\text{false} \lor P \rightarrow P) \)

\[
\forall s/\text{seq}(Z) \ (s = <>)
\]

Induction base: \( s = h :: t \land t = <> \)

\[
\forall s/\text{seq}(Z) \ (\text{incrOrd}(s)
\quad \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) (s = <> \lor (s = h :: t \land (t = <> \lor
(t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^3(t)))))
\]

by FOL laws \( (\land \text{distribution}) \)

\[
\rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) (s = <> \lor (s = h :: t \land t = <>)) \lor
(s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^3(t)))
\]

by axiom \( \text{A2.1} \)

\[
\rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) (\text{false} \lor (s = h :: t \land t = <>)) \lor
(s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^3(t)))
\]
by FOL laws \((\text{false} \lor P \rightarrow P)\)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z)((s = h :: t \land t =<> \lor t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t))) \]

because of the induction base \(t =<>\) and by axiom \(\text{A2.i}\)
\[ \rightarrow \exists h, h1/Z, t/\text{seq}(Z)((s = h :: t \land t =<> \lor s = h :: t \land \text{false} \land h \leq h1 \land \text{incrOrd}^S(t))) \]

by FOL laws \((\text{false} \land P \rightarrow \text{false})\)
\[ \rightarrow \exists h/Z, t/\text{seq}(Z)(s = h :: t \land t =<> \lor \text{false}) \]

by FOL laws \((\text{false} \lor P \rightarrow \text{false})\)
\[ \rightarrow \exists h/Z, t/\text{seq}(Z)(s = h :: t \land t =<> \lor \text{false}) \]

because of the completion of \(\text{incrOrd}/1\)
\[ \forall s/\text{seq}(Z) \ (p1(s) \land p2(s) \lor \exists v/Z, s1/\text{seq}(Z) (\neg p1(s) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v))) \quad \rightarrow \quad \text{-/-} \]

because of the completion of \(p1/1\)
\[ \forall s/\text{seq}(Z) \ ((p5(s) \lor p6(s)) \land p2(s) \lor \exists v/Z, s1/\text{seq}(Z) (\neg (p5(s) \lor p6(s)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v))) \quad \rightarrow \quad \text{-/-} \]

because of the completion of \(p5/1\) and \(p6/1\)
\[ \forall s/\text{seq}(Z) \ ((\text{empty_seq}(s) \lor \exists q1/\text{seq}(Z) (p7(s, q1) \land p8(s, q1)) \land p2(s)) \lor \exists v/Z, s1/\text{seq}(Z) (\neg (\text{empty_seq}(s) \lor \exists q1/\text{seq}(Z) (p7(s, q1) \land p8(s, q1)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v))) \quad \rightarrow \quad \text{-/-} \]

by FOL laws \((A \land \exists x P \rightarrow \exists x (A \land P)\) if \(x\) is not free in \(A\);
\(A \lor \exists x P \rightarrow \exists x (A \lor P)\) if \(x\) is not free in \(A\);
\(\exists x (P \lor Q) \rightarrow \exists x P \lor \exists x Q\)
\[ \forall s/\text{seq}(Z) \ (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z)) \]
\[ ((\text{empty_seq}(s) \lor (p7(s, q1) \land p8(s, q1) \land p2(s)) \lor \neg ((\text{empty_seq}(s) \lor (p7(s, q1) \land p8(s, q1)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v))) \quad \rightarrow \quad \text{-/-} \]

because of the specifications of the DT operation \(\text{empty_seq}/1\)
\[ \forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \]
\[ ((s =<> \lor (p7(s, q1) \land p8(s, q1) \land p2(s))) \]

239
\[ \neg(s = <> \lor (p7(s, q1) \land p8(s, q1))) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v)) \rightarrow \neg / \]

because of the completion of \( p7/2 \) and by FOL laws (\( \land \) associativity)
\[
\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \\
((s = <> \lor (p9(s, q1) \land p10(s, q1) \land \\
p2(s)) \lor (\neg(s = <> \lor (p6(s, q1) \land p10(s, q1) \land \\
p8(s, q1))) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

because of the completion of \( p8/2, p9/2 \) and \( p10/2 \)
\[
\forall s/\text{seq}(Z) (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z)) \\
((s = <> \lor (\neg empty\_seq(s) \land tail(s, q1) \land empty\_seq(q1)) \land \\
p2(s)) \lor (\neg(s = <> \lor (\neg empty\_seq(s) \land tail(s, q1) \land \\
empty\_seq(q1))) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

because of the specifications of the DT operations \( empty\_seq/1 \) and \( tail/2 \) and by axiom A2.ii
\[
\forall s/\text{seq}(Z) (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z))((s = <> \lor \\
\neg(s = <> \land s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land p2(s)) \\
\lor (\neg(s = <> \lor (\neg(s = <> \land s \neq <> \land \exists h/Z s = h :: q1 \land \\
q1 = <>)) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

by FOL laws (\( \neg x = y \rightarrow x \neq y \))
\[
\forall s/\text{seq}(Z) (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z))((s = <> \lor \\
s \neq <> \land s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land p2(s)) \\
\lor (\neg(s = <> \lor (s \neq <> \land s \neq <> \land \exists h/Z s = h :: q1 \land \\
q1 = <>)) \land p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

by FOL laws (\( P \land P \rightarrow P \))
\[
\forall s/\text{seq}(Z) (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z)) \\
((s = <> \lor (s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land p2(s)) \\
\lor (\neg(s = <> \lor (s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

because of the induction base \((s = h :: t)\) and by axiom A2.i
\[
\forall s/\text{seq}(Z) (\exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z)) \\
((false \lor (s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land p2(s)) \\
\lor (\neg(false \lor (s \neq <> \land \exists h/Z s = h :: q1 \land q1 = <>)) \land \\
p3(s, v, s1) \land incrOrd(s1) \land p4(s, v))) \rightarrow \neg / \)

240
by FOL laws \((\text{false} \lor P \leftrightarrow P) \land \text{associativity;}
\)
\[A \land \exists x P \leftrightarrow \exists x (A \land P)\] if \(x\) is not free in \(A\);
\[A \lor \exists x P \leftrightarrow \exists x (A \lor P)\] if \(x\) is not free in \(A\);
\[\exists x (P \lor Q) \leftrightarrow \exists x P \lor \exists x Q)\]

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor (\neg(s \neq <> \land s = h : q1 \land q1 = <> \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))\]

by induction base \((s = h : t \land t = <> \leftrightarrow \text{true})\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor (\neg(s \neq <> \land s = h : q1 \land q1 = <> \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))\]

by FOL laws \((P \land \text{true} \leftrightarrow P)\) and by definition of \(\neq\)
\[\text{(i.e. } s \neq <> \leftrightarrow \neg s = <>\)\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor (s = <> \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))\]

by FOL laws \((-\neg P \iff P)\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor (false \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))\]

because of the induction base and by axiom A2.i

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor (false \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)))\]

by FOL laws \((\text{false} \land P \leftrightarrow \text{false})\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land p2(s)) \lor \text{false}\]

because of the completion of \(p2/1\) and by FOL laws \((\text{false} \lor P \leftrightarrow P)\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), h/Z\]
\[\((s \neq <> \land s = h : q1 \land q1 = <> \land \text{true})\]

by axiom A2.ii, by lemma L1 and by FOL laws \((\text{true} \land P \leftrightarrow P)\)

\[\forall s/\text{seq}(Z) \exists q1/\text{seq}(Z), h/Z\]
\[\((s = h : q1 \land s = h : q1 \land q1 = <>))\]
by FOL laws ($P \land P \rightarrow P$)
\[ \forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), h/Z \left( s = h :: q1 \land q1 = <> \right) \right) \rightarrow -/- \]

Induction step: $s = h :: t \land t = h1 :: t1$

\[ \forall s/\text{seq}(Z) \left( \text{incrOrd}(s) \right) \]
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( s = <> \lor (s = h :: t \land (t = <> \lor (t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t)))) \right) \]

by FOL laws ($\land$ distribution)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( s = <> \lor (s = h :: t \land t = <> \lor (s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t)))) \right) \]

because of the induction step ($s = h :: t$) and by axiom A2.i
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( \text{false} \lor (s = h :: t \land t = <> \lor (s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t)))) \right) \]

by FOL laws ($\text{false} \lor P \rightarrow P$)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( (s = h :: t \land t = <> \lor (s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t))) \right) \]

because of the induction step $t = h1 :: t1$ and by axiom A2.i
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( \text{false} \lor (s = h :: t \land t = <> \lor (s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t))) \right) \]

by FOL laws ($\text{false} \land P \rightarrow \text{false}$)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( \text{false} \lor (s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t)) \right) \]

by FOL laws ($\text{false} \lor P \rightarrow \text{false}$)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}^S(t) \right) \]

by induction hypothesis ($t < s$)
\[ \rightarrow \exists h, h1/Z, t, t1/\text{seq}(Z) \left( s = h :: t \land t = h1 :: t1 \land h \leq h1 \land \text{incrOrd}(t) \right) \]

because of the completion of incrOrd/1
\[ \forall s/\text{seq}(Z) \left( p1(s) \land p2(s) \right) \lor \exists v/Z, s1/\text{seq}(Z) \left( \neg p1(s) \land \right) \]
because of the completion of \( p1/1 \)
\[
\forall s/\text{seq}(Z) \left( (\exists q/\text{seq}(Z) \left( p7(s,q1) \land p8(s,q1) \right) \land p2(s)) \lor (\exists v/Z, s1/\text{seq}(Z) (\neg(\exists q1/\text{seq}(Z) \left( p7(s,q1) \land p8(s,q1) \right) \land p2(s))) \right) \\
\left( p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v) \right)
\]

because of the completion of \( p5/1 \) and \( p6/1 \)
\[
\forall s/\text{seq}(Z) \left( \text{empty}_\text{seq}(s) \lor (\exists q1/\text{seq}(Z) \left( p7(s,q1) \land p8(s,q1) \right) \land p2(s)) \lor (\exists v/Z, s1/\text{seq}(Z) (\neg(\text{empty}_\text{seq}(s) \lor \exists q1/\text{seq}(Z) \left( p7(s,q1) \land p8(s,q1) \right) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v))) \right)
\]

by FOL laws \((A \land \exists x P \rightarrow \exists x (A \land P) \text{ if } x \text{ is not free in } A; \)
\((A \lor \exists x P \rightarrow \exists x (A \lor P) \text{ if } x \text{ is not free in } A; \)
\((\exists x (P \lor Q) \rightarrow \exists x P \lor \exists x Q) \)
\[
\forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \left( \text{empty}_\text{seq}(s) \lor (p7(s,q1) \land p8(s,q1) \land p2(s)) \lor (\neg(\text{empty}_\text{seq}(s) \lor (p7(s,q1) \land p8(s,q1)))) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v) \right) \right)
\]

by FOL laws \((A \land \exists x P \rightarrow \exists x (A \land P) \text{ if } x \text{ is not free in } A; \)
\((A \lor \exists x P \rightarrow \exists x (A \lor P) \text{ if } x \text{ is not free in } A; \)
\((\exists x (P \lor Q) \rightarrow \exists x P \lor \exists x Q) \)
\[
\forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \left( \text{empty}_\text{seq}(s) \lor (p7(s,q1) \land p8(s,q1) \land p2(s)) \lor (\neg(\text{empty}_\text{seq}(s) \lor (p7(s,q1) \land p8(s,q1)))) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v) \right) \right)
\]

because of the completion of \( p7/2 \) and by FOL laws \((\land \text{ associativity}) \)
\[
\forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \left( (s =<> \lor (p7(s,q1) \land p8(s,q1) \land p2(s))) \lor (\neg(s =<> \lor (p7(s,q1) \land p8(s,q1)))) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v) \right) \right)
\]

because of the completion of \( p8/2, p9/2 \) and \( p10/2 \)
\[
\forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \left( (s =<> \lor (\neg(\text{empty}_\text{seq}(s) \land \text{tail}(s,q1) \land \text{empty}_\text{seq}(q1))) \land p2(s)) \lor (\neg(s =<> \lor (\neg(\text{empty}_\text{seq}(s) \land \text{tail}(s,q1) \land \text{empty}_\text{seq}(q1)))) \land p3(s,v,s1) \land \text{incrOrd}(s1) \land p4(s,v) \right) \right)
\]

243
because of the specifications of the DT operations empty_seq/1
and tail/2 and by axiom A2.ii

\[ \forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z) \left( (s = <> \lor
\begin{align*}
& (s = <> \land s \neq <> \land \exists h/Z \ s = h :: q1 \land q1 = <> \land \exists h/Z \ s = h :: q1 \land q1 = <>)) \land p2(s) \\
& \lor (\neg(s = <> \lor (s \neq <> \land s = h :: q1 \land q1 = <>)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v))
\right) \right) \]

by definition of \( \neq \) and by FOL laws

\[ (A \land \exists x P \Rightarrow \exists x(A \land P) \text{ if } x \text{ is not free in } A; \]
\[ A \lor \exists x P \Rightarrow \exists x(A \lor P) \text{ if } x \text{ is not free in } A; \]
\[ \exists x(P \lor Q) \Rightarrow \exists x P \lor \exists x Q) \]

\[ \forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z \left( (s = <> \lor (s \neq <> \land s = h :: q1 \land q1 = <>)) \land p2(s) \right) \right) \]

because of the induction step \((s = h :: t)\) and by axiom A2.i

\[ \forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z \left( (false \lor (s \neq <> \land s = h :: q1 \land q1 = <>)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)) \right) \right) \]

by FOL laws \((false \lor P \Rightarrow P; \land \text{associativity})\)

\[ \forall s/\text{seq}(Z) \left( \exists q1/\text{seq}(Z), v/Z, s1/\text{seq}(Z), h/Z \left( (s \neq <> \land s = h :: q1 \land q1 = <> \land p2(s)) \lor (\neg(s \neq <> \land s = h :: q1 \land q1 = <>)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)) \right) \right) \]

by induction step \((s = h :: t \land t = h1 :: t1)\) and axiom A2.i

\[ \forall s/\text{seq}(Z) \left( \exists v/Z, s1/\text{seq}(Z) \left( (s \neq <> \land s = h :: q1 \land false \land p2(s)) \lor (\neg(s \neq <> \land s = h :: q1 \land false)) \land p3(s, v, s1) \land \text{incrOrd}(s1) \land p4(s, v)) \right) \right) \]

by FOL laws \((false \land P \Rightarrow false)\)

\[ \forall s/\text{seq}(Z) \left( \exists v/Z, s1/\text{seq}(Z) \left( false \lor (\neg false \land p3(s, v, s1) \land \right) \right) \]

244
\[
\text{incrOrd}(s1) \land p4(s,v))
\]

by FOL laws \((false \lor P \iff P; \neg false \iff true; true \land P \iff P)\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z) (p3(s,v,s1) \land incrOrd(s1) \land p4(s,v)))
\]

because of the completion of \(p3/3\) and \(p4/3\) and by FOL laws
\((\land \text{associativity})\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z) (p11(s,v,s1) \land p12(s,v,s1) \land p13(s,v,s1) \land incrOrd(s1) \land true))
\]

because of the completion of \(p11/3\) and \(p12/3\) and by FOL laws
\((true \land P \iff P)\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z) (\text{head}(s,v) \land \text{tail}(s,s1) \land p13(s,v,s1) \land incrOrd(s1)))
\]

because of the completion of \(p13/3\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z) (\text{head}(s,v) \land \text{tail}(s,s1) \land \exists v1/Z (p14(s,v1,v,sl) \land p15(s,v1,v,si)) \land incrOrd(s1)))
\]

because of the completion of \(p14/4\) and \(p15/4\) and by FOL laws
\((A \land \exists x P \iff \exists x (A \land P)\text{ if } x \text{ is not free in } A; \land \text{associativity})\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z), v1/Z (\text{head}(s,v) \land \text{tail}(s,s1) \land \text{head}(s1,v1) \land \text{le_int}(v,v1) \land incrOrd(s1)))
\]

because of the specifications of the DT operation \text{head}/2, \text{tail}/2 and \text{le_int}/2 and by axiom A2.ii
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z), v1/Z (s \neq << \land s = v :: s1 \land s \neq << \land s = v :: s1 \land s \neq << \land s = v :: s1 \land s \neq << \land s1 = v1 :: t1 \land v \leq v1 \land incrOrd(s1)))
\]

by FOL laws \((A \land \exists x P \iff \exists x (A \land P)\text{ if } x \text{ is not free in } A)\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z), v1/Z, t1/seq(Z) (s \neq << \land s = v :: s1 \land s \neq << \land s = v :: s1 \land s \neq << \land s1 = v1 :: t1 \land v \leq v1 \land incrOrd(s1)))
\]

by FOL laws \((\land \text{commutativity}; P \land P \iff P)\)
\[
\forall s/seq(Z) (\exists v/Z, s1/seq(Z), v1/Z, t1/seq(Z)
\]
(s ≤<> ∧ s = v :: s1 ∧ s1 ≤<> ∧ s1 = v1 :: t1 ∧ v ≤ v1 ∧ incrOrd(s1)))

by lemma L1 and by axiom A2.ii
∀s/seq(Z) (∃v/Z, s1/seq(Z), v1/Z, t1/seq(Z))
     (s = v :: s1 ∧ s = v :: s1 ∧ s1 = v1 :: t1 ∧ s1 = v1 :: t1 ∧ v ≤ v1 ∧ incrOrd(s1)))

by FOL laws (P ∧ P → P)
∀s/seq(Z) (∃v/Z, s1/seq(Z), v1/Z, t1/seq(Z))
     (s = v :: s1 ∧ s1 = v1 :: t1 ∧ v ≤ v1 ∧ incrOrd(s1)))

246
Appendix D

BNF Syntax of the Refinement Meta-Language

The refinement meta-language $L$ is described by the following BNF production rules.

$$
\text{<refinement_tree>} ::= \text{<node1>} | \text{<node2>} | \text{<node1> <node3>}
$$

$$
\text{<node1>} ::= \text{node ( refine ( <undefined_pred_name> ( <arguments> ) ,}
\hspace{1cm} <\text{design_schema}> ) , <\text{node_links}> )}.
$$

$$
\text{<node2>} ::= \text{node ( refine ( <undefined_pred_name> ( <arguments> ) ,}
\hspace{1cm} \text{dt_eq , <typed_cga> ) )}. 
$$

$$
\text{<node3>} ::= \text{node ( refine ( <undefined_pred_name> ,}
\hspace{1cm} <\text{design_schema}> , <\text{node_links}> )}.
\hspace{1cm} \text{<node3>}
\hspace{1cm} \text{node ( refine ( <undefined_pred_name> ,}
\hspace{1cm} \text{dt_eq , <typed_cga> ) )}.
\hspace{1cm} \text{<node3>}
\hspace{1cm} \text{node ( refine ( <undefined_pred_name> ,}
\hspace{1cm} \text{dt_eq , <typed_cga> ) )}.
$$

$$
\text{<node_links>} ::= [ \text{<links>} ]
$$

$$
\text{<links>} ::= <\text{undefined_pred_name}> | \hspace{1cm} ( <\text{undefined_pred_name}>, \text{unrefined} ) | <\text{undefined_pred_name}> <\text{links}> | ( <\text{undefined_pred_name}>, \text{unrefined} ) <\text{links}>
$$

$$
\text{<typed_cga>} ::= <\text{dt_op_name}> ( <\text{arguments}> ) | \hspace{1cm} <\text{dt_op_name}> ( <\text{arguments}> )
$$
not <dt_op_name> ( <arguments> ) | 
<equality> | 
not <equality>

<equality> ::= eq ( arg(<term> , <mode> , <type>) , arg(<term> , <mode> , <type>) )
<term> ::= <var_id> | <const>
<design_schema> ::= incr | div_and_conq | search | 
    subgoal_A | subgoal3_A | subgoal4_A | subgoal5_A | subgoal6_A | subgoal7_A | subgoal8_A | subgoal9_A | subgoal10_A | subgoal_B | subgoal3_B | subgoal4_B | subgoal5_B | subgoal6_B | subgoal7_B | subgoal8_B | subgoal9_B | subgoal10_B | case | case3 | case4 | case5 | case6 | case7 | case8 | case9 | case10
<arguments> ::= arg(<var_id> , <mode> , <type>) | arg(<var_id> , <mode> , <type>) , <arguments>
<mode> ::= i, d
<type> ::= a data type
<var_id> ::= var(<id>)
<undefined_pred_name> ::= <pred_name_prefix> <id>
<pred_name_prefix> ::= p
<dt_op_name> ::= <small_letter> <rest_symbols>
<rest_symbols> ::= <small_letter> | <capital_letter> | _ | 
    <digit> | <small_letter> <rest_symbols> | 
    <capital_letter> <rest_symbols> | 
    <digit> <rest_symbols> | _ <rest_symbols>

<id> ::= a natural number
<const> ::= a constant
Appendix E

Sample Sessions of SIDS

This appendix illustrates two sample sessions of SIDS. The first sample session is intended to illustrate the following.

1. The use of the informal meaning of atoms schemata in order to justify the design decisions. That is, informal guidance is used for the selection of particular schema refinements and DT refinements.

2. The criteria which are used for matching the arguments of a DT operation or equality with the ones of the predicate it refines.

3. The type and mode checking that is performed by the system.

4. The support that is given to the argument matching process by the modes and types of the matching arguments. That is, certain argument pairs can not be selected for matching because either the types of these arguments are not unifiable or their modes are inconsistent.

The second sample session is intended to illustrate the following.

1. A more complex session of refinements.

2. The detection of non-ground arguments, i.e. unused arguments, in negative literals by the mode analysis procedure.

Note that the argument matching of each DT operation in the first example is followed by a discussion. Let \( p(x_1, \ldots, x_n) \) be the undefined predicate in the leaf of the refinement.
tree which is refined. Let $\Psi_P$ be the global predicate substitution of this refinement tree. If $P/p \in \Psi_P$ then the atom schema $P(u_1, \ldots, u_n)$ is the one which is mentioned in the discussion. Let $q(y_1, \ldots, y_m)$ be a typed cga(s) which exists in a node in the path of the refinement tree from its root up to the leaf which is refined. The atom schema which corresponds to this typed cga is $Q(u_1, \ldots, u_m)$ where $q/Q \in \Psi_P$. A such atom schema is also involved in the next discussion.

### E.1 Summation of Elements of a Sequence.

The predicate $\text{sum}(x_1, x_2)$ is true iff $x_2$ is the sum of the integers in sequence $x_1$. The input into SIDS is a complete refinement tree for the predicate $\text{sum}/2$.

**Input into SIDS:**

**Schema refinements**

- node(refine($\text{sum}(\text{arg(var(11),i,seq(tvar(1)))}, \text{arg(var(12),d,tvar(1)))}$), incr), [p0, p1, p2, p3]).
- node(refine(p2, subgoal_B)), [p4, p5]).

**DT refinements**

- node(refine(p0, dt_eq, empty_seq($\text{arg(var(1),d,seq(tvar(1))))}$))).
- node(refine(p1, dt_eq, neutral_add_subtr_int($\text{arg(var(1),d,int})$))).
- node(refine(p4, dt_eq, head($\text{arg(var(1),i,seq(tvar(1)))}, \text{arg(var(2),d,tvar(1)))}$))).
- node(refine(p5, dt_eq, tail($\text{arg(var(1),i,seq(tvar(1)))}, \text{arg(var(2),d,seq(tvar(1)))})$))).
- node(refine(p3, dt_eq, plus_int($\text{arg(var(1),i,int)}$, $\text{arg(var(2),i,int)}, \text{arg(var(3),d,int})$))).

The construction process is initiated by running the following goal.

?- construct_program([refine($\text{sum}(\text{arg(var(11),i,seq(tvar(1)))}, \text{arg(var(12),d,tvar(1)))}$), incr])

$\text{sum}(\text{arg(var(11),i,seq(tvar(1)))}, \text{arg(var(12),d,tvar(1)))}$ is the ground representation of the typed cga $\text{sum}(\text{var(11)}, \text{var(12)})$ of the top-level undefined predicate including the type of its terms in the atom and the mode of its arguments. The types of $\text{var(11)}$
and \( \text{var}(12) \) in atom \( \text{sum}(\text{var}(11), \text{var}(12)) \) are \( \text{seq}(\text{tvar}(1)) \) and \( \text{tvar}(1) \) respectively. The expected modes of these arguments are \( i \) and \( d \) respectively.

**Argument matching:**

Give the pairs of matching arguments \((\text{Pred}\_\text{arg}, \text{DT}\_\text{op}\_\text{arg})\)

\[
\text{Type } p0: \text{seq(a2)} \\
\text{Type empty_seq: seq(a2)}
\]

Mode \( p0: i \)  
Mode \( \text{empty_seq}: d \)

\( p0(X2) <-- \text{empty_seq}(X3) \)

Give pair 1:

\( I: 1 \)

If you want to repeat the argument matching type \( y \) else \( n \)

\( I: n \)

Note that the informal meaning of the atom schema \( \text{Terminating}(u_1) \) of Incremental schema suggests the consideration of the primitive cases of the problem represented by \( X2 \). The primitive cases of \( X2 \) for this problem is when the sequence \( X2 \) is empty. That is, the DT operation \( \text{empty_seq}(X3) \) is appropriate for refining the predicate \( p0(X2) \). The matching argument pair is \((X2, X3)\). In this case, there is no any other alternative.

Give the pairs of matching arguments \((\text{Pred}\_\text{arg}, \text{DT}\_\text{op}\_\text{arg})\)

\[
\text{Type } p1: \text{seq(a2)} \times a2 \\
\text{Type neutral_add_subtr_int: int}
\]

Mode \( p1: i,d \)  
Mode \( \text{neutral_add_subtr_int}: d \)

\( p1(X2,X4) <-- \text{neutral_add_subtr_int}(X3) \)
Give pair 1:
|: 1.
|: 1.

If you want to repeat the argument matching type y else n
|: n.

Note that the informal meaning of the atom schema $Initial.result(u_1, u_2)$ of Incremental schema suggests the construction in argument $X_4$ of the initial result for the primitive cases of $X_2$. $X_4$ should be the neutral element of addition. The DT operation $neutral.add_subtr.int(X_3)$ is appropriate for refining the predicate $p_1(X_2, X_4)$. That is, the argument pair $(X_4, X_3)$ should be matched. The type of the arguments $X_2$ and $X_3$ excludes their matching.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type $p_4$: seq(int) X a3 X seq(int)
Type head: seq(a2) X a2

Mode $p_4$: i,d,d
Mode head: i,d

$p_4(X_2, X_5, X_6) \leftarrow head(X_3, X_7)$

Give pair 1:
|: 1.
|: 1.

Give pair 2:
|: 3.
|: 2.

If you want to repeat the argument matching type y else n
|: n.

*** ERROR: The types of selected arguments do not match

NOTE: Repeat the argument matching
Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p4: seq(int) X a3 X seq(int)
Type head: seq(a2) X a2

Mode p4: i,d,d
Mode head: i,d

p4(X2,X5,X6) <-- head(X3,X7)

Give pair 1:
|: 1.
|: 1.

Give pair 2:
|: 2.
|: 2.

If you want to repeat the argument matching type y else n
|: n.

Initially, the argument pairs (X2,X3) and (X6,X7) are matched. A type error occurs and the argument matching process is repeated.

Note that the informal meaning of the atom schema Deconstruction(u_1,v_1,v_2) of Incremental schema suggests the decomposition of X2 into an element represented by X5 and the remaining part represented by X6. The informal meaning of the atom schema SubGoal1(u,v,w_2) suggests the construction of the solution of the first subproblem which is represented by the argument X5. The DT operation head(X3,X7) is appropriate for refining the predicate p4(X2,X5,X6). The argument pairs which should be matched are (X2,X3) and (X5,X7).

The tuple of types of the pairs \{(X2,X3),(X6,X7)\} are not unifiable. That is, mgu((seq(int), seq(int)), (seq(a2), a2)) does not exist. Any set of pairs of matching arguments which includes the pair (X6,X3) can not be selected because the mode of the argument X6 is not subsumed by the mode of the argument X3. A similar comment applies for the pair (X5,X3). The sets of pairs of arguments which have to be excluded for this reason are
Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p5: seq(int) x int x seq(int)
Type tail: seq(a2) x seq(a2)

Mode p5: i,i,d
Mode tail: i,d

p5(X2,X5,X6) <-- tail(X3,X7)

Give pair 1:
|: 3.
|: 1.

Give pair 2:
|: 1.
|: 2.

If you want to repeat the argument matching type y else n
|: n.

*** ERROR: The call mode of DT operation(s) is/are not subsumed by
its/their declared mode

<table>
<thead>
<tr>
<th></th>
<th>Argument positions with invalid call patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTE: Repeat the argument matching

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p5: seq(int) x int x seq(int)
Type tail: seq(a2) x seq(a2)

Mode p5: i,i,d
Mode tail: i,d

\[ p5(X2,X5,X6) \leftarrow \text{tail}(X3,X7) \]

Give pair 1:
| l: 1 |
| l: 1 |

Give pair 2:
| l: 3 |
| l: 2 |

If you want to repeat the argument matching type \( y \) else \( n \)
| l: n |

Initially, the argument pairs \((X6,X3)\) and \((X2,X7)\) are matched. A mode error occurs and the argument matching process is repeated.

Note that the informal meaning of the atom schema \( \text{Deconstruction}(u_1,v_1,v_2) \) of Incremental schema suggests the decomposition of \( X2 \) into an element represented by \( X5 \) and the remaining part represented by \( X6 \). The informal meaning of the atom schema \( \text{SubGoal2}(u_1,v,v_2) \) suggests the construction of the solution of the second subproblem which is represented by the argument \( X6 \). The DT operation \( \text{tail}(X3,X7) \) is appropriate for refining the predicate \( p5(X2,X5,X6) \). The argument pairs which should be matched are \((X2,X3)\) and \((X6,X7)\).

The tuple of types of the pairs \( \{(X6,X3),(X5,X7)\} \) are not unifiable. Any set of pairs of matching arguments which includes the pair \((X6,X3)\) can not be selected because the mode of the argument \( X6 \) is not subsumed by the mode of the argument \( X3 \). The sets of pairs which are excluded for this reason are \( \{(X6,X3),(X2,X7)\} \) and \( \{(X6,X3),(X5,X7)\} \).

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type \( p3: \text{seq(int)} X \text{int} X \text{int} X \text{int} \)
Type \( \text{plus_int: int} X \text{int} X \text{int} \)

Mode \( p3: i,i,i,d \)
Mode \( \text{plus_int: i,i,d} \)
\[
p3(X2,X5,X8,X4) \leftarrow \text{plus_int}(X3,X7,X9)
\]

Give pair 1:
| 1: 2.  |
| 1: 1.  |

Give pair 2:
| 1: 3.  |
| 1: 2.  |

Give pair 3:
| 1: 4.  |
| 1: 3.  |

If you want to repeat the argument matching type y else \( n \)
| 1: \( n \). |

Note that the informal meaning of the atom schema \( \text{Non\_initial\_result}(u_1, v_1, v_3, u_2) \) of Incremental schema suggests the construction of the solution for the general cases of the problem. The appropriate DT operation is \( \text{plus\_int}/3 \). In addition, it suggests the construction of the solution into the argument \( X4 \) by using the other arguments of \( p3/4 \). The argument \( X8 \) represents the result from processing the remaining part of \( X2 \). \( X5 \) has the component from the deconstruction of \( X2 \). \( X8 \) together with \( X5 \) will be used to construct the non-initial result into \( X4 \). That is, the arguments pairs which should be matched are \( (X5,X3), (X8,X7) \) and \( (X4,X9) \).

The type of the argument \( X2 \) suggests us that \( X2 \) can not be matched with any argument of the DT operation \( \text{plus\_int}/3 \). In addition, the mode of the argument \( X4 \) suggests us that \( X4 \) can not be matched neither with the first not with the second argument of \( \text{plus\_int}/3 \).

The output of SIDS is the SI-program which is presented in Section 3.7.

### E.2 Insertion Sort.

The input into SIDS is a complete refinement plan for insertion sort.

*Input into SIDS:*
Schema refinements

node(refine(insSort(arg(var(11),i,seq(tvar(1)))),
arg(var(12),d,seq(tvar(2)))), incr), [p0,p1,p2,p3]).
node(refine(p2,subgoal_B), [p4,p5]).
node(refine(p3,incr), [p6,p7,p8,p9]).
node(refine(p6,case), [p10,p11]).
node(refine(p8,subgoal3_B), [p14,p15,p16]).
node(refine(p11,subgoal_A), [p12,p13]).

DT refinements

node(refine(p0,dt_eq, empty_seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p1,dt_eq, empty_seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p4,dt_eq, head(arg(var(1),i,seq(tvar(1)))),arg(var(2),d,tvar(1))))).
node(refine(p5,dt_eq, tail(arg(var(1),i,seq(tvar(1)))),arg(var(2),d,seq(tvar(1))))).
node(refine(p7,dt_eq, seq_cons(arg(var(1),i,seq(tvar(1))),'
arg(var(2),i,tvar(1)),arg(var(3),d,seq(tvar(1)))))).
node(refine(p9,dt_eq, seq_cons(arg(var(1),i,seq(tvar(1))),'
arg(var(2),i,tvar(1)),arg(var(3),d,seq(tvar(1)))))).
node(refine(p10,dt_eq, empty_seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p12,dt_eq, head(arg(var(1),i,seq(tvar(1)))),arg(var(2),d,tvar(1))))).
node(refine(p13,dt_eq, le_int(arg(var(1),i,int),arg(var(2),i,int)))).
node(refine(p16,dt_eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p14,dt_eq, head(arg(var(1),i,seq(tvar(1)))),arg(var(2),d,tvar(1))))).
node(refine(p15,dt_eq, tail(arg(var(1),i,seq(tvar(1)))),arg(var(2),d,seq(tvar(1))))).

The construction process is initiated by running the following goal.

?- construct_program([refine(insSort(arg(var(11),i,seq(tvar(1)))),
arg(var(12),d,seq(tvar(2)))), incr]).

insSort(arg(var(11),i,seq(tvar(1))), arg(var(12),d,seq(tvar(2)))) is the ground representation of the typed cga insSort(var(11),var(12)) of the top-level undefined predicate that the user wants to construct a program including the type of its terms in the atom and the mode of its arguments. The types of its terms var(11) and var(12) in atom
insSort(var(11),var(12)) are seq(tvar(1)) and seq(tvar(2)) respectively. The expected modes of these arguments are i and d respectively.

Argument matching:

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p0: seq(a2)
Type empty_seq: seq(a2)

Mode p0: i
Mode empty_seq: d

p0(X2) <-- empty_seq(X3)

Give pair 1:
|: 1.
|: 1.

If you want to repeat the argument matching type y else n
|: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p1: seq(a2) X seq(a3)
Type empty_seq: seq(a2)

Mode p1: i,d
Mode empty_seq: d

p1(X2,X4) <-- empty_seq(X3)

Give pair 1:
|: 2.
|: 1.

If you want to repeat the argument matching type y else n

258
Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p4: seq(a2) X a4 X seq(a2)
Type head: seq(a2) X a2

Mode p4: i,d,d
Mode head: i,d

p4(X2,X5,X6) <-- head(X3,X7)

Give pair 1:
  |: 1.
  |: 1.

Give pair 2:
  |: 2.
  |: 2.

If you want to repeat the argument matching type y else n
  |: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p5: seq(a4) X a4 X seq(a4)
Type tail: seq(a2) X seq(a2)

Mode p5: i,i,d
Mode tail: i,d

p5(X2,X5,X6) <-- tail(X3,X7)

Give pair 1:
  |: 1.
  |: 1.
Give pair 2:
|: 3.
|: 2.

If you want to repeat the argument matching type y else n
|: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type pl0: seq(a4) X a4 X seq(a3)
Type empty_seq: seq(a2)

Mode pl0: i,i,i
Mode empty_seq: d

pl0(X2,X5,X8) <-- empty_seq(X3)

Give pair 1:
|: 2.
|: 2.

*** ERROR: Invalid argument

Give pair 1:
|: 3.
|: 1.

If you want to repeat the argument matching type y else n
|: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p12: seq(a4) X a4 X seq(a3) X a5
Type head: seq(a2) X a2

260
Mode pl2: i,i,i,d
Mode head: i,d

pl2(X2,X5,X8,X9) ← head(X3,X7)

Give pair 1:
| i: 3.
| i: 1.

Give pair 2:
| i: 4.
| i: 2.

If you want to repeat the argument matching type y else n
| i: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type pl3: seq(a4) X a4 X seq(a5) X a5
Type le_int: int X int

Mode pl3: i,i,i,i
Mode le_int: i,i

pl3(X2,X5,X8,X9) ← le_int(X3,X7)

Give pair 1:
| i: 2.
| i: 1.

Give pair 2:
| i: 4.
| i: 2.

If you want to repeat the argument matching type y else n
| i: n.
Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p7: seq(int) X int X seq(int) X seq(int)
Type seq_cons: seq(a2) X a2 X seq(a2)

Mode p7: i,i,i,d
Mode seq_cons: i,i,d

p7(X2,X5,X8,X4) <-- seq_cons(X3,X7,X10)

Give pair 1:
| 2.
| 2.

Give pair 2:
| 3.
| 1.

Give pair 3:
| 4.
| 3.

If you want to repeat the argument matching type y else n
| n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p14: seq(int) X int X seq(int) X a6 X seq(int) X int X seq(int)
Type head: seq(a2) X a2

Mode p14: i,i,i,d,d,d,d
Mode head: i,d

p14(X2,X5,X8,X11,X12,X13,X14) <-- head(X3,X7)

Give pair 1:
| 3.
Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p15: seq(int) X int X seq(int) X int X seq(int) X int X seq(int)
Type tail: seq(a2) X seq(a2)

Mode p15: i,i,i,i,d,d,d
Mode tail: i,d

pl5(X2,X5,X8,X11,X12,X13,X14) <-- tail(X3,X7)

Give pair 1:
|: 3.
|: 1.

Give pair 2:
|: 7.
|: 2.

If you want to repeat the argument matching type y else n
|: n.

Give the pairs of matching arguments (Pred_arg, DT_op_arg)

Type p16: seq(int) X int X seq(int) X int X seq(int) X int X seq(int)
Type eq: a2 X a2

Mode p16: i,i,i,i,d,d,i
Mode $\text{eq: } d,d$

$p_{16}(x_2,x_5,x_8,x_{11},x_{12},x_{13},x_{14}) \leftarrow \text{eq}(x_3,x_7)$

Give pair 1:
I: 2.
I: 1.

Give pair 2:
I: 6.
I: 2.

If you want to repeat the argument matching type $y$ else $n$
I: n.

*** Warning in negative literal:
The argument(s) in position(s) 1 of the predicate $p_6$
which appears in the clauses defining the predicate $p_3$
is/are neither ground nor universally quantified

Give the pairs of matching arguments (Pred_arg, DT_op_arg)
=====================================================================

Type $p_9$: seq(int) X int X seq(int) X int X seq(int) X seq(int)
Type seq_cons: seq(a2) X a2 X seq(a2)

Mode $p_9$: d,i,i,i,i,d
Mode seq_cons: i,i,d

$p_9(x_2,x_5,x_8,x_{11},x_{15},x_4) \leftarrow \text{seq_cons}(x_3,x_7,x_{10})$

Give pair 1:
|: 4.
|: 2.

Give pair 2:
|: 5.
|: 1.
Give pair 3:
1: 6.
1: 3.

If you want to repeat the argument matching type y else n
1: n.

*** Warning in negative literal:  
The argument(s) in positions 1 of the predicate p6
which appears in the clauses defining the predicate p3
is/are neither ground nor universally quantified

The output of SIDS is the SI-program which is presented in Section 3.10.
Appendix F

Refinement Trees

This appendix contains refinement trees for trivial programs which are shown by the other schema-based methods surveyed in 1.3. These simple refinement trees illustrate the argument of this thesis discussed in 7.1.4 that program reasoning in this method should be performed at the level of refinement trees.

Example 1: subset(x1,x2) is true iff the set x1 is subset of the set x2.

Refinement tree.

Schema refinements

node(refine(subset(arg(var(11)),i,set(tvar(11)))),
    arg(var(12),i,set(tvar(11))), incr), [p0,p1,p2,p3]).
node(refine(p2, subgoal4.A), [p4,p5,p6,p7]).

DT refinements

node(refine(p0, dt_eq, empty_set(arg(var(1),d,set(tvar(1)))))),
node(refine(p1, dt_eq, true)).
node(refine(p3, dt_eq, true)).
node(refine(p4, dt_eq,
    member_set(arg(var(1),d,tvar(1)),arg(var(2),i,set(tvar(1)))))).
node(refine(p5, dt_eq,
    member_set(arg(var(1),d,tvar(1)),arg(var(2),i,set(tvar(1)))))).
node(refine(p6, dt_eq,
    rest_set(arg(var(1),i,tvar(1)),arg(var(2),i,set(tvar(1)))),
    arg(var(3),d,set(tvar(1))))).
Example 2: prefix(x1,x2) is true iff the sequence x1 is a prefix of sequence x2.

Refinement tree.
Schema refinements
node(refine(prefixl(arg(var(l),i,seq(tvar(l)))) , arg(var(12),i,seq(tvar(11))))) , incr), [p0,p1,p2,p3]).
node(refine(p2, subgoal4_A), [p4,p5,p6,p7]).

DT refinements
node(refine(p0, dt.eq, empty.seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p1, dt.eq, true)).
node(refine(p3, dt.eq, true)).
node(refine(p4, dt.eq,
  head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p5, dt.eq,
  head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p6, dt.eq,
  tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1)))))).
node(refine(p7, dt.eq,
  tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1)))))).

Example 3: lengthAcc(x1,x2) is true iff x2 is the length of sequence x1.

Refinement tree.
Schema refinements
node(refine(lengthAcc(arg(var(11),i,seq(int)),arg(var(12),d,nat)),
  subgoal_A), [p0,p1]).
node(refine(p1, incr), [p2,p3,p4,p5]).
node(refine(p4, subgoal3_A), [p6,p7,p8]).

DT refinements
node(refine(p0, dt.eq, neutral.add.subtr.nat(arg(var(1),d, nat)))).
node(refine(p2, dt.eq, empty.seq(arg(var(1),d,seq(tvar(1)))))).
node(refine(p3, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p5, dt.eq, eq(arg(var(1),d,tvar(1)), arg(var(2),d,tvar(1))))).
Example 4: sumAcc(x1,x2) is true iff x2 is the sum of the integers in sequence x1.

Refinement tree.
Schema refinements

node(refine(sumAcc(arg(var(11),i,seq(int)),arg(var(12),d,int)), 
subgoal_A), [p0,p1]).
node(refine(p1, incr), [p2,p3,p4,p5]).
node(refine(p4, subgoal3_B), [p6,p7,p8]).

DT refinements

node(refine(p0, dt.eq, neutral_add_subtr_int(arg(var(1),d, int))))).
node(refine(p2, dt.eq, empty_seq(arg(var(1),d,seq(tvar(1))))).
node(refine(p3, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).
node(refine(p5, dt.eq, 
plus.int(arg(var(1),i,int),arg(var(2),i,int), arg(var(3),d,int)))).
node(refine(p6, dt.eq, 
head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p7, dt.eq, 
tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1))))).
node(refine(p8, dt.eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).

Example 5: append(x1,x2,x3) is true iff sequence x3 is the concatenation of sequences x1 and x2.

Refinement tree.
Schema refinements

node(refine(append(arg(var(11),i,seq(tvar(11))), 
arg(var(12),i,seq(tvar(12))),arg(var(13),d,seq(tvar(13)))), 
incr), [p0,p1,p2,p3]).
node(refine(p2, subgoal3_B), [p4,p5,p6]).

DT refinements

node(refine(p0, dt.eq, empty_seq(arg(var(1),d,seq(tvar(1))))).
Example 6: efface(x1,x2,x3) is true iff the sequence x3 is the sequence x2 without the first occurrence of the element x1.

Refinement tree.
Schema refinements
node(refine(efface(arg(var(11),i,tvar(11)), arg(var(12),i,seq(tvar(11)))), incr), [p0.pl,p2,p3]).
node(refine(p2, subgoal3.B), [p4,p5,p6]).

DT refinements
node(refine(p0, dt_eq, head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p1, dt_eq, tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1))))).
node(refine(p3, dt_eq, seq_cons(arg(var(1),i,seq(tvar(1))),arg(var(2),i,tvar(1)),
arg(var(3),d,seq(tvar(1))))).
node(refine(p4, dt_eq, head(arg(var(1),i,seq(tvar(1))),arg(var(2),d,tvar(1))))).
node(refine(p5, dt_eq, tail(arg(var(1),i,seq(tvar(1))),arg(var(2),d,seq(tvar(1))))).
node(refine(p6, dt_eq, eq(arg(var(1),d,tvar(1)),arg(var(2),d,tvar(1))))).

Example 7: connected(x1,x2) is true iff the vertices are connected in the graph represented by the relation edge/2.

Refinement tree.
Schema refinements
node(refine(connected(arg(var(11),i,tvar(11)), arg(var(12),i,tvar(11)))),
Example 8: path(x1,x2,x5) is true iff x5 is a sequence with the path from vertex x1 to vertex x2.

Refinement tree.

Schema refinements

DT refinements

270.