Representation of Compositional Relational Programs

Representation of Compositional Relational Programs

Görkem Paçacı

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Lecture hall 2, Ekonomikum, Uppsala, Friday, 28 April 2017 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Alberto Pettorossi (University of Rome Tor Vergata).

Abstract

Paçacı, G. 2017. Representation of Compositional Relational Programs. 216 pp. Uppsala: Department of Informatics and Media, Uppsala University. ISBN 978-91-506-2621-6.

Usability aspects of programming languages are often overlooked, yet have a substantial effect on programmer productivity. These issues are even more acute in the field of Inductive Synthesis, where programs are automatically generated from sample expected input and output data, and the programmer needs to be able to comprehend, and confirm or reject the suggested programs.

A promising method of Inductive Synthesis, CombInduce, which is particularly suitable for synthesizing recursive programs, is a candidate for improvements in usability as the target language Combilog is not user-friendly. The method requires the target language to be strictly compositional, hence devoid of variables, yet have the expressiveness of definite clause programs. This sets up a challenging problem for establishing a user-friendly but equally expressive target language.

Alternatives to Combilog, such as Quine's Predicate-functor Logic and Schönfinkel and Curry's Combinatory Logic also do not offer a practical notation: finding a more usable representation is imperative. This thesis presents two distinct approaches towards more convenient representations which still maintain compositionality.

The first is Visual Combilog (VC), a system for visualizing Combilog programs. In this approach Combilog remains as the target language for synthesis, but programs can be read and modified by interacting with the equivalent diagrams instead. VC is implemented as a split-view editor that maintains the equivalent Combilog and VC representations on-the-fly, automatically transforming them as necessary.

The second approach is Combilog with Name Projection (CNP), a textual iteration of Combilog that replaces numeric argument positions with argument names. The result is a language where argument names make the notation more readable, yet compositionality is preserved by avoiding variables. Compositionality is demonstrated by implementing CombInduce with CNP as the target language, revealing that programs with the same level of recursive complexity can be synthesized in CNP equally well, and establishing the underlying method of synthesis can also work with CNP.

Our evaluations of the user-friendliness of both representations are supported by a range of methods from Information Visualization, Cognitive Modelling, and Human-Computer Interaction. The increased usability of both representations are confirmed by empirical user studies: an often neglected aspect of language design.

Keywords: Programming, Syntax, Logic Programming, Combilog, CombInduce, Prolog, Variable-free, Point-free, Tacit, Compositional Relational Programming, Combinatory Logic, Predicate-Functor Logic, Program Synthesis, Meta-interpreters, Meta-interpretative Synthesis, Decompositional Synthesis, Inductive Synthesis, Inductive Logic Programming, Usability, Cognitive Dimensions of Notations, Visual Variables, Usability testing, Programming Language usability, Empirical evidence

Görkem Paçacı, Department of Informatics and Media, Information Systems, Kyrkogårdsg. 10, Uppsala University, SE-751 20 Uppsala, Sweden.

ISBN 978-91-506-2621-6
urn:nbn:se:uu:diva-317084 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-317084)

Acknowledgements

This journey was made possible and enjoyable thanks to the support and fellowship of these lovely people. I shall express my gratitude and appreciation,

To my invaluable advisor Prof. Dr. Andreas Hamfelt. I would not have been able to chart my way through and concluded this work, if it was not for your continuous and unlimited support. It is impossible to overstate how grateful I am to have you as a friend, guide, and family. I hope I didn't consume all your patience and you still have some left for future students. It was a privilege and honour to learn from you, and I hope to do so for many years to come.

To Dr. Steve McKeever for tirelessly reading, commenting, and contributing with his ideas to this work. I shall miss your witty remarks, and your comments written with your intimidating bright red pen.

To Prof. Dr. Jørgen Fischer Nilsson for his contributions and for his insightful comments in various stages of this work.

To my lovely, caring wife Neşe, for putting up with me all the times my mind was away, and filling my life with cheer, beauty and delight. I will finally have some time to enjoy you.

To my dear mother Nilüfer for letting me go far and always supporting me. To my dear father Levent for always encouraging me for more. To Süheyla for her support, and giving me two lovely sisters. To my dear sister Aylin for always intriguing me. To my dear sister Deniz for always entertaining me, and for designing this brilliant cover (with inspiration from Felix Prax). To my gleeful aunts Nükhet and Funda, and to my lively cousins Hande, Betül, Mertcan, Buse and Nehir.

To my dear friends Serkan Barut, Dr. Onur Yılmaz, Dr. Selva Baziki, and Dr. Ali Sina Önder for all the fun times and reassuring me I'll get to the finish line.

To all past and current Ph.D. students in Informatics and Media, for making this place as cosy and crazy as it is. Thank you for the camaraderie Siddharth Chadha, Kirill Filimonov, Dr. Sylvain Firer-Blaess, Dr. Mareike Glöss, Cristina Ghita, Dr. Anders Larsson, Ruth Lochan, Daniel Lövgren, Dr. Elena Márquez, Mudassir Imran Mustafa, Dr. Patrick Prax, Hafijur Mohammad Rahman, Asma Rafiq, Paulina Rajkowska, Martin Stojanov, Dr. Emma Svensson, Christopher Okhravi, Laia Turmo Vidal, Dr. Stanislaw Zabramski.

To all the nice people in IM, for making my years here pleasant and cheerful, and making IM a cool and warm place to work at, and never

passing an opportunity to party. Thank you Anton Backe, Mikael Berg, Carina Boson, Ylva Ekström, Dr. Jenny Eriksson Lundström, Prof. Dr. Mats Edenius, Dr. Anneli Edman, Eva Enefjord, Madelen Hermelin, Martin Landahl, Prof. Dr. Mats Lind, Eva Karlsson, Tina Kekkonen, Jakob Malmstedt, Dr. Therese Monstad, Mattias Nordlindh, Dr. Else Nygren, Mathias Peters, Sofie Roos, Klara Runesson, Christian Sandström, Sophie Skogehall, Christer Stuxberg, Lars-Göran Svensk, Dr. Göran Svensson, Dr. Claes Thorén, Dr. Franck Tétard, Prof. Dr. Sten-Åke Tärnlund, Prof. Dr. Annika Waern, GunBritt Wass, Prof. Dr. Vladislav Valkovsky, Prof. Dr. Pär Ågerfalk.

To Dr. David Johnston, for reading multiple versions of this thesis without getting tired of it.

To my professors and colleagues in Izmir, especially to Prof. Dr. Brahim Hnich, Prof. Dr. Cemali Dinçer and Dr. Mahmut Ali Gökçe. To my dear friends Dr. Umut Avcı and Dr. Kaya Oğuz, with whom we started a journey that unpredictably lead me to Uppsala.

To Dr. Özgür Akgün and Dr. Migual Nacenta for their support and hospitality in St Andrews.

To everyone who contributed their valuable time to participate in the user studies in this thesis.

To the Smålands nation in Uppsala, and the Anna Maria Lundin's Fund for generously supporting this work with multiple grants throughout the years.

Contents

1. Introduction

"...
I like to think
(right now, please!)
of a cybernetic forest
filled with pines and electronics
where deer stroll peacefully
past computers
as if they were flowers
with spinning blossoms.
...
Richard Brautigan "

Composing programs for computers to execute is still a mostly mundane task. It is so despite continual efforts to improve programming languages and building associated sophisticated abstractions. In any program of non-trivial length, it is likely that a significant number of lines are of a repetitive nature. We trust algorithms to drive vehicles and operate the safety systems of nuclear reactors; surely we can trust them to write other algorithms, at least partially? Automatic programming has been around almost since people started writing programs, albeit early usages were often the most routine. Generating code for database schemas or targetting multiple platforms are some examples. There are methods of automatic programming capable of producing more *creative* results, incorporating a level of *intellect* in a generated program. This thesis addresses making such certain techniques more accessible by improving their usability.

One of the methods of automatic programming is *deductive synthesis*, which takes a *specification* of the program to be written, and outputs a program that is guaranteed to follow the given specification. As efficient as this method is, there are some fundamental shortcomings. The specification language has to be as expressive as the target language, and often the specification itself is as long as the program it produces [57]. Deductive synthesis guarantees the generated program will follow the specification, but logical errors within it are still a concern. Ease of use

issues are related, in turn, to the usability of the specification language, as the user is required to devise what is often a demanding specification.

Another method is *inductive synthesis*, which takes input/output examples (called observables) of how the program should work. As a result it induces a program that does produce the given outputs for the given inputs. A shortcoming of this method is that the fallacy of specification still applies. It is guaranteed that the generated program will entail the given observables, but the appropriateness of the generated program is dependent on the specificity of the given observables. Inductive synthesis is inherently more applicable where the observables are easy to obtain, such as in *test-driven programming*, where practice dictates program observables are documented before any attempt at writing the program itself. Engaging in inductive synthesis techniques at this stage may usefully reduce the amount of hand-written code.

The usability concerns regarding inductive synthesis are different to those in deductive synthesis. In deductive synthesis the specification language's ease of use matters, because the user is expected to write the specification by hand. With inductive synthesis, in contrast, the observables (which stand as a form of specification) are trivially easy to devise, but effort is required to confirm the generated program as an appropriate one. For example, let us assume the task is to synthesize a function $f(x, y)$ with two observables, $f(1, 1) = 1$ and $f(2, 2) = 4$. The implementation $f(x, y) = x * y$ does satisfy the observables, but so does $f(x, y) = x^y$. Ultimately it is up to the user to evaluate the appropriateness of the suggested program. On the other side of the coin, deductive synthesis guarantees the generated program to be correct, as long as the given specification is correct, so the user is more involved with the specification language. In inductive synthesis, providing a complete set of observables for a program is rarely feasible, and therefore the aim is to use as few observables as possible. In consequence, the usability of the target language is more of a concern in inductive synthesis, in order that the user can more easily read, comprehend, and if the need arises, modify the generated program to its final appropriate form. This thesis is primarily concerned with improving the usability of a particular target language, COMBILOG, which was devised for a particular method of inductive synthesis, namely *meta-interpretative synthesis*, and specifically for inducing *definite clause programs*.

Meta-interpretative synthesis essentially involves invoking the meta-interpreter of an object language in the reverse direction. Instead of giving a program in the object language and obtaining its consequences, the consequences are given and the program is obtained. This is made possible by writing the meta-interpreter as a bi-directional provability relation, which is a technique that has been developed since the 1990's, including

the works of Sato [102], Hamfelt and Nilsson [45, 48, 49], Christiansen [22, 23], Basin et al [11], and Muggleton [74].

Hamfelt and Nilsson's work in CombInduce is distinct in the way they incorporate list recursion operators as language constructs. The object language for CombInduce, namely combilog, was devised to be strictly *compositional*, and together with the recursion operators it presents a unique approach to synthesize recursive programs. When the meta-interpreter for combilog invoked in the reverse direction and provided with input/output examples, it also implicitly hypothesizes the program may be a recursive one, in which case it *decomposes* observables into new observables corresponding to the components of said recursion operators. As a result, CombInduce can synthesize programs up to two levels of list recursion, for example *naive reverse*, which iterates through a list backwards and keeps appending each element to the end of a running list. The appending operation is the first level of recursion, and iterating through the list is the second. This efficient technique for synthesis of recursive programs is possible due to the compositionality of the object language. The lack of a context reduces the complexity, allowing the reversible meta-interpreter to decompose observables into observables of sub-programs without surrendering to a combinatorial explosion.

In the following section, let us describe the principle of compositionality, to enable the subsequent exposition of the core concerns of this thesis.

1.1 Compositionality

Compositional programming languages allow new programs to be defined in terms of subprograms using composition operators. Compositionality exists in almost every high level programming language to some degree. For example, in the C programming language, it is possible to write sub-programs and invoke these sub-programs within other programs, effectively defining new programs in terms of sub-programs. The form of compositionality that concerns us is a stronger one, where the language allows *only* compositional expressions. These are the languages where the meaning of a compound expression is defined by solely the meanings of component expressions, and the composition operator.

To achieve this, it is useful to note the fundamental rules for compositionality in a programming language. The *principle of compositionality* is described as follows: "The meaning of a compound expression is a function of the meanings of its parts." [119] It should be stated that the concept of compositionality assumed here is different from the concept of *or-compositionality* [27], which refers to composing logic programs by taking the set-union of their denotations.

The definition of compositionality can be elaborated to a more specific from. For a language to comply with our definition of compositionality, the meaning of every compound expression in the language:

1. Should be defined by only its components and the composition operator.
2. Should *not* depend on the context it appears, the state or the variables that may exist in the context or globally.
3. Should *not* depend on other expressions that may appear before or after it, unless that expression is explicitly one of the name-called components of the expression.

This definition can be formalized as follows.

Definition 1.1.1. *The relational denotation of any operation* $operator(\varphi_1, \ldots, \varphi_m)$ *with operands* $\varphi_1, \ldots, \varphi_m$ *is defined compositionally as a relation-valued function* $F_{operator}$ *of relational denotations of the operands:*

$$[\![operator(\varphi_1, \ldots, \varphi_m)]\!] = F_{operator}([\![\varphi_1]\!], \ldots, [\![\varphi_m]\!])$$

This strict definition also covers the notion of ***referential transparency***. An expression is referentially transparent if it evaluates to the same value regardless of context. Similar concepts have been described earlier, but the term was coined in the form we use now, by Quine in [95]. This property makes it possible to replace any expression with its value without changing the meaning of the host expression or program, and this is related to the principle of *extensionality* in logic. As an example, let us look at the C code below:

```
int k = 5;
int f(int x) { return x + k++; }
...
{
    int a = f(3); // first call
    int b = f(3); // second call
}
```

In the example above, although there are two calls to `f(3)`, they do not produce the same value. While the first one assigns 8 to `a`, the second one assigns 9 to `b`. It is not possible to replace the first call of `f(3)` with its value, because value of the second call depends on the presence of the first call because of the mutable variable `k`. Mutability and free variables break referential transparency, and by extension also compositionality.

In contrast to the example above, let us consider the following simple COMBILOG program and its semantics. The binary *equal* predicate is defined as a composition of two other binary predicates *lessThanOrEq* and *greaterThanOrEq*:

$$equal \leftarrow and(lessThanOrEq, greaterThanOrEq)$$

The denotation of the *equal* predicate is defined as a result of the *and* operator, where the operands are two other predicates. COMBILOG defines the semantics of the *and* operator as follows:

$$[\![and(P, Q)]\!] = [\![P]\!] \cap [\![Q]\!]$$

This definition is in line with the earlier abstract definition of compositionality 1.1.1. The simplicity is greatly due to the lack of free variables. As a result, the denotation of the *equal* predicate is calculated through a function (intersection) of the components of the *and* operator:

$$\begin{aligned} [\![equal]\!] &= [\![and(lessThanOrEq, greaterThanOrEq)]\!] \\ &= [\![lessThanOrEq]\!] \cap [\![greaterThanOrEq]\!] \end{aligned}$$

In contrast to the earlier imperative programming example, in the COMBILOG program each predicate symbol can be replaced by its extension without changing the meaning of the program. This is due to the lack of free variables, and due to every operator in the language following the principle of compositionality. Naturally this feature is not unique to COMBILOG: SKI and BCKW calculi, as well as Predicate-Functor Logic, follow the same principle, but COMBILOG is uniquely suitable for the intended method of synthesis due to compositional recursion operators and proven equivalence to definite clauses.

The COMBILOG example above demonstrates the compositionality principle without explicitly dealing with bindings of arguments (also commonly referred to as domains) of predicates. For this purpose, COMBILOG provides compositional mechanics for argument binding without variables through a *make* operator, which is a generalized version of the argument modification operators found in Quine's Predicate-Functor Logic [98]. This operator will be discussed through examples in the following section. The compositionality principle inflicts a strict restriction on the object language to be used for synthesis. In particular, the lack of variables incapacitates any potential language from employing a common cognitive tool. In the next section we look into how COMBILOG handles argument binding without employing variables.

1.2 Compositional relational argument binding problem

In COMBILOG, a program consists of a set of predicate definitions, mapping predicate names p to compositional expressions φ, such as:

$$p_i \leftarrow \varphi_i$$

COMBILOG expressions are formed using some pre-defined elementary predicates and composition operators. Any compound expression can also be used as a component in another composition. The elementary predicates in COMBILOG are *true* for the universal predicate, *id* for the identity, $const_c$ for constants, and *cons* for constructing data structures, primarily lists. Any elementary predicate is a valid COMBILOG expression, as well as any compound expression written following the syntactic rules of composition.

Composition is established with logic operators *and* and *or*. Let us first look at an example of a simple predicate definition:

$$isFather \leftarrow and(isMale, hasChildren)$$

Since the expression above consists of components that are only unary predicates, it is not necessary to deal with their arguments. For reference, the equivalent definite clause is written as:

$$isFather(X) \leftarrow isMale(X) \wedge hasChildren(X)$$

Logic operators take components with equal number of arguments, and bind them in the order they appear. For components that have more than one argument, and in cases where the arguments are not intended to be bound in the order they appear in the components, a generalized projector operator called *make* is used to modify the arguments of a component.

In order to demonstrate how generalized projection works using the *make* operator, let us assume we have two component predicates available: a unary predicate *isFemale*, and a *parentOf* predicate, which has two arguments, the parent's name followed by the child's name. In order to compose a predicate called *daughterOf* that has two arguments, first, the daughter's name, then the parent's name, in COMBILOG we would write:

$$daughterOf \leftarrow and(make([2,1], parentOf), make([1,2], isFemale))$$

which is equivalent to the following definite clause:

$$daughterOf(X,Y) \leftarrow parentOf(Y,X) \wedge isFemale(X)$$

In the COMBILOG code above, there are 2 components of the predicate *daughterOf*, and they are both projected using *make*. The first

make inverts the order of arguments in *parentOf*, because in *daughterOf*, we want the child to come first. The second *make* merely expands *isFemale* by adding one unbound argument. Because the second index in $make([1,2], isFemale)$ is higher than the arity of *isFemale*, *make* operator produces a predicate expression where the second argument is unbound. This is necessary, because the logic operator *and* expects both components to be of the same arity. This restriction is inherent in a direct variable-free variant of logic programming. In an example case such as $and(p^1, r^3)$, where the arities if p and r are 1 and 3, respectively, which one of r's arguments value will the sole argument of p be bound to? There is not an obvious way to match the arguments of two component predicates if they are not of the same arity.

In order to further expose the issue at hand, we will look at the definition of the *member* predicate. It is a binary predicate which succeeds when the term in the first argument exists as an element in the list term in the second argument. First, let us look at the COMBILOG definition for *member*, and dissect it step by step in order to comprehend its semantics through the corresponding definition in PROLOG.

$$\begin{aligned} member \leftarrow & or(make([1,3], cons), \\ & make([1,3], and(make([4,2,3], cons), \\ & \qquad make([1,2,3], member)))) \end{aligned}$$

The *cons* predicate is one of the elementary predicates in COMBILOG, and it is defined as $cons(X, Y, [X|Y])$, where $[X|Y]$ is a list construction with X as the head and Y as the tail. Because of the nested uses of the *make* operator, the argument bindings in the *member* predicate are difficult to follow. Let us dissect this definition into multiple definitions by writing each operation as a separate clause with a unique name:

$$\begin{aligned} member \leftarrow & or(p, q) \\ & p \leftarrow make([1,3], cons) \\ & q \leftarrow make([1,3], r) \\ & r \leftarrow and(s, t) \\ & s \leftarrow make([4,2,3], cons) \\ & t \leftarrow make([1,2,3], member) \end{aligned}$$

The *member* predicate definition is the disjunction of p and q, placing the restriction that they should have the same number of arguments, which is a result of the *or* operator's semantics in COMBILOG. p has two arguments, where the first is bound to the value of first argument in *cons*, and the second is bound to the value of the third argument in *cons*. The second argument of *cons* is left unbound, due to the index list $[1,3]$ omitting (cropping) the second argument. Similarly, the definition of s

omits the first argument of *cons*, while introducing an unbound argument as the first argument of the head, with the use of the index 4, which is higher than the number of arguments in *cons*. Let us transform this modified COMBILOG program above into a PROLOG program.

$$
\begin{aligned}
member(X, L) &\leftarrow p(X, L).\\
member(X, L) &\leftarrow q(X, L).\\
p(X, L) &\leftarrow cons(X, \_, L).\\
q(X, L) &\leftarrow r(X, \_, L).\\
r(X, Y, L) &\leftarrow s(X, Y, L), t(X, Y, L).\\
s(\_, Y, L) &\leftarrow cons(\_, Y, L).\\
t(X, L, \_) &\leftarrow member(X, L).
\end{aligned}
$$

The *or* operator is split into two separate clauses while preserving the order of the components. Other than the *or* operator, every operator in the COMBILOG code corresponds to a single clause in the PROLOG program above. The argument bindings can be relatively easily observed in the PROLOG program with the use of variables. In the definitions of predicates *member* and r, the variables in the head and each of the components are identical. This is a direct result of the restriction placed by the *and* and *or* operators in COMBILOG. The unbound arguments are written using the anonymous variable '_'. Let us transform this further into a simpler PROLOG program, eliminating the intermediate clauses but leaving the *cons* predicate in place.

$$
\begin{aligned}
member(X, L) &\leftarrow cons(X, \_, L).\\
member(X, L) &\leftarrow cons(\_, Y, L), member(X, L).
\end{aligned}
$$

which can be read as X is a member of L if it is the head of L, or if it is a member of L's tail. Compared to the COMBILOG definition given above, this is much easier to comprehend, but uses variables, and it does not follow the compositionality principle. On the other hand, COMBILOG notation is compositional, but much more difficult to read and modify.

The logic operators in COMBILOG are relatively simple to read, but it is difficult to comprehend the argument bindings. Information that should be immediately accessible requires heavy mental tracing and continuous recollection. Seemingly simple questions such as "is the first argument of the first *cons* bound to the value of any other argument?" are not straightforward to answer. The introduction of unbound arguments only to bring components of a logic operator to the same arity becomes a commonplace pattern, often resulting in unnecessarily repetitive code.

This issue does not arise in functional programming because the value of a function call is only a single argument. This makes it possible to place

the function call as an operand without modifications. But bidirectionality of relational arguments make it cumbersome to write compositional relational expressions, since any argument of a relation may be used as input or output, or both, according to what is permitted by the particular semantics.

There are some variable-free relational notations in the literature, such as *Combinatory Logic* [29, 30, 103], *Predicate-Functor Logic* [96, 98], and *Relational Calculus* [113]. Although these approaches can be mostly considered compositional, they are not practical enough for useful programming. Let us here observe this in the example of Quine's Predicate-Functor Logic (PFL).

We will describe a predicate named *gtePass*, which has three arguments, and behaves as follows: when the numeric value of the first argument is greater than or equal to the numeric value of the second argument, then the value of the third argument is identical to that of the first; otherwise, the value of the third argument is $nil([])$. The definition of the *gtePass* predicate depends on other predicates, namely *greater than*, *greater than or equal to*, *identity*, and *is equal to nil*:

$$gt(X, Y) \leftarrow X > Y.$$
$$gte(X, Y) \leftarrow X \geq Y.$$
$$id(X, Y) \leftarrow X = Y.$$
$$isNil(X) \leftarrow X = [].$$

Let us first define *gtePass* as a pair of definite clauses defining a predicate, using variables X, Y, and Z:

$$gtePass(X, Y, Z) \leftarrow gte(X, Y) \wedge id(X, Z).$$
$$gtePass(X, Y, Z) \leftarrow gt(Y, X) \wedge isNil(Z).$$

The behaviour of the *gtePass* predicate can be demonstrated by the following examples:

$$gtePass(3, 5, []).$$
$$gtePass(4, 5, []).$$
$$gtePass(5, 5, 5).$$
$$gtePass(6, 5, 6).$$
$$gtePass(7, 5, 7).$$

In COMBILOG, the definition of *gtePass* is as follows:

$$\begin{aligned} gtePass \leftarrow or(&and(make([2,1,3], gt),\\ &\quad make([2,3,1], isNil)),\\ &and(make([1,3,2], id),\\ &\quad make([1,2,3], gte))) \end{aligned}$$

As observed earlier with other examples, use of the *make* operator renders COMBILOG programs difficult to comprehend, and modify.

Let us observe the unsuitability of Quine's Predicate-Functor Logic by attempting to write the *gtePass* predicate in terms of these. Consider the predicate-functors *padding*, *expansion*, *rotation* (or *major inversion*), and *minor inversion*. Padding (*exp*) adds an unbound argument to the beginning of an argument list. Cropping (*crop*) erases the first argument. Rotation (*rot*) moves the last argument to the beginning. Minor inversion (*inv*) switches the positions of the first two arguments. Here let us formulate these predicate-functors in set-builder notation:

$$\begin{aligned} [\![exp\ p]\!] &= \{\langle X_0, X_1, \ldots X_n\rangle \mid \langle X_1, \ldots, X_n\rangle \in [\![p]\!]\}\\ [\![crop\ p]\!] &= \{\langle X_2, \ldots X_n\rangle \mid \langle X_1, \ldots, X_n\rangle \in [\![p]\!]\}\\ [\![rot\ p]\!] &= \{\langle X_n, X_1, \ldots X_{n-1}\rangle \mid \langle X_1, \ldots, X_n\rangle \in [\![p]\!]\}\\ [\![inv\ p]\!] &= \{\langle X_2, X_1, \ldots X_n\rangle \mid \langle X_1, X_2, \ldots, X_n\rangle \in [\![p]\!]\} \end{aligned}$$

Let us now write the *gtePass* predicate using the predicate-functors above:

$$\begin{aligned} gtePass \leftarrow or(&and(rot\ rot\ exp\ inv\ gt,\\ &\quad rot\ exp\ isNil),\\ &and(inv\ exp\ id,\\ &\quad rot\ rot\ exp\ gte)) \end{aligned}$$

As is immediately evident by trying to interpret the representation above, comprehending the predicate definition involves a significant cognitive load to simulate the mechanics of argument manipulation. This is by no means a shortcoming of PFL, since it is not intended to be a practical notation. Quine's motivation was to establish a variable-free notation for First-order logic, which he did accomplish with PFL.

As demonstrated with COMBILOG and PFL through example, existing compositional representations of relational programs are not practical. This makes the particular method of synthesis, namely, meta-interpretative inductive synthesis by decomposition, inaccessible due to usability issues of the object language. This brings us to our research question:

Research question: Can we make this particular method of synthesis more accessible by improving the usability of the target language? Specifically, we aim to find out if it is possible to improve the representation of the variable-free compositional relational language COMBILOG in terms of comprehensibility and modifiability. Since COMBILOG is model-theoretically equivalent to definite clauses, the question can also be stated as: Are there more usable forms of expressing definite clause programs, which follow the principle of compositionality at the expression level?

1.3 Method

> "*There are many senses in which a program can be 'good', of course. In the first place, it's especially good to have a program that works correctly. Secondly it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language.*
> *Donald Knuth, Turing Award lecture (1974) [58]*"

It is evident that the programming language community has been aware of the importance of human aspects in programming language design. Yet, the cognitive aspects of the programming language notation has mostly been an after-thought. There are many usability studies on various programming languages, but they are often performed after a language is already established. A historical survey on the design process of programming languages reveals a lack of employing empirical data for language design [66], therefore most claims about language usability features are not reassuring. While some empirical evidence can be found, those involving user studies are rare [59]. Recent research confirms that human factors have very little effect on programming language design [111]. Let us give an overview of the methods available for incorporating human factors within programming language design.

Some methodological tools for evaluating and improving development environments and programming languages are based on those established for graphical user interfaces [123]. One of these is *Cognitive Walkthroughs* [94], which determines sequences of actions for accomplishing specific tasks, and refines the user experience by going through each sequence with questions directed towards improving the interface. Another is *Heuristic*

Evaluations [77], which is an evaluation of a user experience against an established set of principles (heuristics) by an expert evaluator.

Abstract mental models of programmer behaviour is a necessary input for most methods. There were some attempts to build such cognitive models from a series of usability experiments [107]. Some other attempts involve cognitive simulation, building models using established cognitive modelling methods such as *Cognitive Architectures* [2, 53].

There are also more specific techniques for assessing the usability of programming languages and tools. Some of these focus on low level aspects, such as eye tracking with specialized equipment. A recent example uses eye tracking to extract developers' focus points in program code in order to combine it with user interaction information from the IDE [56]. With this novel method, building a complete aggregated model of program interaction was accomplished. Some studies use even lower level observations such as Functional Magnetic Resonance Imaging (fMRI), to measure activations in brain regions, during tasks such as program comprehension and locating syntax errors [108].

The main aim of this thesis is to establish more usable representations for the *Compositional relational argument binding problem* described earlier. We intend to establish a method that does not ignore human factors, as described in [66] and [111]. Accordingly our representational claims are based on empirical evidence gathered critically from usability studies confirming the alleged improvements. Designing usability tests often requires specific intentions, such as target groups or features. In the case of COMBILOG, specializing to a narrow audience or domain is not necessary since the problem of notation readability is a fundamental issue, surpassing specific cases. In a broad sense, COMBILOG would appeal to the people involved in purely declarative logic programming, for academic or industrial purposes.

To identify fundamental issues regarding the notation of COMBILOG, we shall refer to *Cognitive Dimensions*. These can be considered as a set of heuristics for evaluating formal notation design, defined as a set of definitions for various aspects of notation usability, introduced by Green in 1989 [42] and improved further in multiple publications by Green and other contributors [43, 44]. Although they do not provide an exact and objective evaluation framework, they are suitable for labelling usability phenomena from a subjective perspective. Therefore we will use cognitive dimensions as a means of identifying the issues with COMBILOG notation but not for assessing the success of any improvements made. For the latter purpose, we will employ usability tests that are targeted for the specific issues identified by our analysis through cognitive dimensions. These methods constitute the main component of this work, which involves identifying and addressing the usability issues with compositional relational argument binding in the example of COMBILOG, and ultimately

confirming our solutions. In order to demonstrate the feasibility of these solutions, we put forward concrete implementations in each case.

As an initial step, we will give an analysis of COMBILOG through cognitive dimensions and identify the core aspects that make the notation difficult to comprehend and modify. Then, we will present two separate instances of improvement over COMBILOG.

Our first approach is an augmentation of the COMBILOG notation via coloured diagrams to aid comprehension. This diagram system which we refer to as VISUAL COMBILOG is devised with the help of Bertin's *Visual Variables*, a formalization of visual dimensions available and their powers corresponding to specific uses [13]. An early iteration of our work on VISUAL COMBILOG is presented in previous publications [83, 82]. In particular, our implementation of a split-view editor is shown, as are the results of a usability test comparing VISUAL COMBILOG to read COMBILOG programs as opposed to plain COMBILOG. Here we give a more detailed account on the development of the visual system, as well as a graph formalization of VISUAL COMBILOG.

Our second approach is an alteration of the COMBILOG notation while keeping it text-based. This new notation, which we call *Combilog with Name Projection* (CNP), replaces the indices used in the COMBILOG notation with names (or labels). Using names directly enhances usability, as well as indirectly allowing us to redistribute the cognitive load on the operators of the language. A formal treatment of names for argument positions is not straightforward as it requires a new concept of relational extensions that includes argument names. We present the formal semantics of CNP; describe how it can be transformed to COMBILOG; and provide a meta-interpreter implementation in PROLOG. We also give the results of a user study measuring the effect of using CNP versus a similar first-order notation that uses variables. As a demonstration of how CNP can be used instead of COMBILOG, we implement decompositional synthesis which uses CNP as the target language.

In summary, the methods of this thesis have been gathered from various fields. We use heuristic evaluations, visual design principles, and usability testing from human-computer interaction as a foundation for improving usability, while using programming language theory from computer science to prove the reliability of our improvements.

1.4 Related work

In COMBILOG, the variable-free form is a result of compositionality. Since the meaning of COMBILOG expressions cannot depend on anything other than their specified arguments: free variables, or variables that cross the boundaries of a single operation are not permitted. For this reason, here

we give an account of earlier forms of variable-free algebra and their representation. For a comprehensive historical account of quantified variables, λ-calculus, and Combinatory Logic, the reader is referred to the earlier work by Cardone and Hindley [17].

Variables are an essential concept in mathematics, but their formalization as we understand them today is a relatively recent accomplishment. Giuseppe Peano, in 1889, described a system of substitution, where symbols are replaced by other symbols via a well-defined set of axioms [86]. Gottlob Frege was the first to define quantified variables, by associating a symbol with a domain of discourse, the objects it can *stand for* [39, 40]. In 1885, Charles Sanders Peirce independently formalized universal quantification and existential quantification [89], which later led to his development of *existential graphs* [90].

The place of variables in mathematics has been a topic of ongoing conversation. Schönfinkel, Curry, Quine, De Bruijn, and Tarski and Givant proposed various efforts to establish formalizations of mathematical concepts without depending on variables. In the following sections, we give an account of these attempts.

1.4.1 Combinatory Logic

We can find the earliest attempt at variable-free notations in Schönfinkel's work in [103]. Schönfinkel considered variables not to be fundamental to the nature of mathematics, and formalized a set of combinators that enable writing combinator expressions that do not depend on variables. He showed that using the combinators he defined, S, K, and I, the same expressibility as predicate logic is established.

Independently from Schönfinkel's work, Curry also established a system consisting of combinators he named B, C, K, and W in [29]. He devised a method for transforming expressions with variables into the variable-free form he defined, and proved the completeness of the BCKW system. Curry's BCKW and Schönfinkel's SKI systems of combinatory logic are equally expressive and can be transformed into each other.

Predicate-Functor Logic is a method defined by Quine in [96, 97, 98], and refined in [99], also describing a way of manipulating first-order predicates with variables without directly dealing with variables themselves as a part of the notation. He defines operations including *rotation*, *inversion*, *cropping* and *padding*: each denoting a specific way to modify the argument list of a given predicate, yielding a new derived predicate. By repeatedly applying these operations, desired bindings for arguments of predicates can be established. The soundness and completeness of the Predicate-Functor Logic was proven later by Bacon [6].

All three of these methods are intended for eliminating variables for theoretical intentions, not human-centric ones. Hence, the use of any of these notations for actual programming purposes with Compositional Relational Programming would be troublesome at the very least.

1.4.2 Relational calculi

In [113] Tarski derived a *Relational Calculus* building on the earlier work of De Morgan [34], Peirce [90] and Schröder. His work in [113] comprises axioms of strictly binary relations, and defines operations including but not limited to *relative multiplication* (corresponding to conjunction over a common domain) and *relative addition* (corresponding to disjunction over a common domain), which we refer to as forms of relational composition. He makes the distinction between *individual variables* and *relational variables*, and restricts the definitions of the theory to relational variables only, which would qualify as a variable-free form in a first-order configuration. When he employs individual variables, he uses infix notation as in xRy as opposed to $R(X, Y)$ or $R : X \times Y$ which could be considered more common notations today. Later, Tarski and Givant show that formulae in set theory can be translated to Relational Calculus without using individual variables. [115]

Codd developed *Relational Algebra* [25, 26], during his work at IBM, primarily for establishing a theoretical framework for data retrieval queries on relational databases, which formed a basis for commercial relational query languages such as the Structural Query Language (SQL) [18, 118] and later the Language Integrated Query (LINQ) [70]. He proposed using names instead of indices for relation domains, aiming for a user-friendly notation. He observed that the number of domains (columns) in a commercial database table often reaches up to 30, and concluded that demanding users remember the position of each domain is not practical. This is perhaps the work with the greatest similarity to our development of CNP, since we also invoke names for domains of predicates. Although the usability improvements are similar, Relational Algebra is not suitable for programming purposes in general, particularly due to decreased expressiveness such as the lack of fixpoint operators [1].

1.4.3 De Bruijn indices

De Bruijn indices is an alternative notation to λ-calculus, introduced in [33]. It employs indices instead of variable names, referring to the λ-bound variables in the context. De Bruijn indices are syntactically destitute of variable names, which grant them significant advantages. Two formulas

using De Bruijn indices are equivalent if and only if they are syntactically identical. This is in contrast to λ-calculus, where two formulas such as $\lambda x.x$ and $\lambda y.y$ need to be checked for alpha-equivalence to answer if they are equivalent with respect to alpha-conversion. Another issue that arises while using variables is variable capture after substitution, which is not a concern with De Bruijn indices. On the other hand, because De Bruijn indices refer to specific arguments in the context, they exhibit high coupling with the context and hence do not follow the compositionality principle. COMBILOG's *make* operator uses a similar method with indices, but in that case the index list refers to the immediate arguments of a given source predicate expression. This contrasts with De Bruijn indices, which by definition refer to the *outside* context, for any index greater than 1.

A notable application of De Bruijn indices is by McBride and McKinna in [68] which uses a combination of indices and names, providing a more practical way for dealing with free variables using names, and using indices for bound variables, avoiding alpha conversion. It stands as an example of impracticality of De Bruijn indices for human manipulation, and their appropriateness for computation.

1.4.4 Nominal logic

In [19] and [20], Cheney presents αProlog, a programming language specializing in dealing with programs which manipulate and reason about name binding, such as interpreters, compilers, and theorem provers. The language is based on *nominal logic* developed earlier in [41] and [92] by Gabbay and Pitts that utilizes name abstraction, swapping and freshness as constructs for writing logical clauses.

Our work differs from Cheney's in that we focus on more general logic programs with usability in mind, while his focused on name-manipulating logic programs. Our solution also does not require an addition for capture-avoiding substitution as the names defined in CNP have no scope in contrast to variables. The freshness and swapping constraints in the language lead to use of a constraint solver for αProlog, while CNP expressions are bijectively transformed into COMBILOG ones and into definite clauses which can be directly executed by SLD Resolution.

1.4.5 Compositional Logic Programming

McPhee, in his doctoral thesis *Compositional Logic Programming* [69], takes the fair computation rule as a basis for compositionality, and focuses on a few novel implementations of SLD Resolution using tabling and prioritization. The main concern is identifying a search algorithm that is fair and terminates more often than others. The main difference

between [69] and us, is that McPhee focuses on operational aspects of compositionality, such as the fair computation rule, while we focus on human-centric, syntactic usability. The Prioritized SLD Resolution presented can be used for future implementations of the languages that we devise here, since we share the common compositionality requirements. In [69], it is argued that a compositional relational declarative language can never compete with imperative languages in efficiency. We do not make any efficiency claims regarding our implementations, but we base the importance we give to purely declarative languages on the idea that the execution models of these languages may employ automatic planning more efficiently for implicit parallelization.

In [104, 105], Seres et al. present an embedding of algebraic logic programming in Haskell using its lazy evaluation principle. In particular, they investigate the program transformation and algebra of functional-logical expressions produced in this manner. The laziness property of Haskell aids in eliminating branches of computation in the implementation. The computation mechanism is a form of *narrowing* rather than *resolution*. Perhaps the most important result of [105] is the demonstration of how examples of program transformation from functional programming can be applied to logic programming. Although we argue that a compositional relational paradigm inherently allows algebraic uses, we do not delve into the subject in this work.

Compared to Cheney's, McPhee's and Seres's work, one of our contributions, CNP, differs in that we define names as a part of the extension of a predicate, which provides a more practical notation for dealing with variable-free expressions. None of the listed work submits to a variable-free form. We present a notation for relation composition via name-inferring logic operators without consulting to variables. The most significant contrast is that we approach the problem of representation with an human-centric toolbox. All the earlier work mentioned has contributed significantly to the field of logic programming, but none of it deals with the problem of representation practicality.

1.5 Contributions

As remarked earlier, our main contribution is making compositional relational programming, in the example of COMBILOG, more accessible by altering its representation. To achieve this, a strict contextual definition of compositionality and declarativity is fundamental, since any alterations that would destroy the principle of compositionality would not constitute an appropriate solution for COMBILOG. Both representations we design adhere to the principle of compositionality defined in section 1.1, and remain transformable to COMBILOG, and by extension to definite clause

programs. This also maintains the execution model of COMBILOG programs, i.e. SLD Resolution.

Our contributions in this work are predicated upon usability. We would like to establish user-friendly representations for COMBILOG programs, that make working with compositional relational programs easier.

We first develop a graphical system that augments the textual representation of COMBILOG with corresponding diagrams that provide a rich visual realisation. We present the results of a usability study measuring interpretation time and the comprehensibility of this visual language called VISUAL COMBILOG. We observe that it brings significant improvements to readability over plain COMBILOG. We also describe our prototype implementation of the VISUAL COMBILOG, which uses a split-view to allow editing of COMBILOG code or the corresponding diagram interchangeably.

As a second attempt to improve the usability of COMBILOG programs, we present a new textual syntax which redistributes the cognitive load on the operators of the language, particularly lifting some of the load on the *make* operator by moving the argument introduction function to logic operators. We accomplish this by devising a concept we name α-extensions, which are relations consisting of tuples that use names for accessing their elements instead of numeric positions. These name-aware tuples are identical to the well-known concept of *records*. Embedding argument names as a part of the predicate extensions allows us to define name-aware operators which work in cooperation with this new information, without depending upon variables. This new notation with argument names and altered operators is named Combilog with Name Projection, or CNP. Like we did with VISUAL COMBILOG, we present the results of a usability test comparing CNP to more conventional representations with variables found in logic programming languages such as Prolog or Mercury, observing that CNP is similarly practical for short definitions. We prove the model-theoretic equivalence between CNP and COMBILOG, and present an implementation of the CNP language as a meta-interpreter written in PROLOG. Finally we present an example of Inductive Logic Programming that uses CNP as a target language, in the same essential decompositional synthesis method that is applicable to COMBILOG, demonstrating the feasibility of using CNP instead of COMBILOG.

Surprisingly, the design of programming languages has mostly been based on technical requirements, rather than human-centric usability [111]. Even with mainstream programming languages that have frequent release cycles, e.g. Java, C++ and C#, it is hard to find evidence that human-centric empirical studies on language usability are used in the design process. Post-partum, after-the-fact studies such as [24] exist, but their results are rarely integrated into the language design. In this work we take a different approach, and treat the notation of a programming

language as a user interface. As a tool we use Green's *Cognitive Dimensions* [42] to discuss shortcomings of the plain COMBILOG notation, and we base our improvements of VISUAL COMBILOG and CNP on these observations. By pairing an analysis stage using Cognitive Dimensions with confirmatory usability tests, we establish a methodology which is potentially applicable for improving the usability of other programming languages.

1.6 Overview

In Chapter 1, we define the concepts of compositionality and declarativity for compositional relational logic programming. We also identify the problem of argument binding, and briefly visit earlier work investigating this issue. We conclude, by describing the contribution this thesis makes to the field.

Chapter 2 revisits the fundamental concepts of Logic Programming to act as a glossary relevant for later chapters in the thesis. It follows with a definition of COMBILOG, which is the language we take as the point of departure. We include COMBILOG's syntax and semantics, discuss their relation to definite clause programs, and prove their equivalence through a fixpoint correspondence.

Chapter 3 discusses COMBILOG's textual representation in terms of Cognitive Dimensions, and establishes design goals for VISUAL COMBILOG and CNP.

Chapter 4 presents the design of VISUAL COMBILOG, the results of the usability study, and describes the split-view prototype editor.

Chapter 5 introduces CNP syntax, and explain its constructs in relation to COMBILOG. It includes the proof of fixpoint equivalence between CNP and COMBILOG.

Chapter 6 gives the results of the usability test evaluating the effectiveness of CNP syntax.

Chapter 7 presents an application of CNP for program synthesis, which is the original motivation behind the invention of COMBILOG.

Chapter 8 summarizes our contribution, discusses the results and present an overview of possible future research paths that have emerged.

Appendix B gives an implementation of a COMBILOG interpreter. Appendices C and D present an implementation of the CNP language, and a decompositional synthesizer that uses CNP as the target language, respectively.

2. Background

This chapter comprises a brief introduction to concepts of *Logic Programming* in the first section. This introductory summary is limited to the concepts regarded as fundamental for understanding the material in this work, and follows the definitions in [62]. Readers who are familiar with first-order logic, definite clause programs and their semantics can skip to the next section. For a thorough reading on the subject the reader is referred to Lloyd [62].

The second section defines the COMBILOG language and its semantics, and does not present any novel contributions. But it is included for a better understanding of the rest of this thesis, as the main contributions are related to improving the usability of COMBILOG.

2.1 Logic Programming

Logic Programming refers to a family of programming languages and the practices based on various forms of formal logic. A logic program consists of clauses written in a suitable logical form, comprising a *theory* [62].

First-order logic is the formal logic system characterised by quantified variables ranging over individuals (terms), as opposed to second-order or higher-order logic systems in which variables may range over predicates or functions as well. In the context of this work, the relevant forms of logic programs are those based on first-order logic.

2.1.1 First-order logic

The elements of first-order logic consist of variables, variable quantifiers, constants, functions, predicate, and logical connectives. These concepts constitute a first-order language. Conventionally variables are denoted as upper-case letters such as $X, Y, Z, \ldots$, and constants as lower-case letters such as $a, b, c, \ldots$. Quantifiers are the *universal quantifier* $\forall$ and the *existential quantifier* $\exists$, and they are always associated with a variable such as $\forall X\ p(X)$. Predicate and function symbols are usually denoted with lower-case letters $p, q, r, \ldots$ and $f, g, h, \ldots$ respectively.

The logical connectives of a first-order language include conjunction $\wedge$, disjunction $\vee$, negation $\neg$, implication $\rightarrow$ and bidirectional implication (logical equivalence) $\leftrightarrow$.

Next, we shall give the basic definitions regarding the well-formed formulas of a first-order language, Let us start with *term*s.

Definition 2.1.1. *Terms of a first-order language consist of the constants, variables, and functions, where a function term is the application of a function symbol to zero or more terms.*

It follows that, if there is at least one function symbol in a first order language, the set of terms is infinite due to the recursive definition of a function term. When a number of terms are constructed as a sequence, these structures such as $\langle t_1, \ldots, t_n \rangle$ are called tuples.

Definition 2.1.2. *Ground terms of a language are those that do not contain variable terms. This includes constants and function terms constructed using only ground terms.*

Terms can be used to construct instantiations of predicates or functions. Let us give a definition of a first-order predicate:

Definition 2.1.3. *A predicate is a structure that maps tuples to a boolean value true or false.*

Definition 2.1.4. *The number of arguments of the predicate, or equally, the number of terms in each tuple of the predicate determine the arity of a predicate.*

Since we have defined terms, predicates, and logical connectives from earlier, we can define a well-formed formula in a first-order language:

Definition 2.1.5. *A first-order formula is well-formed if it follows one of the following definitions:*

1. *Given that p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, $p(t_1, \ldots, t_n)$ is a well-formed formula (also more specifically an atomic formula, or an atom, or a positive literal).*
2. *Given that E is a well-formed formula, $\neg E$ is a well-formed formula. If E is an atom, $\neg E$ is also called a literal.*
3. *Given that E and F are well-formed formulas, $E \vee F$, $E \wedge F$, $E \rightarrow F$, $E \leftrightarrow F$ are well-formed formulas.*
4. *Given that X is a variable and E is a well-formed formula, $\forall X\ E$ and $\exists X\ E$ are well-formed formulas.*

Let us observe this statement in first-order logic:

$$\forall X\ \forall Y\ isFather(X, Y) \leftarrow isParent(X, Y) \wedge isMale(X)$$

This statement can be translated to plain English as 'Whenever X is the parent of Y and X is a male, X is the father of Y'. It is a well-formed first-order formula, since it follows the rules of constructing well-formed formulas. The variables in this formula are X and Y, predicate symbols are *isFather*, *isParent* and *isMale*, and atoms are $isFather(X, Y)$, $isParent(X, Y)$ and $isMale(X)$. Since every variable is quantified, it is considered a *closed formula*, and since every variable is quantified in the most general scope, it is considered to be in the *prenex normal form*. All quantifiers in this formula are universal quantifiers, which qualifies it to be in the *Skolem normal form* which is a special form of the prenex normal form where all the quantifiers are universal.

Definition 2.1.6. *A* clause *is a disjunction of literals in Skolem normal form, exemplified as follows, where $L_1, \ldots, L_n$ are literals and $X_1, \ldots, X_m$ are the distinct variables appearing in the literals:*

$$\forall X_1 \ldots \forall X_i \ (L_1 \vee \ldots \vee L_n)$$

it is often written omitting the quantifiers as well, as:

$$L_1 \vee \ldots \vee L_n$$

As defined earlier, literals are either positive (A) or negative ($\neg B$) atoms. A clause can also be written in the *implicative form*, where the disjunction of positive atoms $A_1 \vee \ldots \vee A_n$ are placed at the consequent side of an implication, and using the logical equivalence

$$\neg(\neg B_1 \vee \ldots \vee \neg B_m) \iff B_1 \wedge \ldots \wedge B_m$$

the conjunction of atoms $B_1 \wedge \ldots \wedge B_m$ are placed at the antecedent side:

$$A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m$$

or replacing the disjunction and conjunction connectives with commas, as they are constantly the same sign on both sides of the implication:

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

Next, we describe a special category of clauses, definite clauses, and follow with their semantics.

2.1.2 Definite clause programs

Definite clauses have certain properties which make them useful for practical applications. Let us first define a definite clause:

Definition 2.1.7. *A definite clause C is a clause that has exactly one positive literal, which is written as:*

$$\text{in implicative form: } A_1 \leftarrow B_1, \ldots, B_m$$
$$\text{in disjunctive form: } A_1 \vee \neg B_1 \vee \ldots \neg B_m$$

The atom in the consequent A_1 is called the *head* of the definite clause, and the conjunction of atoms in the antecedent $\neg B_1, \ldots, \neg B_m$ is called the *body*. A definite clause with an empty body is called a *unit clause*, or a *fact*.

Definition 2.1.8. *A* definite program $\mathcal{P}$ *is a set of definite clauses.*

In a definite program, there may be multiple definite clauses with their heads containing the same predicate symbol p. The collection of these definite clauses is called the *definition of* p.

Definition 2.1.9. *A clause with an empty consequent is called a* definite goal*, written as:*

$$\text{in implicative form: } \leftarrow B_1, \ldots, B_m$$
$$\text{in disjunctive form: } \neg B_1 \vee \ldots \vee \neg B_m$$

2.1.3 Model-theoretic semantics

In the earlier sections, the grammar of first-order languages, and definite clause programs were defined. In order to discuss model-theoretic semantics of definite programs, further concepts need to be introduced.

Initially the concept of predicate extension, interpretations and models will be introduced. Then, Herbrand variants of these concepts will be discussed, which restrict the domain of discourse to the Herbrand Universe. Upon these concepts, we will build a fixpoint theorem which constructs a model of the program, that is, the model-theoretic meaning of the program. This is established via interpretations as partial models, using an immediate consequence operator.

Definition 2.1.10. *The extension of a predicate is the set of tuples which the predicate maps to true. The extension of a predicate p that has n arguments is denoted as $[\![p]\!]$, and defined as follows:*

$$[\![p]\!] = \{\langle t_1, \ldots, t_n \rangle \mid p(t_1, \ldots, t_n)\}$$

In order to assign a meaning to a program, or a clause, it is imperative to begin from the smallest elements. For this reason, the concept of an *interpretation* is used, which assigns meanings to parts of a formula.

For any definite program $\mathcal{P}$, the *domain of discourse* (D) is the non-empty set which all the variables in the program may range. If D is empty for a particular program, a constant c is assumed to be included in D.

The concept of an *interpretation* from [62]:

Definition 2.1.11. *An* interpretation *is a set of assignments for constants, function and predicate symbols in a program. Particularly:*

(*a*) *Every constant in* $\mathcal{P}$ *is assigned to an element from* D.

(*b*) *Every n-ary function symbol in* $\mathcal{P}$ *is assigned to a mapping from elements in* D^n *to an element in* D*, where* D^n *is the n-fold Cartesian product of* D.

(*c*) *Every n-ary predicate symbol in the program is assigned to a mapping from* D^n *to boolean values* true *or* false

Interpretations can be used to assign truth values to clauses by looking up the truth value of a predicate for particular arguments, and calculating the truth value of the whole clause via the logical connectives. This would involve first finding the distinct set of variables in a clause, assigning them to elements in D, looking up the truth values of every atom from an interpretation I, and calculating the final truth value of a clause via the negations and disjunction connectives, according to the following:

- $p \wedge r$ is true only when both p and r are true.
- $p \vee r$ is true if at least one of p and r is true.
- $\neg p$ is true when p is false.
- $p \leftarrow r$ is true if 'p is true whenever r is true'.
- $p \leftrightarrow r$ is true when both p and r are true or both of them are false.

If we can assign *true* to a clause in this way, that clause is *satisfiable* with regard to I. Similarly, if we can assign *true* for every instance of a clause, that clause is *valid* with regard to I.

Definition 2.1.12. *An interpretation* I *is a* model *for a clause* C *if* C *is* true *with regard to* I*. Consequently,* I *is a model of a program if it is a model for every clause in the program. If there is a model for a program* $\mathcal{P}$*,* $\mathcal{P}$ *is* consistent.

For a program $\mathcal{P}$ and its every interpretation I, if I being a model of $\mathcal{P}$ implies that it is a model for a clause C, then C is a *logical consequence* of $\mathcal{P}$.

Further definitions will involve Herbrand concepts. These are counterparts to interpretations and models defined earlier, but with the domain of discourse being limited to a so-called Herbrand universe.

Definition 2.1.13. *The* Herbrand universe H *of a program* $\mathcal{P}$ *consists of ground terms appearing in the program, and all terms that can be built*

from these ground terms and the function symbols in the program without using variables. If there are no ground terms in a program, an arbitrary constant c is added to make the Herbrand universe non-empty.

Definition 2.1.14. *The* Herbrand base *B of a program $\mathcal{P}$ is the set of ground atoms formed by instantiating every n-ary predicate symbol p appearing in $\mathcal{P}$, with terms from the Herbrand universe. This is formulated as follows, where H^n refers to the n-fold Cartesian product of H:*

$$B_{\mathcal{P}} = \{p(t_1, \ldots, t_n) \mid \langle t_1, \ldots, t_n \rangle \in H^n, \text{and } p \text{ is a predicate in } \mathcal{P}\}$$

Definition 2.1.15. *A* Herbrand interpretation *of a program $\mathcal{P}$ is an interpretation where the domain of discourse is restricted to H ($D = H$). Consequently, a Herbrand interpretation is a subset of the Herbrand base for the same program.*

Definition 2.1.16. *A* Herbrand model *of a program $\mathcal{P}$ is a Herbrand interpretation of $\mathcal{P}$ which is a model for $\mathcal{P}$.*

With the help from Herbrand concepts, we can now define a model-theoretic meaning for a predicate:

Definition 2.1.17. *The least Herbrand model of p in a program $\mathcal{P}$ is the set of ground atomic instances of p from the Herbrand base B, which are logical consequences of the program $\mathcal{P}$:*

$$\mathcal{M}_{\models}(\mathcal{P})(p) \;=\; \{p(t_1, \ldots, t_n) \in B_{\mathcal{P}} \mid \mathcal{P} \models p(t_1, \ldots, t_n)\}$$

Definition 2.1.18. *The least Herbrand model for a program $\mathcal{M}(\mathcal{P})$ is the union of least Herbrand models for every predicate p in $\mathcal{P}$. This is also called the* model-theoretic meaning *of the program [3, 62, 120].*

In the next section, we look at the construction of a least Herbrand model for a program through fixpoint semantics.

2.1.4 Fixpoint semantics

For definite clause programs, the least Herbrand model of a program can be obtained by the least fixpoint of a monotonic logical consequence operator associated with the program, as proven by van Emden and Kowalski [120]. Here we first define this operator, and other relevant concepts towards establishing the fixpoint semantics of a definite clause program $\mathcal{P}$.

Definition 2.1.19. *The set of ground clauses of a program $\mathcal{P}$ is written as $\Gamma_{\mathcal{P}}$, and is constructed by substituting ground terms from the Herbrand universe for every variable in the clause.*

The immediate consequence operator for a program $\mathcal{P}$ maps Herbrand interpretations to Herbrand interpretations, where the returned Herbrand interpretation includes the ground atoms that are immediately implied by the ground clauses of the program and the given Herbrand interpretation.

Definition 2.1.20. *An immediate consequence operator for a program $\mathcal{P}$ is written as $T_{\mathcal{P}}$, and it is defined as follows:*

$$T_{\mathcal{P}}(I) \;=\; \{A_0 \mid A_0 \leftarrow A_1, \ldots, A_n \in \Gamma_{\mathcal{P}} \wedge \{A_1, \ldots, A_n\} \subseteq I\}$$

Then the least fixpoint of a program $\mathcal{P}$ is defined as the least I such that $I = T_{\mathcal{P}}(I)$ [114].

The next step is defining an ordinal power function $\uparrow$ for T.

Definition 2.1.21. *The power function $T_{\mathcal{P}} \uparrow i$ is defined as follows, where ω is the first infinite ordinal:*

$$\begin{aligned} T_{\mathcal{P}} \uparrow 0 &= \emptyset \\ T_{\mathcal{P}} \uparrow (i+1) &= T_{\mathcal{P}}(T_{\mathcal{P}} \uparrow i) \\ T_{\mathcal{P}} \uparrow \omega &= \bigcup_{i=0}^{\infty} T_{\mathcal{P}} \uparrow i \end{aligned}$$

The model-theoretic meaning of a program $\mathcal{M}_{\models}(\mathcal{P})$ coincides with the unique least fixpoint [120]:

$$\mathcal{M}_{\models}(\mathcal{P}) = \mathcal{T}_{\mathcal{P}} \uparrow \omega \tag{2.1}$$

Next, we move on to the proof-theoretic semantic of definite programs with the concepts of *unification* and *resolution*.

2.1.5 Proof-theoretic semantics

The proof-theoretic meaning of definite clauses is determined by the SLD Resolution (SL Resolution for Definite Clauses) algorithm [120]. This is a specialization of the earlier SL Resolution (Selective Linear Resolution) [61] and the original Resolution algorithm [101].

The Resolution algorithm, together with a first-order unification algorithm provides the operational semantics for definite clause programs.

Let us begin by describing the unification algorithm.

Unification

Unification is a procedure for establishing equality between objects of a program, or the lack of it. It dates back to Herbrand [62] but was practically adopted for the first time by Robinson in [101]. Briefly, if the two objects are strictly identical, they are said to *unify*. If they are not strictly identical but can be made identical with certain modifications, they are said to *unify with substitutions*. If there are not any set of modifications that can make a pair of objects identical, they are said to not unify. Let us observe a few examples of unification, where constants are displayed in lower-case, and variables in upper:

$$\begin{aligned} a \text{ and } a &\mapsto \textit{unify}. \\ a \text{ and } b &\mapsto \textit{do not unify}. \\ X \text{ and } a &\mapsto \textit{unify with substitutions } \sigma = \{X = a\}. \\ p(X, a) \text{ and } p(c, Y) &\mapsto \textit{unify with substitutions } \sigma = \{X = c, Y = a\}. \end{aligned}$$

A unification algorithm takes a pair of objects as input, and outputs a *most general unifier* through a set of substitutions, if a unification is possible. A unifying substitution σ for a set of objects is the *most general unifier* if, for any unifying substitution ω for the same set of objects, there is a substitution δ so that the composition of ω and δ gives σ: $\sigma = \omega\delta$ [62]. This establishes a method to curb the search space for the Resolution algorithm by allowing higher-level syntactic eliminations as opposed to testing for ground instances as in earlier works such as the Davis-Putnam algorithm [32]. Variations of the unification algorithm exist, including more efficient ones such as [67, 85], parallelly executable ones [10], or ones with different purposes such as the variable-free unification in [5], but this basic definition is sufficient for our purposes in this work.

SLD Resolution

The SLD Resolution is a search algorithm which takes a set of definite clauses, together with a definite goal, and answers if that goal logically follows from the given set of definite clauses [120].

The set of definite clauses are assumed to be in the *conjunctive normal form* (CNF), that is, a conjunction of disjunctive clauses. A definite program is considered to be in this form, since every clause is a disjunction of literals, and the program is taken as a conjunction of its clauses.

The elementary component of the algorithm is the resolution of a single definite clause and a definite goal, given by the *resolution inference rule*.

Definition 2.1.22. *Resolving a definite goal G with a definite clause C results with a new definite goal G' by unifying a literal from G with a complementary literal from C. This is provided by the* resolution inference

rule. *Given a definite clause and a definite goal:*

$$G = \{\neg B_0, \ldots, \neg B_m\}$$
$$C = \{A_0, \neg A_1, \ldots, \neg A_n\}$$

application of the resolution inference rule requires a literal in G, namely B_k, and a literal in C, namely A_j to be complementary, and to unify with substitutions θ. If they unify, the resulting resolvent *goal clause is given as:*

$$resolvent(G, C) = subs(\theta, (G - B_k) \cup (C - A_j))$$

Intuitively, assuming the C and G are standardized apart (they share no common variables) to begin with, the resolvent goal clause is the union of the two, except the literals used to modify them. The resolvent also reflects the substitutions required by the unification, if any.

Since every literal in the goal clause is negated, they can only resolve with the head literal of a definite clause, since it is the only positive literal in a definite clause. When the unifying literals are removed, the only remaining literals are negative, which renders the resulting clause another goal clause.

The Resolution algorithm can only be used to derive *false* from an unsatisfiable set of formulas, hence refuting the set of formulas, but it cannot be used to derive every consequence. Therefore, it is only *refutation-complete*. When coupled with a complete search algorithm, the resolution inference rule gives a theorem proving algorithm which is also only refutation-complete. But it is possible to exploit refutation-completeness for logical inference. Given a conjunction of clauses $\mathcal{KB}$ (knowledge base) and an original goal G, if the negation of the original goal $\neg G$ does not follow from $\mathcal{KB}$, we can conclude that G in fact does follow ($\mathcal{KB} \vdash \mathcal{G}$), according to the *law of excluded middle*. This is an instance of *proof by contradiction*.

The resolution algorithm is initiated with a set of clauses $C_1, \ldots, C_n$ corresponding to the program, and a goal $\neg G$. A clause C_i is selected to resolve with $\neg G$, and their resolvent G_1 is assumed as the new goal. A new clause C_k is selected to resolve with G_1, and this is repeated until an empty clause $\emptyset$ is reached, which is a clause with no literals. If an empty clause is reached as a resolvent, it implies that the initial goal $\neg G$ was inconsistent with the program, and hence G must follow. The search for empty clause may not terminate, in which case it implies that an inconsistency cannot be found. If it terminates, it will be through a linear sequence of resolvents, ending with the empty clause $\emptyset$, which signifies the contradiction.

$$\neg G,\ G_1,\ \ldots,\ G_{n-1},\ \emptyset$$

If a goal G can be shown to follow from a program $\mathcal{P}$ using the SLD Resolution method, it is written as:

$$\mathcal{P} \vdash G$$

In the next section, we will define the proof-theoretic meaning of a program based on SLD Resolution.

Proof-theoretic meaning

In the previous section, the SLD Resolution method was described, and a proof-theoretic consequence ($\vdash$) was established. Based on this we can devise a proof-theoretic meaning of a program.

Definition 2.1.23. *The proof-theoretic meaning of a predicate p in a definite program $\mathcal{P}$ is the set of goals which can be shown to follow from the program using SLD Resolution:*

$$\mathcal{M}_{\vdash}(\mathcal{P})(p) \ = \{p(t_1, \ldots, t_n) \in B_{\mathcal{P}} \mid \mathcal{P} \vdash p(t_1, \ldots, t_n)\}$$

Similarly to model-theoretic meaning from definition 2.1.18, the proof-theoretic meaning of a program $\mathcal{M}_{\vdash}(\mathcal{P})$ is defined as the union of proof-theoretic meanings of all its predicates.

The significant result proven by van Emden and Kowalski [120] is that, due to soundness and refutation-completeness of SLD Resolution, the model-theoretic meaning of a program $\mathcal{M}_{\models}(\mathcal{P})$ and the proof-theoretic meaning of a program $\mathcal{M}_{\vdash}(\mathcal{P})$ coincide:

$$\mathcal{M}_{\models}(\mathcal{P}) = \mathcal{M}_{\vdash}(\mathcal{P}) \tag{2.2}$$

In this section, we have given a fundamental glossary of Logic Programming. Starting from First-order logic, we described definite clause programs and their model-theoretic meaning via a fixpoint theorem.

As for the proof-theoretic meaning of definite clause programs, we described a first-order Unification and the SLD Resolution algorithms, and separately established the proof-theoretic meaning of definite clause programs by SLD Resolution, which coincides with the model-theoretic counterpart.

In the next section, we will describe COMBILOG, a variable-free, compositional relational programming language. The canonical form of COMBILOG predicate definitions is an aggregation of definite clauses, and as a result COMBILOG programs are a variable-free form of definite programs. Moreover, the semantics of COMBILOG programs is established in a similar way to definite clauses.

2.2 Combilog

COMBILOG is developed as a variable-free language that is semantically equivalent to definite clauses, particularly intended for program synthesis [52, 79]. The variable-free representation was necessary to avoid explicitly dealing with variables in the synthesis algorithm. Instead, COMBILOG uses a set of higher-order composition predicates for argument binding. Its combinatory form relates to Quine's *Predicate Functor Logic* presented in [98] and [99], which also has a variable-free form, and uses higher-order combinators such as *inv*, *ref*, *rotl*, or *exp* to compose predicates without consulting to variables.

COMBILOG is also strictly compositional, meaning the semantics of any expression is defined with regard to only its components, independent from the context, which is partially thanks to the lack of variables. The compositionality gives COMBILOG and the method of synthesis COMBINDUCE an edge by making it especially suitable for synthesizing recursive programs. Inductive Synthesis is the task of obtaining programs suggested by expected input/output examples [73]. Because COMBILOG operators construct programs in terms of other programs only, the input/output examples given for synthesis can be de-constructed to search for component programs by using the reverse semantics of its operators [46, 48, 49].

An application of COMBILOG is given in [11], where it is shown and demonstrated that COMBILOG is capable of being the target language for inductive synthesis, by composing the *atpos* predicate in COMBILOG with only one example. In [50], inductive synthesis of a COMBILOG program with two levels of recursion is shown. The duality between the recursion operators of COMBILOG, namely the fold-left and fold-right are discussed in [46].

In the following sections we will present the structure of COMBILOG programs and their semantics.

2.2.1 Compositional Relational Programming

A COMBILOG program consists of a set of predicate definitions. A predicate definition is in the form of a name assignment to a predicate expression. The left side (p) of the assignment is a predicate identifier, while the right side (φ_p) is a predicate expression:

$$p \leftarrow \varphi_p$$

The predicate expression consists of a hierarchy of predicate compositions. COMBILOG defines higher-level operators to combine existing predicate definitions and expressions into new ones. The operators COMBILOG provides are strictly referentially-transparent. The logic-relational meaning

of an expression is only determined by its components and the semantics of the operator. The context a predicate expression appears in has no effect on the meaning. This makes COMBILOG a strictly compositional programming language.

Every predicate expression has a list of arguments associated with it. The number of arguments in a predicate expression (or in a predicate definition) determines its *arity*. In order to compose predicate expressions, *elementary predicates* provide a basic set of primitive predicate definitions. *Logic operators* are the combinator operators which provide conjunction and disjunction. A *generalized projection operator* is used to modify the argument lists of predicate expressions. Both logic operators and the projection take predicate expressions as arguments and give a predicate expression as result. There are no explicit data type declarations, all arguments are implicitly of the single data type *term*.

We will begin by elementary predicates, and move on to combinator operators.

2.2.2 Elementary predicates

COMBILOG defines a set of elementary predicates in order to deal with fundamental concepts such as identity, constants, and lists. These are separate from user-defined predicates in that they are provided as a part of the language, as language constructs, or as a core library.

The elementary predicates are:

true is the logical truth predicate with no arguments.

'$const_c$' is a unary predicate which is used for introducing constants into the language. It has a parametric definition to give the constant desired. For example, $const_{[]}$ is a predicate expression that is satisfied only for the empty list symbol [].

id is the identity predicate, which has two arguments. It is only satisfied when both of its arguments take the same value.

cons is the list construction predicate, and has three arguments. It is satisfied only when the third argument is a term constructed with the list construction functor (|) applied with the values of the first and second arguments h and t, written as $[h|t]$. As well as lists, this predicate can be used for other data structures based on lists.

Here we give denotations of elementary predicates of COMBILOG. In the formulas below H refers to the Herbrand Universe. The extension of

a predicate p is usually written as $[\![p]\!]$. We will also show the arity of a predicate as a superscript, as in $[\![p]\!]^n$ for a predicate p with arity n.

$$\begin{aligned}
[\![\text{true}]\!]^0 &= \{\langle\rangle\} \\
[\![\text{const}_c]\!]^1 &= \{\langle c\rangle\} \text{ where } c \text{ is the parameter} \\
[\![\text{id}]\!]^2 &= \{\langle t, t\rangle \in H^2 \mid t \in H\} \\
[\![\text{cons}]\!]^3 &= \{\langle t', t'', [t'|t'']\rangle \in H^3 \mid t', t'' \in H\}
\end{aligned}$$

Next, we shall describe the generalized projection operator *make*.

2.2.3 Projection operator *make*

COMBILOG is a variable-free language, and uses a generalized projection operator as a way of manipulating the arguments of predicate expressions. This operator named *make* has two parameters: a list of indices $[\mu_1, \ldots, \mu_m]$ as its first parameter, and a source predicate expression p for its second parameter:

$$make([\mu_1, \ldots, \mu_m], p)$$

The index list determines the arguments in predicate expression produced as a result of *make*. When an index μ_i is in the range of argument positions of p, meaning if p has n arguments and $1 \leq \mu_i \leq n$, then that index refers to the corresponding argument of p. This results in i^{th} argument of the produced expression being bound to the value of the ${\mu_i}^{th}$ argument of p. If an index is larger than the arity of p, meaning $\mu_i > n$, then the i^{th} argument of the produced expression is unbound.

The denotation of the *make* operator is as follows, where H^m refers to the m^{th}-degree Cartesian product of H, the Herbrand universe of the program.

$$[\![make([\mu_1, \ldots, \mu_m], p)]\!]^m = \{\langle t_{\mu_1}, \ldots, t_{\mu_m}\rangle \in H^m \mid \exists\langle t_1, \ldots, t_n\rangle \in [\![p]\!]^n\}$$

This definition of *make* enables three essential operations it can be used for. When $m = n$, it is either an identity operation, as in $\langle\mu_1, \ldots, \mu_m\rangle = \langle 1, \ldots, m\rangle$, or it is a *reordering* of arguments. When $m < n$ it is a *cropping*, and when $m > n$, it is *expansion*. Naturally, the cropping and expansion may also reorder the arguments they pick from p, but they are distinct in the ways that cropping eliminates some arguments of p and expansion introduces some unbound ones.

Let us give some example uses of the *make* operator.

Defining the 1-ary universal relation uni_1, which has one argument which is satisfied by any value in the Herbrand universe of a program:

$$uni \leftarrow make([1], true)$$

Since the elementary predicate *true* has no arguments, this constitutes an *expansion*, and the only argument of *uni* is unbound.

Let us define a *head* predicate that has two arguments, the head of a list and the list itself. In PROLOG, it could be defined as follows, in relation to the elementary predicate *cons* ($cons(X, T, [X|T])$.):

$$head(L, X) \leftarrow cons(X, T, L)$$

while in COMBILOG it takes the following form:

$$head \leftarrow make([3, 1], cons)$$

which constitutes a *cropping*, binding the third argument of *cons* to the first argument of *head*, and the first argument if *cons* to the second argument of *head*. The second argument of *cons* is left unbound. The denotation of this instance of the *make* operation can be calculated as follows:

$$\begin{aligned}
[\![head]\!] &= [\![make[3, 1](cons)]\!] \\
&= \{\langle t_3, t_1\rangle \mid \exists t_2 \langle t_1, t_2, t_3\rangle \in [\![cons]\!]\} \\
&= \{\langle t_3, t_1\rangle \mid \exists t_2 (\langle t_1, t_2, t_3\rangle \in \{\langle x, y, [x|y]\rangle \mid x, y \in H\})\} \\
&= \{\langle [x|y], x\rangle \mid x, y \in H\}
\end{aligned}$$

It is important to note that the use of repeated indices $\mu_i = \mu_k$ $(i \neq k)$, such as:

$$make([1, 1], cons)$$

is not rejected by the semantics, but is not a practised use.

2.2.4 Logic operators

COMBILOG defines two logic operators. These are *and* for conjunction and *or* for disjunction. Their set-theoretic denotations using set intersection and set union are as follows:

$$\begin{aligned}
[\![and(\varphi, \psi)]\!]^n &= [\![\varphi]\!]^n \cap [\![\psi]\!]^n \\
[\![or(\varphi, \psi)]\!]^n &= [\![\varphi]\!]^n \cup [\![\psi]\!]^n
\end{aligned}$$

Which can also be written as follows, in a logical meta-language, with the use of higher-order operators *and* and *or*:

$$\begin{aligned} and(P,Q)(X_1,\ldots,X_n) &\leftarrow P(X_1,\ldots,X_n) \wedge Q(X_1,\ldots,X_n) \\ or(P,Q)(X_1,\ldots,X_n) &\leftarrow P(X_1,\ldots,X_n) \vee Q(X_1,\ldots,X_n) \end{aligned}$$

The logic operators only accept operands of the same arity. For this reason, they are mostly used together with the *make* operator, which can reorder, crop, or introduce arguments of the component predicates. By using *make*, the arguments intended to be bound to the same values are aligned in corresponding positions.

2.2.5 Sample predicate *append*

Let us give the implementation of the *append* predicate in COMBILOG as an example. It has three arguments, *L1*, *L2*, and *L*, all lists. The predicate is satisfied when the argument *L* contains the items of *L1* and *L2* joined together in the given order. Let us first look at the PROLOG implementation of the *append* predicate:

$$\begin{aligned} &append([],L,L). \\ &append([X|T],L2,[X|Lmid]) \leftarrow append(T,L2,Lmid). \end{aligned} \tag{2.3}$$

where the first line gives the base case, which determines that if the *L1* is empty, *L2* and *L* are identical. The second line scripts the recursive case. To compare this code better with COMBILOG, let us remove the syntactic sugar. This is done by using the list constructor notation such as $L = [X|T]$ with the *cons* predicate as $cons(X,T,L)$. The variable bindings within the head, such as *L* in $append([],L,L)$ are also replaced with the *id* predicate, as in $id(L,L)$. Similarly, the constant [] in the head is replaced by the parametric *const* predicate with the parameter [], which has only the constant [] itself in its extension. Both these changes are only syntactical, and do not change the semantics, due to the denotations of the elementary predicates used.

$$\begin{aligned} append(L1,L2,L) \leftarrow & const_{[]}(L1), id(L2,L). \\ append(L1,L2,L) \leftarrow & cons(X,T,L1), \\ & append(T,L2,Lmid), \\ & cons(X,Lmid,L). \end{aligned}$$

In order to translate the clauses above to a COMBILOG predicate definition, the atoms with variables are replaced by expressions which use the *make* operator. For example, the first clause, which we refer to as

the *base case*, there are three variables $L1$, $L2$, and L. Since all three variables also exist in the head, we write *make* operations that produce ternary predicate expressions for both $const_{[]}$ and *id*. For $const_{[]}$, since its first variable is bound to the value of the first variable in the head, the first index is 1. Since we need a ternary *make* operator, and since the second and third variables of the head are not bound to the value of an argument in $const_{[]}$, we place dummy indices $(2, 3)$ higher than the arity of $const_{[]}$. The resulting *make* expression becomes $make([1, 2, 3], const_{[]})$. Similarly, the *id* atom is translated to the expression $make([3, 1, 2], id)$, where the first index 3 is higher than the arity of *id* since the first argument in the head ($L1$) is not bound to the value of an argument in *id*. The conjunction of $const_{[]}$ and *id* is written as an *and* operation that covers the first clause: $and(make([1, 2, 3], const_{[]}), make([3, 1, 2], id))$. The same principles are applied to the second clause, referred to as the *recursive case*, while composing the COMBILOG code below. The difference is that only 3 of the variables are bound to variables in the head, so a separate *make* operation is needed to crop only the first three of the total six arguments. This is not necessary for the first operand of *or*, since there are only three arguments, all bound to the head (*append*), in the same order, and there are no arguments that need to be cropped.

$$
\begin{aligned}
append \leftarrow or(&and(make([1,2,3], const_{[]}),\\
&\quad make([3,1,2], id)),\\
&make([1,2,3],\\
&\quad and(make([3,4,5,1,2,6], cons),\\
&\qquad make([4,2,5,6,1,3], append),\\
&\qquad make([4,5,3,1,6,2], cons))))
\end{aligned}
\tag{2.4}
$$

If we inspect the projection in the last line, $make([4, 5, 3, 1, 6, 2], cons)$, we observe that it produces a predicate expression with six arguments, where the 3rd argument is bound to a list (through the 3rd argument in *cons*), the 4th argument bound to its head (through the 1st argument in *cons*), and the 6thth argument bound to its tail (through the 2nd argument in *cons*). All other arguments are unbound. If we consider the second *and* expression (line 4..6), only the first three arguments are projected, due to the containing $make([1, 2, 3], \ldots)$ operation. The values of other arguments of *and* (4th, 5th, and 6th) are only accessible within this expression, but not to the containing context, the *or* operator.

The COMBILOG predicate definition above is translated from the PROLOG one given earlier, through trivial transformation steps. The both definitions are semantically equivalent, as it will be proven later in Section 2.2.8.

2.2.6 Recursion operators

List recursion operators have been defined for COMBILOG in the earlier work [47]. The main operators are *foldr* and *foldl*, and they function similarly to their counterparts in functional programming, but here they must be implemented in a compositional relational context, since COMBILOG context cannot contain variables. Both operators take two predicate arguments, P for recursive case, and Q for the base case. This is in line with the ordinary way of writing recursive predicates in definite clause programs, as seen in the PROLOG *append* implementation in Definition 2.3 from the previous section.

Before giving their set-theoretic denotations, let us look at the definitions if *foldr* and *foldl* in meta-logic language, with a description following:

$$\begin{aligned} \mathit{foldr}(P,Q)(Y,[],Z) &\leftarrow Q(Y,Z) \\ \mathit{foldr}(P,Q)(Y,[X|T],W) &\leftarrow \mathit{foldr}(P,Q)(Y,T,Z) \wedge P(X,Z,W) \end{aligned}$$

$$\begin{aligned} \mathit{foldl}(P,Q)(Y,[],Z) &\leftarrow Q(Y,Z) \\ \mathit{foldl}(P,Q)(Y,[X|T],W) &\leftarrow P(X,Y,Z) \wedge \mathit{foldl}(P,Q)(Z,T,W) \end{aligned}$$

Since the empty list constant ([]) always exists in the base case, it is kept as a constant, embedded in the definition, and the base case is given as a single predicate parameter Q. Similarly, list packing/unpacking literals are embedded in the definitions of the folds, reducing the recursive case to a single ternary predicate parameter P. The non-predicate parameters are as follows. The first argument (Y) carries the initial element, the second argument carries the list to be iterated through, and the third (Z or W) argument is the result of folding.

The *append* predicate from Definition 2.4 from the previous section written using the *foldr* operator, in fold-extended COMBILOG is as follows:

$$\mathit{append} \leftarrow \mathit{make}[2,1,3](\mathit{foldr}(\mathit{cons}, \mathit{id}))$$

where the *make* operator encompassing the *foldr* is used to flip the positions of the first two arguments, since the definition of the *foldr*, the second argument is the recursion parameter, while in *append* it is the first one. This final definition that uses the *foldr* operator is semantically equivalent to the COMBILOG code for the *append* predicate earlier, and also to the PROLOG implementation of *append* as well. Yet as seen above, it is devoid of variables.

These operators provide a declarative form of single recursion schemes, and contribute to expressiveness of COMBILOG programs significantly. Fi-

nally, let us look at their set-theoretic denotations:

$$[\![foldr(\varphi,\psi)]\!]^3 = \bigcup_{i=0}^{\infty} [\![foldr_i(\varphi,\psi)]\!]^3$$

where

$$[\![foldr_0(\varphi,\psi)]\!]^3 = \{\langle y,[],z\rangle \in H^3 \mid \langle y,z\rangle \in [\![\psi]\!]^2\}$$

$$\left[\!\left[foldr_{i+1}(\varphi,\psi)\right]\!\right]^3 = \{\langle y,[t_1|t_2],w\rangle \in H^3 \mid \exists z \in H \text{ s.t.} \\ \langle y,t_2,z\rangle \in [\![foldr_i(\varphi,\psi)]\!]^3 \wedge \langle t_1,z,w\rangle \in [\![\varphi]\!]^3$$

and similarly for *foldl*:

$$[\![foldl(\varphi,\psi)]\!]^3 = \bigcup_{i=0}^{\infty} [\![foldr_i(\varphi,\psi)]\!]^3$$

where

$$[\![foldl_0(\varphi,\psi)]\!]^3 = \{\langle y,[],z\rangle \in H^3 \mid \langle y,z\rangle \in [\![\psi]\!]^2\}$$

$$\left[\!\left[foldl_{i+1}(\varphi,\psi)\right]\!\right]^3 = \{\langle y,[t_1|t_2],w\rangle \in H^3 \mid \exists z \in H \text{s.t.} \\ \langle t_1,y,z\rangle \in [\![\varphi]\!]^3 \wedge \langle z,t_2,w\rangle \in [\![foldl_i(\varphi,\psi)]\!]^3$$

The recursion operators conclude the denotational semantics. In the following sections, we will observe the equivalence between COMBILOG and PROLOG programs, and finally describe the execution model of COMBILOG programs.

2.2.7 Transformation from definite clauses

In this section, following and partially quoting from [52], we describe the transformation $comb(\mathcal{P})$, which produces a COMBILOG program from a given ordinary logic program $\mathcal{P}$.

Consider an ordinary logic program $\mathcal{P}$, containing one or more definite clauses for every predicate p defined in the program. The transformations would produce a COMBILOG form of the definition of predicate p as:

$$p \leftarrow \varphi_p$$

where φ_p is a ground predicate term composed solely of the operators *and*, *or* and *make*.

Definition 2.2.1. *Let $\mathcal{P}$ be an ordinary logic program. Let $comb(\mathcal{P})$ be the* COMBILOG *program resulting from the transformation stages: varterm, which eliminates non-variable terms, and combintro, which introduces combinators, with*

$$comb(\mathcal{P}) \;=\; combintro(varterm(\mathcal{P}))$$

Elimination of Non-variable Terms (varterm)
The **varterm** stage involves normalizing the arguments of a definite clause so that it does not include any constants, list terms, or multiple occurrences of a variable. It is the first step towards COMBILOG's variable-free form.

The transformation steps are given as a set of rules in the form

$$\begin{array}{c} Head \leftarrow Body \\ \Downarrow \\ Head' \leftarrow Body' \end{array}$$

each transforming one aspect of the clause in the head or in the body. Here we recite the steps of **varterm**, quoting from [52]:

The list constructs such as $l(H,T)$ in the head are replaced with *cons* atoms in the body:

$$\begin{array}{c} p(t_1,\ldots,t_{i-1},l(H,T),t_{i+1},\ldots,t_n) \leftarrow Body \\ \Downarrow \\ p(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n) \leftarrow cons(H,T,X), Body \end{array} \tag{2.5}$$

The list constructs such as $l(H,T)$ in the body are also replaced with *cons* atoms:

$$\begin{array}{l} Head \leftarrow \ldots, q(t_1,\ldots,t_{i-1},l(H,T),t_{i+1},\ldots,t_n),\ldots \\ \quad \Downarrow \\ Head \leftarrow \ldots, q(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n), cons(H,T,X),\ldots \end{array} \tag{2.6}$$

Every constant C in the head is replaced with a $const_C$ atom in the body.

$$\begin{array}{c} p(t_1,\ldots,t_{i-1},C,t_{i+1},\ldots,t_n) \leftarrow Body \\ \Downarrow \\ p(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n) \leftarrow const_C(X), Body \end{array} \tag{2.7}$$

Every constant C in the body is also replaced with a $const_C$ atom in the body:

$$\begin{array}{l} Head \leftarrow \ldots, q(t_1,\ldots,t_{i-1},C,t_{i+1},\ldots,t_n),\ldots \\ \quad \Downarrow \\ Head \leftarrow \ldots, q(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_n), const_C(X),\ldots \end{array} \tag{2.8}$$

Multiple occurrences of a variable in the head are replaced with distinct variables, and an *id* atom in the body that binds the distinct variables:

$$\begin{array}{l} p(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_{i+(j-1)},X,t_{i+(j+1)},\ldots,t_n) \leftarrow \mathit{Body} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Downarrow \\ p(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_{i+(j-1)},Y,t_{i+(j+1)},\ldots,t_n) \leftarrow \mathit{id}(X,Y),\mathit{Body} \end{array} \tag{2.9}$$

Multiple occurances of a variable in the body are also replaced with distinct variables, and an *id* atom that binds these distinct variables:

$$\begin{array}{l} \mathit{Head} \leftarrow \ldots,q(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_{i+(j-1)},X,t_{i+(j+1)},\ldots,t_n),\ldots \\ \qquad \Downarrow \\ \mathit{Head} \leftarrow \ldots,q(t_1,\ldots,t_{i-1},X,t_{i+1},\ldots,t_{i+(j-1)},Y,t_{i+(j+1)},\ldots,t_n), \\ \qquad\quad \mathit{id}(X,Y),\ldots \end{array} \tag{2.10}$$

The definitions of auxiliary predicates introduced in **varterm**, such as *id* are given as elementary predicates in Section 2.2.2. This finalizes the steps of transformation stage **varterm**. The final form containing all defining clauses of a predicate p can be written as follows:

$$p(X_{\sigma_{i1}},\ldots,X_{\sigma_{in}}) \leftarrow \bigvee_{i=1}^{\gamma} \bigwedge_{j=1}^{\beta_i} q_{ij}(X_{ij1},\ldots,X_{ij\mathit{arity}(q_{ij})}) \tag{2.11}$$

where γ is the number of defining clauses for p and β_i is the number of atoms in the body of the ith clause. The variables $X_{\sigma_{i1}},\ldots,X_{\sigma_{in}}$ are distinct.

In order to show that the varterm transformation stage preserves meaning, we quote the theorem of equality from [52]:

Theorem 2.2.1. *For every predicate p in $\mathcal{P}$*

$$\mathcal{M}(\mathbf{varterm}(\mathcal{P}))(p) \;=\; \mathcal{M}(\mathcal{P})(p)$$

Proof: Consider the transformations 2.5 to 2.10. By unfolding the right hand side of 2.5 to 2.10 with the unit clauses for predefined predicates, the clauses of the left hand side are regained. By the theorem of Tamaki and Sato [112], see also [91], every transformation sequence constructed by unfolding is totally correct with respect to the least Herbrand semantics. □

Introduction of Combinator Terms

The transformation stage **combintro** defines steps to introduce COMBILOG's generalized projection operator *make*, and also replaces ordinary

logic operators with their COMBILOG counterparts. This stage starts from the form that the **varterm** stage ends with, and by the end the variable terms in the clause are made redundant by the *make* operator, rendering it *variable-free*.

We shall first define a few fundamental concepts necessary for writing clauses in COMBILOG form. The n-ary logic operators, which recursively convert n-ary operand lists to their equivalent binary counterparts is given below.

Definition 2.2.2. *Let* $\prod$ *and* $\sum$ *denote, respectively, aggregation of the logic operators 'and' and 'or' in* COMBILOG*:*

$$\begin{aligned}
\prod_{i=1}^{1} e_i &= e_1 \\
\prod_{i=1}^{n} e_i &= and(\prod_{i=1}^{n-1} e_i, e_n) \\
\sum_{i=1}^{1} e_i &= e_1 \\
\sum_{i=1}^{n} e_i &= or(\sum_{i=1}^{n-1} e_i, e_n)
\end{aligned}$$

The next definitions are the index functions π_{ijk} and σ_{ik}. These are used for producing the index lists which are the first operands of the *make* operator, according to the variable bindings existing in the clause and the rules defining these functions. π_{ijk} in particular involves the *make* for the j^{th} atom of the i^{th} clause, while σ_{ik} involves the *make* for the i^{th} clause itself.

Definition 2.2.3. *Index function* π_{ijk}*. Consider the i'th clause: If the variable with index jk where* $k \in \{1, \ldots, \#Vars_i\}$ *is found in some position n in the atom* $q_{ij}(X_{ij1}, \ldots, X_{ijarity(q_{ij})})$ *then* $\pi_{ijk} =_{def} n$*; otherwise* $\pi_{ijk} =_{def} m$*, where m is an integer greater than* $arity(q_{ij})$*, and it is different from other indices that are not found in the atom.*

Definition 2.2.4. *Index function* σ_{ik}*. Consider the i clauses for p, and* $k \in \{1, \ldots, arity(p)\}$*, then* $\sigma_{ik} =_{def} k$*.*

Note that the σ_{ik} function produces the same instance of the *make* operator for each defining clause of a predicate p. This is the case because definite clauses for the same predicate always have the same number of variables in the head, and index modifications such as reordering or trun-

cating are not necessary as it was with the π_{ijk} function.

We shall now give the steps in rewriting stage **combintro**, starting from the last form in **varterm** and finishing with the canonical COMBILOG form.

The last form left by the **varterm** steps is:

$$p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) \leftarrow \bigvee_{i=1}^{\gamma} \bigwedge_{j=1}^{\beta_i} q_{ij}(X_{ij1}, \ldots, X_{ij\,arity(q_{ij})})$$

The first step introduces *make* operators around every literal q_{ij}, which projects every literal in the body of i^{th} clause to same arity, and produces predicate expressions to which logic operator *and* can be applied.

$$\begin{aligned} p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) &\leftarrow \bigvee_{i=1}^{\gamma} \bigwedge_{j=1}^{\beta_i} q_{ij}(X_{ij1}, \ldots, X_{ij\,arity(q_{ij})}) \\ &\Downarrow \text{step 1, inner } \textit{make}\text{s} \\ p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) &\leftarrow \\ &\bigvee_{i=1}^{\gamma} \bigwedge_{j=1}^{\beta_i} \mathit{make}([\pi_{ij1}, \ldots, \pi_{ij\#\,Vars_i}], q_{ij})(X_{\pi_{ij1}}, \ldots, X_{\pi_{ij\#\,Vars_i}}) \end{aligned}$$

the second step replaces the aggregate conjunction operator $\bigwedge$ with the corresponding aggregated form of the *and* operator, namely $\prod$.

$$\begin{aligned} &\Downarrow \text{step 2, } \textit{and} \text{ operators} \\ p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) &\leftarrow \\ &\bigvee_{i=1}^{\gamma} \prod_{j=1}^{\beta_i} \mathit{make}([\pi_{ij1}, \ldots, \pi_{ij\#\,Vars_i}], q_{ij})(X_{\pi_{ij1}}, \ldots, X_{\pi_{ij\#\,Vars_i}}) \end{aligned}$$

the third step produces *make* operators around every i^{th} defining body for predicate p, practically in the identity form $\mathit{make}([1, \ldots, n], p)$, where $n = \mathit{arity}(p)$.

$$\begin{aligned} &\Downarrow \text{step 3, outer } \textit{make}\text{s} \\ p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) &\leftarrow \\ &\bigvee_{i=1}^{\gamma} \mathit{make}([\sigma_{i1}, \ldots, \sigma_{in}], \\ &\qquad \prod_{j=1}^{\beta_i} \mathit{make}([\pi_{ij1}, \ldots, \pi_{ij\#\,Vars_i}], q_{ij}))(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) \end{aligned}$$

in the final step, the aggregate disjunction operator $\bigvee$ is replaced with the corresponding aggregate operator $\sum$ for *or*, revealing the canonical form of COMBILOG clauses.

$$\Downarrow \text{ step 4, } or \text{ operator}$$

$$p(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}}) \leftarrow \sum_{i=1}^{\gamma} make([\sigma_{i1}, \ldots, \sigma_{in}], \prod_{j=1}^{\beta_i} make([\pi_{ij1}, \ldots, \pi_{ij\#Vars_i}], q_{ij}))(X_{\sigma_{i1}}, \ldots, X_{\sigma_{in}})$$

The resulting COMBILOG clause defining the predicate p conforms thus with a canonical "*or-make-and-make*" form. In the transformation steps above the variables are used for expressing how the indices in the *make* operators establish argument binding between the component predicates and the head. The standard variable-free COMBILOG form is as follows:

Definition 2.2.5. *A canonical combinator expression which constitutes a distinguished combinatory clause normal form, has the following structure:*

$$p \leftarrow \sum_{i=1}^{\gamma} make([\sigma_{i1}, \ldots, \sigma_{in}], \prod_{j=1}^{\beta_i} make([\pi_{ij1}, \ldots, \pi_{ij\#_i}], q_{ij}))$$

In this section we gave the transformation from ordinary logic programs to COMBILOG programs, which consisted mainly of eliminating the non-variable terms such as constants or list constructs, replacing bound variables with *make* operators, and replacing the logic operators with their counterparts in COMBILOG. In the next section we observe the semantic equivalence between these two sorts of programs.

2.2.8 Correspondence with definite clause programs

In their work on semantics of programming with predicate logic [120], van Emden and Kowalski show that the unique least fixpoint of a program coincides with the model-theoretic meaning of a program. Restating from Section 2.1, Equation 2.1, and Equation 2.2, which refers to the coincidence of the model-theoretic meaning and the proof-theoretic meaning:

$$\mathcal{M}_{\models}(\mathcal{P}) = T_{\mathcal{P}} \uparrow \omega$$
$$\mathcal{M}_{\models}(\mathcal{P}) = \mathcal{M}_{\vdash}(\mathcal{P})$$

It follows that by showing the least fixpoints of two programs to be equal, we can prove that their model-theoretic meanings are equal. In this section we shall give the theorem of coincidence between model-theoretic semantics of COMBILOG and ordinary definite clause programs. We will begin by some fundamental definitions necessary to state our theorem.

The definition of a fixpoint of a logic program depends on an immediate consequence operator, which maps a Herbrand interpretation I to I', which monotonously expands with the immediately entailed ground instances, yielding a fixpoint when applied repeatedly until it maps I to itself. Here let us restate the immediate consequence operator from Definition 2.1.20 and the power function from Definition 2.1.21:

For an ordinary logic program $\mathcal{P}$, the immediate consequence operator $\mathbf{T}_{\mathcal{P}}^{ord} : I \to I$ maps a Herbrand interpretation I into I':

$$\mathbf{T}_{\mathcal{P}}^{ord}(I) = \{A_0 \mid A_0 \leftarrow A_1, \ldots, A_n \in \Gamma(\mathcal{P}) \wedge \{A_1, \ldots, A_n\} \subseteq I\}$$

where $\Gamma(\mathcal{P})$ is the set of ground instances of the clauses in $\mathcal{P}$.

The power function $\mathbf{T}_{\mathcal{P}}^{ord} \uparrow : \mathbb{N} \to 2^B$ is given as:

$$\mathbf{T}_{\mathcal{P}}^{ord} \uparrow 0 = \emptyset$$
$$\mathbf{T}_{\mathcal{P}}^{ord} \uparrow (i+1) = \mathbf{T}_{\mathcal{P}}^{ord}(\mathbf{T}_{P}^{ord} \uparrow i)$$
$$\mathbf{T}_{\mathcal{P}}^{ord} \uparrow \omega = \bigcup_{i=0}^{\infty} \mathbf{T}_{\mathcal{P}}^{ord} \uparrow i$$

where the final definition $\mathbf{T}_{\mathcal{P}}^{ord} \uparrow \omega$ is the result of infinitely applying the operator, hence arriving at the least fixpoint for the program P.

Intuitively, the power function starts with the empty set for $i = 0$. Immediately at $i = 1$, it includes ground instances of facts in the program. At $i = 2$, it expands to include the ground instances that directly follow from the facts at $i = 1$, and every step $i + 1$, produces new ground instances following from i, until there are no new consequences.

Before we move on to fixpoint definitions for COMBILOG programs, it is necessary to define some fundamental concepts of program denotation in first-order logic:

Definition 2.2.6. *Let an extension of an n-ary predicate p be any n-ary relation over H assigned to p, that is, a subset of the n-ary tuples forming the product H^n. The extension of a predicate p, denoted as $[\![p]\!]$:*

$$[\![p]\!] = \{\langle t_1, \ldots, t_n\rangle \in H_{\mathcal{P}}^n \mid p(t_1, \ldots, t_n) \in \mathcal{M}_{\models}(\mathcal{P})(p)\}$$

where the model-theoretic meaning of a predicate p in program $\mathcal{P}$ is given as $\mathcal{M}_{\models}(\mathcal{P})(p)$, restated from the earlier Definition 2.1.17:

$$\mathcal{M}_{\models}(\mathcal{P})(p) \;=\; \{p(t_1, \ldots, t_n) \in B_{\mathcal{P}} \mid \mathcal{P} \models p(t_1, \ldots, t_n)\}$$

which enables us to define extension maps as a mapping from predicate symbols to extensions:

Definition 2.2.7. *An extension map E of a program $\mathcal{P}_{comb}$ maps program predicates $p_{i(n)}$, with arity n, into corresponding extensions e_i, (where $e_i \subseteq H^n_{\mathcal{P}}$)*

$$E = \bigcup_{i=1}^{m} \{p_{i(n)} \mapsto e_i\}$$

Let E be an extension map and p a program predicate. Then

$$E(p) = e \iff (p \mapsto e) \in E$$

and as necessary concept, the lookup for the extension of a predicate symbol with regard to an extension map:

Definition 2.2.8. *For a program predicate p the extension is obtained by a lookup in the extension map:*

$$[\![p]\!]_E = E(p)$$

After defining predicate extensions, extension maps, and the extension lookup operation, we can move on to denotations of COMBILOG programs. In this section, we introduce the semantics of COMBILOG programs via the least fixpoint of a novel immediate consequence operator $\mathbf{T}^{comb}_{\mathcal{P}_{comb}}$ dedicated to combinator logic programs [52].

Definition 2.2.9. *The immediate consequence operator for a* COMBILOG *program $\mathcal{P}_{comb}$ is defined below, which maps extension maps into extension maps:*

$$\mathbf{T}^{comb}_{\mathcal{P}_{comb}}(E) \;=\; \bigcup_{i}^{m} \{p_i \mapsto [\![\varphi_i]\!]_E\}$$

and hence

$$\mathbf{T}^{comb}_{\mathcal{P}_{comb}}(E)(p) = [\![\varphi]\!]_E$$

The operator $\mathbf{T}^{comb}_{\mathcal{P}_{comb}}$ is a counterpart to the operator $\mathbf{T}^{ord}_{\mathcal{P}}$ for ordinary logic programs $\mathcal{P}$ stated earlier, but does not call for the generation of the ground instances of the program clauses.

The extension of a predicate term φ with regard to an extension map is calculated according to the following definition:

Definition 2.2.10. *The extension of a compound predicate term φ with regard to an extension map E, namely $[\![\varphi]\!]_E$ is obtained according to the compositional denotations of the operators and by looking up the extensions of the operands from the extension map E, as follows:*

$$
\begin{aligned}
[\![and(\varphi,\psi)]\!]_E &= [\![\varphi]\!]_E \cap [\![\psi]\!]_E \\
\left[\!\!\left[\prod_i \varphi\right]\!\!\right]_E &= \bigcap_i [\![\varphi]\!]_E \\
[\![or(\varphi,\psi)]\!]_E &= [\![\varphi]\!]_E \cup [\![\psi]\!]_E \\
\left[\!\!\left[\sum_i \varphi\right]\!\!\right]_E &= \bigcup_i [\![\varphi]\!]_E \\
[\![make[\mu_1,\ldots,\mu_m](\phi)]\!]_E &= \{\langle t_{\mu_1},\ldots,t_{\mu_m}\rangle \in H^m \mid \exists\langle t_1,\ldots,t_n\rangle \in [\![\phi]\!]_E\}
\end{aligned}
$$

or if we define more explicitly for make:

$$
\begin{aligned}
&[\![make[\mu_1,\ldots,\mu_m](\phi)]\!]_E = \{\langle t'_1,\ldots,t'_m\rangle \in H^m \mid \\
&\quad \exists\langle t_1,\ldots,t_n\rangle \in [\![\phi]\!]_E \ \textit{such that for } 1 \le i \le m \textit{ if } \mu_i \le n \textit{ then } t'_i = t_{\mu_i}\}
\end{aligned}
$$

Analogously to that of $\mathbf{T}^{ord}_{\mathcal{P}}$, the power function of $\mathbf{T}^{comb}_{\mathcal{P}_{comb}}$ is defined similarly:

Definition 2.2.11. *The powers $\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow : \mathbb{N} \to E$ for the* COMBILOG *operator are constructed as iterated mappings of extension maps into extension maps defined as follows, where $\mathcal{P}_{comb}$ refers to a* COMBILOG *program with m predicate definitions.*

$$
\begin{aligned}
\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow 0 &= \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\} \\
\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow (i+1) &= \mathbf{T}^{comb}_{\mathcal{P}_{comb}}(\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow i) \\
\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow \omega &= \bigcup_{j=1}^{m} \{p_j \mapsto \bigcup_{i=0}^{\infty} ((\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow i)(p_j))\}
\end{aligned}
$$

Note that in the last equation above, the intention is to look up the extension mapped to a p_j for every iteration, and then to take their union. This is slightly more involved than the counterpart for definite clauses, where taking the union of every step is sufficient.

Definition 2.2.12. *Extension maps e_i can be defined in terms of Herbrand interpretations I and vice versa by:*

$$p(t_1,\ldots,t_n) \in I \iff \langle t_1,\ldots,t_n\rangle \in E(p).$$

These mutual definitions give rise to two functions ext and int, being each others inverse such that

$$ext(I) = E$$
$$int(E) = I$$

The connection between the immediate consequence operator $\mathbf{T}^{ord}$ and the immediate extension operator $\mathbf{T}^{comb}$ is manifested by Theorem 2.2.2 [51].

Theorem 2.2.2. *The least fixpoint of an ordinary logic program $\mathcal{P}$ is equal to the least fixpoint of the corresponding* COMBILOG *program* $\mathbf{comb}(\mathcal{P})$.

$$\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow \omega = ext\left(\mathbf{T}^{ord}_{\mathcal{P}} \uparrow \omega\right)$$

Proof of Theorem 2.2.2: Since both power functions $\mathbf{T}^{ord}_{\mathcal{P}} \uparrow i$ and $\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i$ are defined inductively, we shall prove their equality by induction. First we show that they are equal at $i = 0$, and also that assuming they are equal at any step i, they will be equal at the following step $i + 1$.

Induction Base:

$$\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow 0 = ext\left(\mathbf{T}^{ord}_{\mathcal{P}} \uparrow 0\right)$$

Proof of induction base: Beginning with the base case of the power function from Definition 2.2.11:

$$\begin{aligned}
\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow 0 &= \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\} \\
&\textit{by Definitions in 2.2.12:} \\
&= ext(int(\bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\})) \\
&= ext(\emptyset) \\
&\textit{by Definition 2.1.21:} \\
&= ext\left(\mathbf{T}^{ord}_{\mathcal{P}} \uparrow 0\right)
\end{aligned}$$

Note that,

$$int(\bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\}) = \emptyset$$

since if $E = \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\}$, for each j $(1 \leq j \leq m)$, the lookup from the extension map, $E(p_j) = \emptyset$. Which entails, through Definition 2.2.12, the union of individual extensions to be empty, hence $I = \emptyset$.

Induction Hypothesis:

$$\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i = ext\left(\mathbf{T}^{ord}_{\mathcal{P}} \uparrow i\right)$$

Induction Step:

$$\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow (i+1) = ext\left(\mathbf{T}^{ord}_{\mathcal{P}} \uparrow (i+1)\right)$$

Proof of induction: Let us show the equality in two directions, $\Rightarrow$ and $\Leftarrow$, separately.
For $\Rightarrow$ assume a tuple $\langle t_1, \ldots, t_n \rangle$, such that

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow (i+1)(p_j)$$

then either (i) or (ii):

(i) The tuple belongs directly to the preceding step:

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i\ (p_j)$$

by the induction hypothesis,

$$p_j(t_1, \ldots, t_n) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow i$$

and by monotonicity,

$$p_j(t_1, \ldots, t_n) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow (i+1)$$

(ii) The tuple is an immediate consequence of a predicate body in the preceding step. Assuming

$$(p_j \mapsto [\![\varphi]\!]) \in \mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i$$

and

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}^{comb}_{\mathcal{P}_{comb}}(\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow i)(p_j)$$

then the tuple must be in the extension of predicate body calculated with regard to the extension map in the preceding step:

$$\langle t_1, \ldots, t_n \rangle \in [\![\varphi]\!]_{\mathbf{T}^{comb}_{\mathcal{P}_{comb}} \uparrow i}$$

then by Lemma 2.2.1:

$$p_j(t_1, \ldots, t_n) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow (i+1)$$

For $\Leftarrow$ assume a tuple $\langle t_1, \ldots, t_n \rangle$, such that

$$p_j(t_1, \ldots, t_n) \in \mathbf{T}_{\mathcal{P}}^{ord} \uparrow (i+1)$$

then either (i) or (ii):

(i) The tuple belongs directly to the preceding step:

$$p_j(t_1, \ldots, t_n) \in \mathbf{T}_{\mathcal{P}}^{ord} \uparrow i$$

and by the induction hypothesis,

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}_{\mathbf{comb}(\mathcal{P})}^{comb} \uparrow i \ (p_j)$$

and by monotonicity:

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}_{\mathbf{comb}(\mathcal{P})}^{comb} \uparrow (i+1)(p_j)$$

(ii) There is the following ground clause in $\Gamma(\mathcal{P})$:

$$p_j(t_1, \ldots, t_n) \leftarrow \\ q_{\iota_1}(t_{1_1}, \ldots, t_{1_{arity(q_{\iota_1})}}), \ldots, q_{\iota_{\beta_\iota}}(t_{\beta_{\iota 1}}, \ldots, t_{\beta_\iota arity(q_{\iota_{\beta_\iota}})}) \in \Gamma(\mathcal{P})$$

and for $1 \leq \kappa \leq \beta_\iota$,

$$q_{\iota_\kappa}(t_{\kappa_1}, \ldots, t_{\kappa_{arity(q_{\iota_\kappa})}}) \in \mathbf{T}_{\mathcal{P}}^{ord} \uparrow i$$

or equivalently,

$$\langle t_{\kappa_1}, \ldots, t_{\kappa_{arity(q_{\iota_\kappa})}} \rangle \in [\![q_{\iota_\kappa}]\!]_{ext(\mathbf{T}_{\mathcal{P}}^{ord} \uparrow i)}$$

then by Lemma 2.2.1,

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}_{\mathbf{comb}(\mathcal{P})}^{comb} \uparrow (i+1)(p_j).$$

□

Lemma 2.2.1. *Assume a tuple t such that*

$$\langle t_1, \ldots, t_n \rangle \notin \mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i(p_j)$$

and thus by the induction hypothesis,

$$p_j(t_1, \ldots, t_n) \notin \mathbf{T}^{ord}_{\mathcal{P}} \uparrow i$$

then the following holds:

$$\langle t_1, \ldots, t_n \rangle \in \mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow (i+1)(p_j) \iff p_j(t_1, \ldots, t_n) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow (i+1)$$

Proof of Lemma 2.2.1. Consider the following aggregate form of an ordinary predicate definition:

$$p_j(X_{\sigma_{\iota_1}}, \ldots, X_{\sigma_{\iota n}}) \leftarrow \bigvee_{\iota=1}^{\gamma} \bigwedge_{\kappa=1}^{\beta_\iota} q_{\iota\kappa}(X_{\iota_{\kappa_1}}, \ldots, X_{\iota_{\kappa_{arity(q_{\iota\kappa})}}})$$

corresponding to the canonical form of a COMBILOG predicate definition:

$$p_j \leftarrow \sum_{\iota=1}^{\gamma} make[\sigma_{\iota_1}, \ldots, \sigma_{\iota n}](\prod_{\kappa=1}^{\beta_\iota} make[\pi_{\iota_{\kappa_1}}, \ldots, \pi_{\iota_{\kappa_{\#Vars_\iota}}}](q_{\iota_\kappa}))$$

A tuple $\langle t_{\sigma_{\iota_1}}, \ldots, t_{\sigma_{\iota n}} \rangle$ is found in the calculation of a predicate body with regard to the extension map in the preceding step,

$$\langle t_{\sigma_{\iota_1}}, \ldots, t_{\sigma_{\iota n}} \rangle \in \left[\!\!\left[\sum_{\iota=1}^{\gamma} make[\sigma_{\iota_1}, \ldots, \sigma_{\iota n}](\prod_{\kappa=1}^{\beta_\iota} make[\pi_{\iota_{\kappa_1}}, \ldots, \pi_{\iota_{\kappa_{\#Vars_\iota}}}](q_{\iota_\kappa})) \right]\!\!\right]_{\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i}$$

if and only if it is the result of, κ^{th} conjunction of the ι^{th} disjunction, for a fixed ι where $1 \leq \kappa \leq \beta_\iota$:

$$\langle t_{\pi_{\iota_{\kappa_1}}}, \ldots, t_{\pi_{\iota_{\kappa_{\#Vars_\iota}}}} \rangle \in \left[\!\!\left[make[\pi_{\iota_{\kappa_1}}, \ldots, \pi_{\iota_{\kappa_{\#Vars_\iota}}}](q_{\iota_\kappa}) \right]\!\!\right]_{\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i}$$

which, considering the first rewriting rule from the previous section:

$$p(X_{\sigma_{\iota_1}}, \ldots, X_{\sigma_{\iota n}}) \leftarrow \bigvee_{\iota=1}^{\gamma} \bigwedge_{\kappa=1}^{\beta_\iota} q_{\iota\kappa}(X_{\iota_{\kappa_1}}, \ldots, X_{\iota_{\kappa_{arity(q_{\iota\kappa})}}})$$

$$\Downarrow \text{step 1, inner } \textit{make}\text{s}$$

$$p(X_{\sigma_{\iota_1}}, \ldots, X_{\sigma_{\iota n}}) \leftarrow \bigvee_{\iota=1}^{\gamma} \bigwedge_{\kappa=1}^{\beta_\iota} make[\pi_{\iota_{\kappa_1}}, \ldots, \pi_{\iota_{\kappa_{\#Vars_\iota}}}](q_{\iota\kappa})(X_{\pi_{\iota_{\kappa_1}}}, \ldots, X_{\pi_{\iota_{\kappa_{\#Vars_\iota}}}})$$

holds if and only if, the unmodified (pre-make) tuple $\langle t_{\iota_{\kappa_1}}, \ldots, t_{\iota_{\kappa_{arity(q_{\iota_\kappa})}}} \rangle$ (for $1 \leq \kappa \leq \beta_\iota$) is in the extension of the literal q_{ι_κ} with regard to the extension map in the preceding step:

$$\langle t_{\iota_{\kappa_1}}, \ldots, t_{\iota_{\kappa_{arity(q_{\iota_\kappa})}}} \rangle \in [\![q_{\iota_\kappa}]\!]_{\mathbf{T}^{comb}_{\mathbf{comb}(\mathcal{P})} \uparrow i}$$

equivalently, by the induction hypothesis, if and only if

$$q_{\iota_\kappa}(t_{\iota_{\kappa_1}}, \ldots, t_{\iota_{\kappa_{arity(q_{\iota_\kappa})}}}) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow i$$

which considering the conjunctive clause

$$p_j(t_{\sigma_{\iota_1}}, \ldots, t_{\sigma_{\iota_n}}) \leftarrow \bigwedge_{\kappa=1}^{\beta_\iota} q_{\iota_\kappa}(t_{\iota_{\kappa_1}}, \ldots, t_{\iota_{\kappa_{arity(q_{\iota_\kappa})}}}) \in \Gamma(\mathcal{P})$$

holds if and only if the modified (post-make) tuple, as a ground atom of p_j, exists in the interpretation in the succeeding step:

$$p_j(t_{\sigma_{\iota_1}}, \ldots, t_{\sigma_{\iota_n}}) \in \mathbf{T}^{ord}_{\mathcal{P}} \uparrow (i+1)$$

□

With that, the model-theoretic equivalence of least fixpoints of a corresponding COMBILOG and definite clause program is established. This equivalence is not literal, due to the difference in structure between an ordinary interpretation and an extension map, but their isomorphism is trivially observable through Definition 2.2.12.

In the next section, we will observe the execution of COMBILOG programs through a meta-interpreter and conclude the semantics of COMBILOG.

2.2.9 Execution of COMBILOG programs

COMBILOG predicate expressions are executed through SLD Resolution through a meta-interpreter written in PROLOG, in line with the semantics of the language that is equivalent to definite clauses.

The meta-interpreter consists of a predicate named *comb*, where the first argument is a term in the object language, that is, a COMBILOG predicate expression given as a complex term, and the second argument is a list containing the object language arguments. The predicate *comb* stands for the application of given predicate expression to the given arguments. Warren's earlier work outlines this approach for implementing higher-order constructs in Logic Programming through an *apply* predicate [121]. In our implementation, this corresponds to the *comb* predicate, in

order to differentiate it from the CNP interpreter which is of concern in later Chapters 5 and 7.

For example, the elementary predicates *id* and *cons* are implemented as follows:

$$comb(id, [X, X]).$$
$$comb(cons, [X, T, [X|T]]).$$

The operators of the language correspond to separate clauses of the *comb* predicate. For example, the *and* operator employs the higher-order arguments P and Q as operand predicate expressions within the complex term in the first argument:

$$comb(and(P, Q), Args) \leftarrow comb(P, Args) \wedge comb(Q, Args).$$

Similarly, the *foldr* operator employs higher-order arguments, P for recursive case, and Q for the base case, implemented as two clauses:

$$\begin{aligned} comb(foldr(P, Q), [Y, [], Z]) &\leftarrow comb(Q, [Y, Z]). \\ comb(foldr(P, Q), [Y, [X|T], W]) &\leftarrow comb(foldr(P, Q), [Y, T, Z]) \wedge \\ &\quad comb(P, [X, Z, W]). \end{aligned}$$

The meta-interpreter is given as a PROLOG program in Appendix B.

2.3 Program synthesis using a reversible meta-interpreter

For an object language L, the provability relation of L can be constructed as a demonstration predicate $demo(\mathcal{P}_L, \mathcal{G})$ where $\mathcal{P}_L$ is a meta-representation of a program P in L and $\mathcal{G}$ is the meta-representation of a goal G. This *demo* predicate stands as a meta-interpreter, which succeeds only if with some substitutions θ the goal follows from the given program, $P_L \vdash G_\theta$. This approach to meta-logic programming was first described by Kowalski [60].

The meta-interpreter can be implemented as a reversible predicate, in which case can work in the reverse direction. Instead of taking a program P and goal G to produce substitutions θ to satisfy $P \vdash G_\theta$, it can be used to input a goal G to find a program P that satisfies $P \vdash G$.

Exploitation of a reversible meta-interpreter was described by Numao and Shimura [80] as a part of a system for explanation-based learning. The figure from this work which explains the use of a meta-interpreter in reverse direction is given in Figure 2.1. Similar approaches were developed by Sato [102], Christiansen [21, 22, 23], and Hamfelt and Nilsson [45, 46], and recently by Muggleton [74].

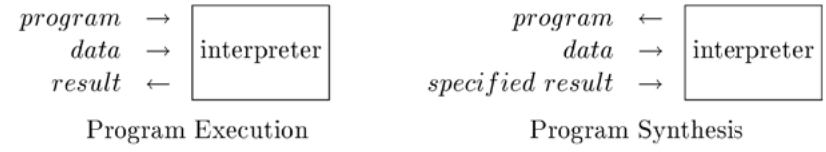


Figure 2.1. Reversible Interpreter figure from Numao and Shimura [80]

Hamfelt and Nilsson's approach CombInduce is distinct from the others in that they assume a strictly compositional language void of free variables with a dual of generic list recursion operators *foldl* and *foldr*. These grounds prove to enrich the capabilities for synthesis. Since the recursion operators are also compositional, the given examples of program data can be decomposed according to the semantics of the operators, obtaining reduced examples for operand predicates which can be plugged-in as operands of the fold operators [50].

2.4 Summary

In this chapter, we first looked into essential concepts in Logic Programming, including model-theoretic and proof-theoretic semantics. Second, we described the structure of COMBILOG programs, including the elementary predicates, logic and generalized projection operators, and the recursion operators.

We followed with the transformation from definite clauses to COMBILOG clauses in the canonical form, and finally gave the fix-point correspondence of definite programs and the COMBILOG programs. Finally we briefly described the concept of program synthesis using reversible meta-interpreters.

This chapter did not contain any novel contributions, but it stands as a theoretical background for Logic Programming and COMBILOG, and as a reference for later chapters.

In the following chapter, we will analyse the COMBILOG syntax from a usability point of view.

3. An analysis of COMBILOG notation

In the Introduction, shortcomings of the COMBILOG syntax were briefly addressed, particularly those related to argument binding. In this chapter, a number of usability factors affected by these shortcomings are investigated with help from a cognitive load evaluation framework. Finally the notational issues will be summarized. In the later chapters two separate approaches to address these issues will be presented.

To discuss these usability factors with any rigor, we require commonly understood definitions, or a common vocabulary. Green's *Cognitive Dimensions of Notations* is an appropriate established framework. Introduced in [42] by Green, and later developed by Green and his colleagues in [44], and [43], *Cognitive Dimensions of Notations* are an ever growing list of definitions for usability factors for notations. Initially developed for a shallow assessment of notation quality, they are now used in many other usability settings, such as user interfaces, diagrams, and textual programming languages. They do not by themselves provide an absolute way to decide if a notational artifact is better than another or describe how to design better programming languages; but they facilitate these processes by providing a common and established understanding via discrete factors.

Now, we present some of Green's cognitive dimensions that are relevant to our COMBILOGcontext, and discuss shortcomings of COMBILOG syntax in terms of these. Cognitive dimensions framework is often over-applied to make a final assessment of language usability, but they are not designed to serve this purpose [31]. They are intended as a discussion tool only. Even though a complete sweep of every cognitive dimension ever identified would give the impression of a complete formal treatment, it would be an abuse of the framework. Instead, we only employ them as a vocabulary as we identify the difficulties of the notation. For final assessment we consult to within-subjects usability tests for both representations we propose, in later chapters. For a complete list of cognitive dimensions, refer to [42, 43, 44], and [14] for recent additions.

In the following section, we will look at the following cognitive dimensions originally described in Green's [42]. The list of dimensions we will discuss in COMBILOG are:

- Hidden/explicit dependencies
- Viscosity/fluidity
- Premature commitment

- Role-expressiveness
- Hard mental operations

As an extension, we will discuss the following two dimensions that were introduced later [44], which we find relevant to issues in COMBILOG's notation:

- Closeness of mapping
- Progressive evaluation

In the next section we will discuss these dimensions in relation to COMBILOG notation.

3.1 Cognitive Dimensions of COMBILOG notation

First, let us look at the *append* predicate in COMBILOG from earlier, given below:

$$
\begin{aligned}
append \leftarrow or(and(&make([1,2,3], const_{[]}),\\
&\quad make([3,1,2], id)),\\
&make([1,2,3],\\
&\quad and(make([3,4,5,1,2,6], cons),\\
&\qquad make([4,2,5,6,1,3], append),\\
&\qquad make([4,5,3,1,6,2], cons))))
\end{aligned}
$$

This example is not arbitrary. It is the actual implementation of a fundamental library predicate a programmer would use frequently, analogous to the list concatenation available in virtually every programming language. It employs the fundamental concept of recursion. Moreover, it contains instances of all three functions of the *make* operator of COMBILOG: reordering, cropping, and expansion.

Using this predicate definition, let us now consider the relevant cognitive dimensions and the usability of the COMBILOG in each dimension.

3.1.1 Hidden dependencies

When a notation does not readily convey the information that is necessary to build a mental model of a concept, it may be said to contain hidden dependencies. In [42], Green gives the example of spreadsheets, where each cell's formula contains references to other cells, but only the computed value of a cell is visible. The computational dependencies between different cells is hidden, and it often requires looking into a cell's formula to figure out which cells it depends on. An even worse instance of hidden dependency emerges when the user has to figure out if there are any cells depending on the particular cell they are about to edit. Green

cites [16], which finds that even experienced users of spreadsheets browse around in cells, looking for dependencies on a particular cell.

This example resonates with the use of *make* operator in COMBILOG, in particular with expansions which introduce new unbound arguments. In the predicate definition, we can readily see all the dependencies as source predicates appearing as a part of the expression body. But the same does not apply when their arguments are modified. Let us look at a line from the example above:

$$make([3, 1, 2], id)$$

It is clear that this expression depends on a predicate named *id*, yet figuring out the intention with the arguments is not straightforward. The first argument index, 3, is introducing an unbound argument, yet inferring this information requires the user to know that the *id* predicate has only 2 arguments; hence a reference to a non-existing third argument results in an unbound one. Besides unbound arguments, referring to arguments via their indices does not convey full information about the bound arguments either. Here, the second argument of the new expression is bound to the value of 1^{st} argument in *id*, but no hint is given about what this 1^{st} argument may actually be in its own context, or over what range of values it may be bound to. While reading or modifying an expression of this sort, a programmer often needs to browse into the definition of *id* to inspect its arguments, which is a symptom of a hidden dependency.

3.1.2 Viscosity

Viscosity of a notation is proportional to how much resistance it shows to change. Some notations may require more points of intervention than others, for a similar even seemingly small conceptual change to the codes's model. Program code is often destined to change radically over time, even in the course of the first development phase. For this reason, a more fluid notation is significantly more accommodating over the lifetime of a codebase.

There are two separate concerns connected to viscosity in COMBILOG. The first regards sub-expressions. Complete compositionality in COMBILOG ensures *referential transparency*, which in turn guarantees a consistent meaning for expressions, regardless of the code context in which they appear. As an example, let us consider the same line of code again:

$$make([3, 1, 2], id)$$

The line above produces an intermediate predicate body (or expression) that takes three arguments, where the first argument is unbound, and the second and third arguments are bound to the same value through the *id* predicate. The meaning of the expression does not change as a result of the host context. This means a programmer may duplicate, move, or define-and-reuse this partial code freely in a different context with fewer modification. This a positive viscosity outcome for COMBILOG.

On the other hand, argument binding in COMBILOG reveals a different difficulty. Let us look at this logic operator and its sub-expressions from the *append* predicate defined earlier:

$$\begin{aligned} &and(make([3,4,5,1,2,6], cons),\\ &\quad make([4,2,5,6,1,3], append),\\ &\quad make([4,5,3,1,6,2], cons)) \end{aligned}$$

The code uses the *cons* predicate, which defines three arguments in the following order: head of a list, tail of a list, and the whole list. Hence, the first *make* expression produces six arguments in the sequence: [a list, unbound, unbound, head of the list, tail of the list, unbound]. The problem arises when the definition of a source predicate changes. Let us suppose the argument order of the *cons* predicate changes due to the refactoring of a library. The arguments of the new *cons* predicate are ordered as: the whole list, head of a list, and tail of a list. This would require both lines above which use the *cons* predicate to be rewritten, since they refer to specific indices in the source predicate. We discuss this issue under viscosity, but it could also be interpreted an issue related to the *error proneness* dimension that we do not observe separately, as it may result in errors being introduced to code after a change.

These circumstance may occur often during program development, and the consequences shown significantly increase the viscosity of COMBILOG.

3.1.3 Premature commitment

During the course of typing a program, premature commitment occurs when the rules of the notation require the programmer to make some decisions earlier than their natural cognitive plan for the task would otherwise dictate. In [42], Green quotes an anecdote about the development of speech-recognition software for writing *Pascal* programs, intended to be used by people who have difficulty typing. The problem was that the program used context-awareness to improve word recognition, and required that words entered always followed correct *Pascal* grammar. Consequentially, the programmer had to construct the complete statement, and also the complete sub-procedure in their mind beforehand in order to use

speech-recognition. Predictably, at least with hindsight, the software was not actually usable.

In COMBILOG, a similar issue arises when working with the generalized projection operator *make*. Let us look at this earlier example:

$make([3, 1, 2], id)$

At one point in the composition of the predicate, the programmer needs to write an expression that will generate an intermediate-predicate with three arguments where the last two are bound to the same value. Naturally, the first concept that enters one's mind is *equality*, followed by the *id* predicate which is the implementation of equality. In the usual left-to-right typing environment, the programmer is requested to first decide the specific reordering, cropping or expansion plan and then the correspnding index list for the *make* operator, before actually typing the name of the *id* predicate.

This effect is magnified with nested applications of the *make* operator. Before sub-expressions are composed, their argument order and count have to be decided. Let us refer back to the second part of the *append* example presented earlier:

$$\begin{array}{l} make([1, 2, 3], \\ \quad and(make([3, 4, 5, 1, 2, 6], cons), \\ \quad\quad make([4, 2, 5, 6, 1, 3], append), \\ \quad\quad make([4, 5, 3, 1, 6, 2], cons))) \end{array}$$

In the code above, the first instance of the *make* operator takes the first three arguments of the following expression, and crops out the rest. At this point, the programmer would have to decide how many intermediate arguments the sub-expression has, and which of those will be relevant to the outer context.

In this case, for example, the arity of the predicate expression produced by the *and* operator is six, even though only the first three of these arguments are projected by the containing $make([1, 2, 3], \ldots)$ operation. The reason for allocating the extra three arguments is establishing some argument bindings that are irrelevant to the outer context. For example, in the context of the *and* operator, the second argument of the first *cons* component is aligned with the first argument of the *append* component. Due to the lack of variables, allocating these arguments is necessary, and this necessity requires the user to plan ahead how many of these *auxiliary* arguments to use. Index list of the *make* operators are affected as a result, since their lengths and indices will be affected by the number of arguments. For example, if there were 7 arguments in total, the index list of the *make* operator that projects the *cons* (line 2) would have been

$[3, 4, 5, 1, 2, 6, 6]$ instead. This often requires the programmer to devise a mental plan for the entire expression before typing, hence prematurely committing to that plan.

3.1.4 Role-expressiveness

Role-expressiveness is one of the dimensions introduced in Green's original work [42]. It questions if the components of the notation convey their role in the semantics efficiently. A common example is the flowcharts often used to describe algorithms. Some semantic components of a flowchart, such as conditionals are representative of their function. On the other hand, some components such as loops are not. In order for the reader of a flowchart to find a loop, he/she has to study a fragment of the flowchart and look for patterns that form a loop. As a result, while flowcharts are role-expressive for a small range of essential operations, they are not role-expressive for higher-level patterns, such as loops or forks in the algorithm.

In COMBILOG, most operators can be considered role-expressive. The logic operators and the recursion operators have particular purposes, and they are consistently used for these purposes. One issue we can note is with the *make* operator. This operator can be used for multiple functions, including adding arguments (*expanding*), removing arguments (*cropping*), or reordering them. But the notation does not make it obvious which of these operations are at play at a given *make* operation. The user has to study the indices, and the context, in order to find out which role(s) a particular *make* operator performs.

3.1.5 Hard mental operations

This dimension covers those aspects of a notation that requires the reader to perform difficult mental operations to gather the information that is not conveyed directly by the notation, but is fundamental to understanding the meaning. The second part of the *append* predicate is a good candidate to demonstrate this dimension:

$$
\begin{array}{l}
make([1,2,3], \\
\quad and(make([3,4,5,1,2,6], cons), \\
\quad\quad make([4,2,5,6,1,3], append), \\
\quad\quad make([4,5,3,1,6,2], cons)))
\end{array}
$$

Let us consider the task of figuring out the binding of the first argument of the first *cons* predicate. The *make* operator that appears before projects the *cons* predicate, so that the first argument becomes the fourth in the newly produced predicate expression. After the logic operator, the

argument orders are kept consistent, but the outer *make* crops out the arguments after the third. Hence, the first argument of the first *cons* predicate is encapsulated, and will not be bound by an operator in the outer context. The fact that tracing even a single argument causes this much cognitive load is obviously not a desirable property of a notation.

Another aspect of COMBILOG that enhances this effect is the fixed-arity nature of logic operators. This requires every operand to have the equal number of arguments, therefore expanding every component predicate to the same arity. This can be observed in the example above, where each component predicate is expanded to arity 6. This is a common issue in relation composition without access to variables, as it becomes difficult to express the argument mapping between relations in composition. Tarski's *Relational Calculus* is another example of this issue [113], which restrains the relation composition $S \circ R$ to only binary relations. In this case, the second domain of R and the first domain of S are bound to the same value, resulting in binding of arguments akin to function composition. For relations of higher arity, it is not straightforward how to establish the argument mapping. Codd's *Relational Algebra* [25] offers a relief with unordered domains which can be referred by a domain name, where the relation domains are mapped using their names.

As demonstrated, the argument binding method in COMBILOG needs a solution, as tracing arguments through *make* indices creates a high cognitive load.

3.1.6 Closeness of mapping

A program is intended to reflect an abstract model of a problem solution. This abstract model is often a mental one, and sometimes a technical, low-level algorithm. In either case the similarity between the notation and the abstract model is important for the usability of a programming language. Green names this measure of similarity *closeness of mapping* [44].

In COMBILOG, argument binding requires handling indices. These are artificial references to the arguments of source predicates, which often may readily have an intuitive name. Indices convey almost no meaning regarding the arguments they stand for, and introduce a decoupling between the abstract model and the actual code.

3.1.7 Progressive evaluation

During development, programmers often feel the need to test an incomplete section of code, to verify whether their cognitive model of the code is in alignment with the actual code. This concept is called *progressive evaluation* by Green [44]. It is most useful for novice programmers, since progressive evaluation helps finding errors in code earlier rather than latter, and is also useful for assessing the progress of the task at hand, and maintaining a direction. Debugging also often benefits from partial evaluation of code segments.

COMBILOG's syntax brings both ease and difficulty to progressive evaluation. A positive feature of the language is that, since it is completely compositional, parts of it can be evaluated in isolation without regard to a context. However, an issue arises if the user would like to evaluate a logical operator where one of the operands has a syntax error. Let us consider this somewhat modified example from earlier:

$$\begin{aligned} &\mathit{and}(\mathit{make}([3,\ ,\ ,1,2,\ ],\mathit{cons}),\\ &\quad \mathit{make}([\ ,2,\ ,\ ,1,3],\mathit{append}),\\ &\quad \mathit{make}([\ ,\ ,3,1,\ ,2],\mathit{cons})) \end{aligned}$$

In this case, since the projection indices for the *make* operators are incomplete, the logical operator *and* is also incomplete and cannot be evaluated. It is possible to imagine a notation that allows the construction of valid expressions from incomplete sub-expressions.

In COMBILOG, the combination of using indices for projection, and the requirement of the logic operators to have operands of the same arity hinders progressive evaluation.

3.2 Summary

In this chapter we identified the shortcomings of the COMBILOG notation in terms of six cognitive dimensions. The root causes of these shortcomings are summarized in three issues:

I1. The *make* operator uses indices to represent argument binding. Indices are unintuitive and prone to break code undergoing change.

I2. The structure of the *make* operator is impractical, since it forces the user to type the projection of arguments of a predicate before the predicate itself. This also limits the use of IDE features such as predictive typing.

I3. The restriction of the logic operators that makes them deal with only operands of the same arity makes the notation difficult to understand and modify. It results in repetitive code to expand com-

ponent predicates expressions to the same arity.

In the remaining chapters, we present two solutions to address these problems. The first solution uses a visual representation side-by-side with COMBILOG code, and is presented in Chapter 4.

The second solution introduces a new syntax to replace the *make* operator, and alters the behaviour of the logic operators. This notation is presented in Chapter 5.

4. VISUAL COMBILOG

In this chapter we develop a visual representation to augment the experience of dealing with textual COMBILOG programs. This technique of visualizing predicate compositions and the accompanying methodology for interacting with these visualizations, which we call VISUAL COMBILOG, has been iteratively published in earlier work [82, 83].

In the following sections, we first review visual programming methods primarily developed for declarative programming, including Bertin's work on *visual variables*. We then present our visual representation of COMBILOG predicate compositions and illustrate using different predicate definitions. As a proof-of-concept, we implement a split-view editor of VISUAL COMBILOG that displays the textual COMBILOG code in one view, and the corresponding VISUAL COMBILOG diagram in another interactive view. To determine whether in fact the VISUAL COMBILOG helps with the comprehension of COMBILOG programs, we present the results of a user study.

4.1 Visual programming

The appropriate application of visualization to the declarative programming paradigm has been established to increase the human-friendliness of programs. [15] gives a survey of visual programming methods, and one of the conclusions is that using text and its visualisation side-by-side achieves the optimal effect. In our work on VISUAL COMBILOG we follow this route and combine the conciseness of the textual representation of COMBILOG with the intuitiveness of a visual one.

Visualization of logic formulas or programs is not a recent development. In fact, it pre-dates computer graphics. Peirce established one of the earliest formalizations of predicate logic in his work on *existential graphs* [87, 88, 90], which are the basis of many forms of predicate logic to this day. Even though there is still a debate about the appropriateness of visualizations to represent formal notations, especially over concerns regarding soundness, in her book [106], Shin discusses this in length with examples and concludes that diagrams are capable of expressing scientific proofs, let alone logic formulae. Shin claims they also provide a medium where our visual reasoning skills can assist. Moreover, she provides proofs

of soundness and consistency for her visual systems based on Venn diagrams.

There has been earlier attempts at visualizing logic programming languages. One of these is Lograph, a visual interactive diagram system analogous to Prolog programs [9, 28]. It visualizes predicate definitions on a 2-dimensional canvas using icons to represent body literals and logic operators, and represents variable bindings with lines (edges) connecting these icons. Another is CUBE, a higher-order logic programming language that uses a 3-dimensional system [75], implemented in [76]. Visualizations methods have been applied also to Answer Set Programming, where an integrated visual editor is used to compose body graphs [36, 37]. The most relevant work is [124], where the object of visualization is also COMBILOG. In this work, a model of the diagrammatic representation for COMBILOG is given in terms of object-oriented data structures. The model can generate COMBILOG code from data structures, but the visualization is only at a preliminary stage with no implementation of a graphical user interface.

Our approach differs in a number of aspects from these earlier systems. Our point of departure is COMBILOG, which fundamentally separates our work from most of those mentioned earlier. In contrast to the work based on COMBILOG in [124], we provide a full visualization system and an implementation of the system that can transform COMBILOG programs to VISUAL COMBILOG and vice versa, with an accompanying interactive graphical user interface. In contrast to all previous work, we build our design of VISUAL COMBILOG on human-centric principles, and assess the effectiveness of our visual system with a usability trial.

4.2 Visual variables

Encoding information in a graphical manner requires making decisions on how to associate graphical concepts with the source data. In his work "Semiology of Graphics" [13], Bertin describes elements of visual representation as position (horizontal and vertical), size, colour, value (density), texture, orientation and shape. These elements were later expanded and came to be called *visual variables* [55] [64]. Bertin was interested in the efficacy of visualizations in representing abstract knowledge, and established the varying accuracy of each visual variable in representing different types of information. He counted these types (levels) as selective, associative, quantitative, ordinal, and measure (length).

For example, let us imagine a group of objects on a 2-dimensional canvas. These objects shall be of the same colour, but different shapes, such as squares, discs, stars or line segments. If we wanted to focus on the objects that have the same shape at a glance, it would be easy. This

is because the shape is an accurate visual variable for selective purposes. On the other hand, if we would like to order these objects, their shapes would be useless. For more analysis and applications, MacKinlay's work in [65] is a good example of how to apply Bertin's conceptions to visualize relational information.

In the following section, we will revisit the issues we identified in Chapter 3 and build a notation that makes use of Bertin's visual variables to aid the comprehensibility of COMBILOG's textual notation.

4.3 Representation of VISUAL COMBILOG

In this section, we will build the representation of VISUAL COMBILOG diagrams step by step, after the analysis we performed via cognitive dimensions in Chapter 3. Let us first recall the results of this analysis as three numbered issues, which we will continually refer back to in this section:

I1. The use of indices as a part of the *make* operator is unintuitive.
I2. The syntactic structure of the *make* operator is not practical.
I3. The use of logic operators is not flexible.

We will address these three issues through visual representation. Let us begin by listing the three kinds of objects that constitute COMBILOG programs.

1. Predicate expression. These include elementary predicates, program predicates, and expressions constructed as a result of a composition operation (through *make*, *and* and *or*.)
2. *make* operations. These operations perform the generalized projection over a predicate expression and produce another predicate expression.
3. *and* and *or* operations. These composition operators are defined over two or more predicate expressions and produce another predicate expression.
4. *foldr* and *foldl* recursion operators.

Let us define the elements of visual representation one by one, covering each of these COMBILOG constructs.

A predicate expression contains a number of arguments with a specific order. Therefore, the visualization of a predicate expression needs to address this. Our design uses the size variable for this purpose (see section 4.2). Each argument of a predicate is displayed with a solid disc of the same colour with successively decreasing size. The choice of the same colour is useful because the colour variable is highly selective: our eyes can focus on a specific colour easily, using a visual search for colour, as shown in [116, 122]. The decreasing size of each argument is determined according to the following rules. The first argument has an area u, and

the last argument has an area $u/2$. All arguments in between have areas calculated via linear interpolation between u and $u/2$. In order to visually associate arguments of a predicate expression, the corresponding discs are connected with a line. This line is employed for visual selectivity, redundant to the colour, and do not represent binding between arguments. A predicate expression with 3 arguments, with *orange* colour assignment is displayed in Figure 4.1. This diagram only shows a predicate with three arguments, and no bindings between them.

Figure 4.1. Visual representation for a predicate expression with 3 arguments.

In the next step, we will display a COMBILOG predicate definition that consist only of an assignment, that is,

$$r \leftarrow p$$

While we describe the example diagrams, we will refer to the predicate or predicate expression being defined as the *target* predicate, and predicate expression(s) in the body as the *component* predicates. In the example above, the target predicate is r, and the single component is p. Assuming p has three arguments, the target predicate r also has three arguments. Since r is also a predicate expression itself, we can represent it also with three discs of decreasing size. In a diagram with multiple predicates, every predicate is assigned a different colour, and the black is reserved for the resulting *target* predicate. It is always planted at the center of the diagram, and serves as a base for the entire diagram. The representation of the COMBILOG predicate definition $r \leftarrow p$ is given in Figure 4.2. Note that every disc of p occludes the corresponding disc in r. This is because assignment $r \leftarrow p$ binds the arguments in the same order with each other. As a logic formula, this could have be written as $\forall X.\forall Y.\forall Z.\ r(X,Y,Z) \leftarrow p(X,Y,Z)$. This is the method for visualizing arguments that are bound together. The discs that represent bound arguments are drawn as occluding (or touching) each other. When two circles do not touch, it means the corresponding COMBILOG code does not bind those arguments; unless, of course, there is a separate *id* predicate involved.

The *make* operator is the fundamental way to manipulate argument binding in COMBILOG. In our visual representation, we do not employ a separate visual element for *make*, but instead we alter the positions of the discs representing each argument, making one disc visually occlude the other. The black discs are always displayed in the background as

Figure 4.2. Visual representation for $r \leftarrow p$ where p is a predicate expression that has 3 arguments.

they belong to the target predicate expression. As a straightforward example, let us look at the visual representation of $make([1,2,3],p)$, given in Figure 4.3. Since p has 3 arguments, and the index list of the *make* operator includes exactly the index of every argument in increasing order, the operation has the same semantic effect as the identity, and produces extensionally the same result as $r \leftarrow p$. Consequentially, the visual representation is also identical to Figure 4.2.

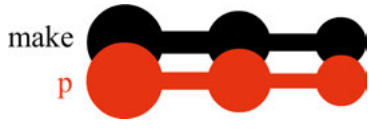


Figure 4.3. Visual representation for $make([1,2,3],p)$ where p is a predicate expression that has three arguments.

As a more functional application of the *make* operator, let us again assume a source predicate p with three arguments. The expression $make([3,4,1],p)$ produces a new predicate expression with three arguments, but now the argument bindings are different. The index list implies that the first argument of the target predicate expression is bound to the value of third argument in p (*reordering*), second argument of the target predicate expression is not bound to the value of any argument of p (*expansion*, since $4 >$ arity of p), and the third argument of the target predicate expression is bound to the value of the first argument in p (*reordering*). Consequentially, the second argument of p is not bound to the value of any arguments in the target predicate expression (*cropping*). The expression is displayed in Figure 4.4. Note that the location of the target predicate expression (painted in black) does not change as it forms the base of the diagram.

This method is capable of representing the semantic effect of the *make* operator solely by intentionally positioning the discs representing the arguments. All three functions of the *make* operator (reordering, cropping, expansion) are covered, and it significantly helps the comprehension of argument bindings as it is visually evident, and relieves some of the difficulty related to issues I1 and I2.

Figure 4.4. Visual representation for $make([3,4,1],p)$ where p is a predicate expression that has three arguments.

Logic operators are displayed with a similar representation, similar to the identity *make*. Since there are two separate logic operators, it is imperative to have a distinct visual representation for each. For this purpose, we employ the *texture* variable, as it is highly selective. When a logic operator is visualized, the base is painted solid black if it is an *and* operator, and dashed black if it is an *or* operator. These are shown in Figures 4.5 and 4.6.

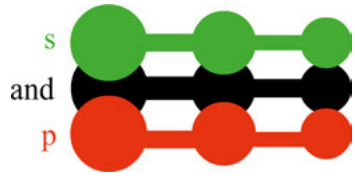


Figure 4.5. Visual representation for $and(s,p)$ where s and p are predicate expressions with three arguments.

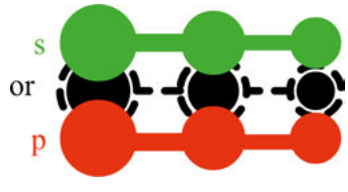


Figure 4.6. Visual representation for $or(s,p)$ where s and p are predicate expressions with three arguments.

In Figures 4.5 and 4.6, the operands of the logic operators are predicate expressions with equal arity, and they are used with no modifications to their argument order. This is not always the case. The canonical form of COMBILOG programs, for example, specifies predicate definitions in a form that has nested applications of the logic operators and the *make* operator.

Let us visualize the expression $and(make([4,2,3],s),make([1,2,4],p))$, where the source predicate expressions s and p both have three argu-

ments. The first *make* has three indices ($[4, 2, 3]$): the first (4) produces an unbound argument, the following two indices produce arguments that are bound to values of those in s in the given positions. The second *make* has three indices as well ($[1, 2, 4]$): the first two indices produce arguments that are bound to the values of arguments in p in the given positions, the last (third) index produces an argument which is bound to the value of no arguments from p. This expression is visualized in Figure 4.7. As a result, the three arguments of the resulting predicate expression are bound as follows: first argument is bound to the the value of the first argument in p, the second argument is bound to the value of the second arguments in both s and p, and the third argument is bound only to the value of the third argument in s.

The example in Figure 4.7 reflects two important results of combining logic operators and the *make* operator in a single diagram. Firstly, when the visual representation is used to compose predicate expressions with a logic operator, the repetitive code which usually arises due to expanding the operands to the same arity is eliminated. By simply leaving some of the argument discs of the base untouched, an unbound argument is implied by the diagram. Secondly, by combining two levels of operations together, the resulting diagram yields a simpler hierarchical structure than its textual counterpart. This improves the readability of logic operators: addressing issue I3.

In those cases where a logic operator is the source for a *make* operator that performs only expanding or cropping, the *make* operator can be incorporated into the diagram with no loss of readability. An example of this can be seen in the next section, in the visualization of the *append* predicate.

Figure 4.7. Visual representation for $and(make([4, 2, 3], s), make([1, 2, 4], p))$ where s and p are predicate expressions with three arguments.

The recursion operators *foldr* and *foldl* in COMBILOG can also be visualized in terms of logic operators and *cons*, according to their meta-logic definitions, but this could imply the operators defining the *fold* can be modified by the user as well. Therefore we commit to a special visualization of *fold*s. When *fold*s are visualized as a component, they're

visualized as an ordinary ternary predicate. When a *fold* itself is visualized, two joint diagrams are used, one binary corresponding to the base case Q displayed on the upper part of the visualization, and one ternary corresponding to the recursive case P displayed on the lower part. For example, the visualization of $foldr(cons, id)$ corresponds to two separate diagrams, where the upper one is a visualization of $Q \leftarrow id$, and the lower one is a visualization of $P \leftarrow cons$.

The representation described here is aimed at local visualizations of COMBILOG code. When entire programs are considered, there are further questions to be answered. One of these is the layout problem that arises when multiple predicates, or nested expressions are involved. Another is a colouring problem; since in every diagram predicates are assigned to unique colours, with a large number of predicates involved it is not obvious how to assign colours while keeping visual complexity low.

We have described construction of VISUAL COMBILOG diagrams for predicate definition, the *make* operator, the logic operators, and finally for the combination of logic operators with instances of the *make* operator. We also showed how the *fold* operators can be reduced to a simpler visualization. Let us exemplify this representation in the next section.

4.4 Example VISUAL COMBILOG definitons

In this section we will present VISUAL COMBILOG diagrams corresponding to COMBILOG predicate definitions *append* and *atpos*.

4.4.1 *append* predicate

The *append* predicate was described in Chapter 2 as an example predicate definition. Here we revisit this predicate in VISUAL COMBILOG. The COMBILOG code for *append* is repeated here in Figure 4.8 for convenience, and the corresponding VISUAL COMBILOG diagrams are given in Figures 4.9, 4.10, and 4.11.

Note that a single COMBILOG predicate definition can only be represented by three VISUAL COMBILOG diagrams. The visualization of the second *and* operator in fact represents the containing $make([1, 2, 3], ...)$ operator as well. These kinds of simplifications are more suitable for instances of *make* which perform only expansion or cropping, as combining multiple levels of reordering would render comprehension of diagrams themselves difficult. An instance of this arises in the *atpos* example which follows in the next section.

The smaller blue discs inside the black discs in Figure 4.10 serve the purpose of relating the colour of this diagram's target predicate to the host context in which it appears (Figure 4.9).

$$
\begin{aligned}
append \leftarrow or(&and(make([1,2,3], const_{[]}),\\
&\quad make([3,1,2], id)),\\
&make([1,2,3],\\
&\quad and(make([3,4,5,1,2,6], cons),\\
&\quad\quad make([4,2,5,6,1,3], append),\\
&\quad\quad make([4,5,3,1,6,2], cons))))
\end{aligned}
$$

Figure 4.8. COMBILOG code for the *append* predicate

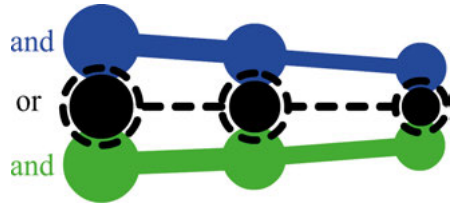


Figure 4.9. Visual representation for the top-level *or* logic operator of the *append* predicate given in Figure 4.8

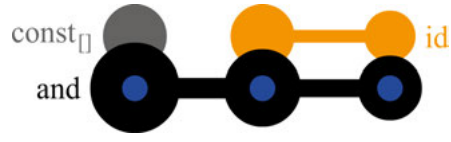


Figure 4.10. Visual representation for the first *and* operator (base case) of the *append* predicate given in Figure 4.8.

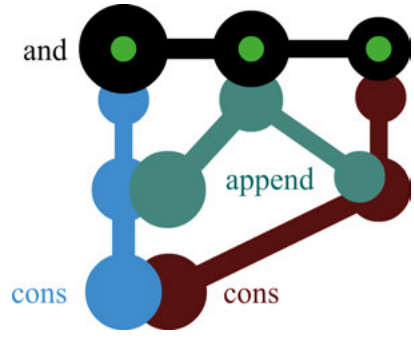


Figure 4.11. Visual representation for the second *and* operator (recursive case) of the *append* predicate given in Figure 4.8

4.4.2 *atpos* predicate

Our second example is the *atpos* predicate, which describes the relation between a list, its positions and elements at those positions. The predicate definition assumes the existence of a library predicate called *length* with two arguments: the first argument is a list, and the second argument is the number of items in that list.

The *atpos* predicate can be displayed in a single diagram given in Figure 4.13, but since the topmost *make* operator is a reordering operation, comprehending the overall structure of the *atpos* predicate is more difficult than the *append* predicate. However, it still offers some help with figuring out the individual argument bindings. For example the second argument of the *atpos* predicate is bound to the value of the second argument of the *length* predicate. While this is difficult to Figure out from the COMBILOG code in Figure 4.12, it is straightforward to see from the VISUAL COMBILOG diagram in Figure 4.13.

$$\begin{aligned} atpos \leftarrow make([3,6,4], and(&make([1,2,3,4,5,6], append),\\ &make([4,3,5,1,2,6], cons),\\ &make([1,3,4,5,6,2], length))) \end{aligned}$$

$$atpos(L, I, E) \leftarrow append(L1, L2, L), cons(E, \_, L2), length(L1, I).$$

Figure 4.12. COMBILOG code, and the corresponding PROLOG code for the *atpos* predicate

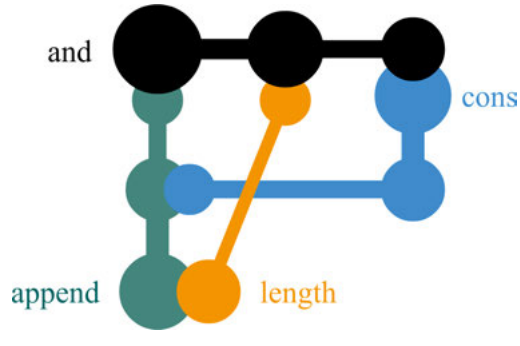


Figure 4.13. Visual representation for the *atpos* predicate given in Figure 4.12

In this section, we presented examples of VISUAL COMBILOG visualizing COMBILOG code. We pointed out a graph drawing problem in the *atpos* example, and how it can be eliminated to some degree by visualizing multiple *make* operators together with a logic operator to minimize

crossing edges. The solutions to this problem are left as future work. A formal description of this problem as well as a literature review is given in [12].

4.5 Graph formalization of VISUAL COMBILOG

A VISUAL COMBILOG diagram is formalized as an undirected simple graph. The diagram graph $G(N, E, E_\circ)$ is constructed as a union of subgraphs $G_0, \ldots, G_n$ and the addition of binding edges $E_\circ$. Subgraph $G_0(N_0, E_0)$ corresponds to the target predicate, and subgraphs $G_1(N_1, E_1), \ldots, G_k(N_k, E_k)$ correspond to component predicates $1, \ldots, k$ respectively. Let us define the subgraphs and binding edges separately, and then the diagram graph in its entirety. With these definitions we will present accompanying visual graph representations that are topographically similar to VISUAL COMBILOG diagrams given earlier, but lacking the visual cues such as thickness of lines and size of discs. Moreover, the argument binding that is represented by touching discs here is shown by binding edges labeled with the $\circ$ symbol.

Every diagram corresponds to a composition either in the form of a single *make* operator, where the single component predicate is q and the target predicate is the predicate denoted by application of *make*:

$$make([\mu_1, \ldots, \mu_n], q)$$

or it is in the form of a logic operator with multiple (k) *make* operators as operands:

$$\begin{aligned}\{and/or\}(&make([\mu_{1,1}, \ldots, \mu_{1,n}], q_1),\\ &\cdots,\\ &make([\mu_{k,1}, \ldots, \mu_{k,n}], q_k))\end{aligned}$$

The two forms are treated uniformly, where an index list for a make is given as $M_i = [\mu_{i,1}, \ldots, \mu_{i,n}]$, and the component predicate as q_i. The arity of a component predicate q_i is denoted as $\#_{q_i}$.

Definition 4.5.1. *A subgraph $G_i(N_i, E_i)$ with a set of nodes N_i and a set of edges E_i corresponding to a component q_i with assigned colour c_i consists of the following:*

- *For every argument j of the predicate q_i ($1 \leq j \leq \#_{q_i}$), a node $(c_i, j) \in N_i$*
- *For every pair of sequential arguments j_n, j_{n+1} of q_i, an edge $(c_i, j_n, j_{n+1}) \in E_i$.*

Figure 4.14. Visual graph representation of a ternary predicate with colour assignment *blue*.

Example: *for a component q_i with two arguments and the colour assignment blue, the corresponding subgraph $G_i(N_i, E_i)$ is given as:*

$$N_i = \{(blue, 1), (blue, 2), (blue, 3)\}$$
$$E_i = \{(blue, 1, 2), (blue, 2, 3)\}$$

This definition of a subgraph is exemplified as a visual graph in Figure 4.14.

Definition 4.5.2. *The set of binding edges $E_\circ$ is constructed as follows. Each of the index lists M_i have the same number (n) of indices given as $\mu_{i,1}, \ldots, \mu_{i,n}$. The number of component predicates is given as k, and the colour assigned to a component predicate q_i is given as c_i.*

- *For every argument position j where $1 \leq j \leq n$, the set of indices of corresponding to various $\mu_{i,j}$ where $1 \leq i \leq k$ and $\mu_{i,j} \leq \#_{q_i}$ is taken as $\circ_j$. For each pair of indices $\mu_{i_1,j_1}, \mu_{i_2,j_2} \in \circ_j$, the set of binding edges contains an edge $(black, (c_{i_1}, \mu_{i_1,j_1}), (c_{i_2}, \mu_{i_2,j_2}))$.*
- *The set of target predicate argument positions is indexed by j where $1 \leq j \leq n$. If the composition is immediately under a cropping make operator, only the argument positions j referred in the index list of this make operator are taken. For each j, the set of binding edges contains an edge $(black, (black, j), (c_i, \mu_{i,j}))$ where $\mu_{i,j} \in \circ_j$.*

Definition 4.5.3. *The diagram graph $G(N, E, E_\circ)$ corresponding to a composition given by make indices $M_1, \ldots, M_k$ and component predicates $q_1, \ldots, q_k$ is defined as follows:*

- *Every component predicate q_i is assigned to a distinct colour c_i.*
- *For every component predicate q_i a subgraph $G_i(N_i, E_i)$ is constructed.*
- *For the n-ary target predicate, a subgraph G_0 is constructed with the colour assignment black.*
- *The set of nodes N is the union of all nodes in the subgraphs, $N = N_0 \cup N_1 \cup \ldots \cup N_k$.*
- *The set of edges E is the union of all edges in the subgraphs, $E = E_0 \cup N_1 \cup \ldots \cup E_k$.*
- *The set of edges $E_\circ$ is constructed according to the Definition 4.5.2.*

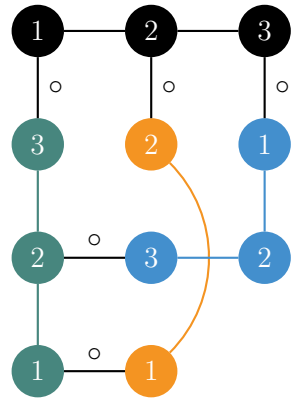


Figure 4.15. Graph formalization example for the *atpos* predicate. Component predicates are assigned to the following colours: *append*(green), *length*(orange), *cons*(blue)

The visual graph constructed for the *atpos* predicate given earlier is shown in Figure 4.15. The edges labeled with ∘ represent argument binding.

4.6 A split-view editor for VISUAL COMBILOG

In the previous section, we devised a visual system which can represent COMBILOG program. In this section, we present our split-view editor that can simultaneously edit COMBILOG programs and the corresponding VISUAL COMBILOG diagrams. A screenshot of the editor is given in Figure 4.16.

The editor can parse COMBILOG code and automatically generate a VISUAL COMBILOG program. The reverse is also true, so when the user manipulates the diagram (e.g. adding arguments, predicate expressions, logic operators and modifying argument bindings by moving discs around) the corresponding COMBILOG code immediately reflects the change.

The diagrams generated by the editor are slightly different from those described earlier. In the editor, the discs representing the arguments of the base are drawn significantly bigger than those corresponding to the operand predicate expressions. This is done intentionally to reduce graphical occlusions, providing a workaround for the graph drawing problem. For a usability comparison this would have major implications but the editor is intended as a proof-of-concept for showing the feasibility of real-time transformation between COMBILOG and VISUAL COMBILOG programs. A second difference is that the target predicate is coloured in orange rather than black. The colour black is reserved for highlighting the selected predicate. The arguments of a component predicate are

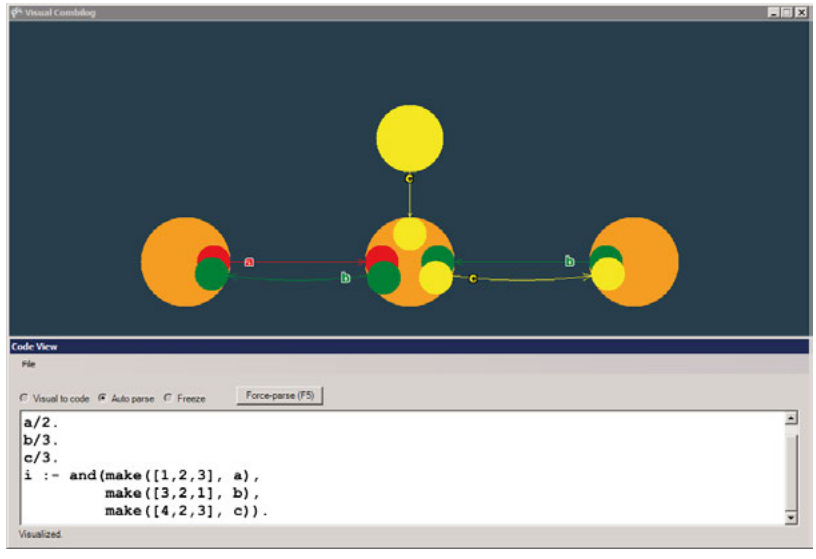


Figure 4.16. A screenshot of the split-view editor

not ordered by size, they are instead ordered by directed edges, as in $1 \rightarrow 2 \rightarrow 3$ for a ternary predicate.

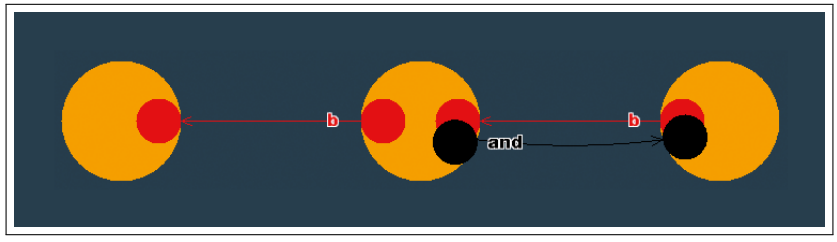
The editor enables some actions to be performed through the diagram without touching the code. These actions are as follows:

Browsing: When the diagram being displayed contains a logic operator as a component, the user can click on it and the editor will switch to displaying the selected logic operator instead. This is shown in Figure 4.17 through screenshots. Similarly, the user can right-click on the diagram and the editor will browse to the outer composition (if there is any).

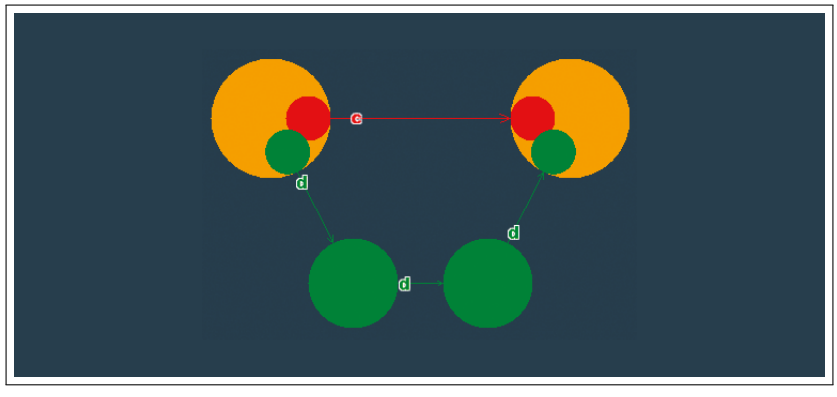
Adding components: Component predicates or logic operators can be added through the diagram. When the editor is displaying a logic operator, if the user double-clicks on the diagram an *add predicate* pop-up appears. Using this input, the user can add a component predicate, or anotler logic operator as a component by typing the name of the logic operator (as in *and*). Optionally an arity may be specified as well (as in *and*/3). This action is demonstrated in Figure 4.18.

Selecting: When the user places the mouse over an element of the diagram, such as a predicate, a logic operator, or an argument, the corresponding section of the code is highlighted.

As a proof-of-concept of the VISUAL COMBILOG diagram system, the split-view editor demonstrates the feasibility of developing a fully graphical system representing COMBILOG code. The implementation highlighted the difficulties of a full implementation, primarily the graph drawing prob-



(a) VC editor displaying definition of p, where *and* is hovered by the user. Upon clicking the diagram focuses on the *and* operator.



(b) VC editor displaying the *and* operator.

Figure 4.17. Two screenshots demonstrating browsing into a logic operator.

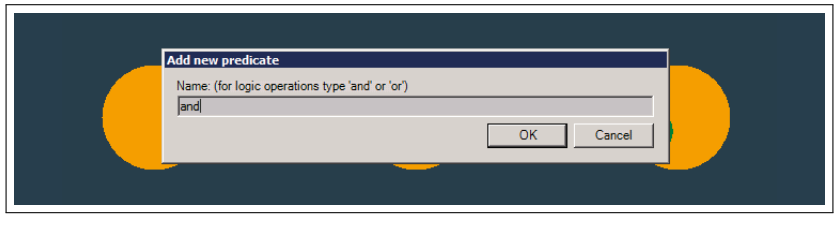


Figure 4.18. Screenshot demonstrating adding a predicate through the diagram.

lem. Another issue is the lack of an established common data format for diagrams, as there is with text e.g. ascii and unicode. This makes it impossible to benefit from tools and operating system capabilities that are available to text, such as difference tools and cut-and-pasting. These issues will be discussed further under section 4.9.

In the next section, we present a user study of the VISUAL COMBILOG notation.

4.7 User study of VISUAL COMBILOG

In this section we present a usability study with the intention of finding out if VISUAL COMBILOG diagrams do in fact help the readability of COMBILOG programs.

For the experiment we reached out to participants and asked them to take a test on paper. The test included a brief introduction to COMBILOG and VISUAL COMBILOG, and afterwards participants were presented with 3 predicate definitions of increasing complexity accompanied by 12 questions related to the argument bindings established by the definitions.

In order to measure the usability of COMBILOG independently from VISUAL COMBILOG, two sets of questions were devised: T and V. In part T, only COMBILOG code was presented for each predicate definition. In part V, both the COMBILOG code and VISUAL COMBILOG diagram(s) were included. Since VISUAL COMBILOG is intended to be used alongside with COMBILOG, we do not attempt to measure it on its own.

To provide the test with controls, we divided participants randomly into two groups, A and B. Group A answered part T first, then part V; group B answered part V first and then part T. This control is intended to factor out the effect of learning the notation as a participant reaches the second part of the test.

In the following sections we give a description of the predicate definitions and the test questions, and present the results.

4.7.1 Questions

The test is based on three predicate definitions of increasing difficulty. Variants of these three questions were produced for the parts T and V, but both variants followed the same structure shown in the following templates:

1. $p \leftarrow make([\ldots], q)$
2. $p \leftarrow or(make([\ldots], q), make([\ldots], r))$
3. $p \leftarrow make([\ldots], and(make([\ldots], q), make([\ldots], r), s))$

The first definition has only one instance of the *make* operator. The second has two instances of *make* inside a logic operator. The third has two instances of *make* inside a logic operator which itself is inside another *make* operator.

Moreover, the index lists of the *make* operators (shown in the templates with placeholders '[...]') were devised with increasing difficulty as well. The index list in the first definition only reflects the cropping behaviour of the *make* operator. The second reflects both reordering and cropping. The third reflects all of the three behaviours: reordering, cropping and expansion.

The 12 questions were designed to reveal the following:

(a) Can the participant tell if particular arguments are bound to the same value?
(b) Can the participant identify the argument(s) that are bound to the same value?
(c) Can the participant identify the components that have at least one argument bound to the same value?
(d) Can the participant work out the arity of the predicate resulting from the composition?

The questions in the study did not differ between parts T and V. This was not found necessary since the predicate definitions were different, and it would have introduced the possibility of bias.

4.8 Results

A total of 20 participants were recruited to take the test. They consisted mainly of graduate students from a declarative programming course and a few colleagues (with no conflicts of interest). Out of the participants, 18 were male, 2 were female, and 4 participants were using corrective glasses.

The study was carried out in a room with an accurate time displayed using a projector. The participants were asked to write down the time at multiple points during the test. This was the main method of measuring the duration. The session was supervised but there was no verbal guidance regarding the contents of the test.

Task time. Participants completed part T (text only) in 9m29s on average. On the other hand, they completed part V (visuals and text) in 5m6s. This is a 46% decrease in task time with significance level $P < 0.01$.

Moreover, group A which took part T first finished the whole test in 17m23s on average, in contrast to group B which took the part V first and finished the whole test in 11m47s, revealing a major advantage in learning

speed ($P < 0.01$) when the participants were introduced to COMBILOG in conjunction with the visualizations.

Difficulty. Out of the 12 questions in part T, participants answered 8.1 questions correctly on average. For part V, the score was 10.1, translating to 25% more correct answers or equivalently 69% fewer mistakes when using the visualizations ($P < 0.01$).

If we look at differing difficulties of each predicate definition, the questions regarding first and second definitions had 90% correct answers, while those regarding the third definition had 75% correct answers on average. This confirms our preliminary experiments using the *make* operator, where we observed that nested applications of the operator render the comprehension of expressions significantly more difficult.

Opinion. At the end of the test, participants were asked two opinion-based questions and allowed to pick answers from a 5-point Likert scale. The questions were "1. Accompanied with visuals, the code was easier to understand." and "2. Accompanied with the visuals, the code was quicker to understand.". For both questions the median response was "Strongly agree".

4.9 Summary and conclusions

In this chapter we presented VISUAL COMBILOG, a visual representation for COMBILOG programs specifically designed to address the issues of usability in COMBILOG notation reported in Chapter 3.

We put these ideas into practice with the implementation of a split-view editor. This is an application of VISUAL COMBILOG where it is serves as an aid for reading and manipulating COMBILOG programs. Finally, we devised a usability test and presented the results, which show significant improvements, including a 46% reduction in task time and 69% fewer mistakes.

These results of the usability test also show that there is significant room for improvement in COMBILOG notation in terms of usability. Actual real-world COMBILOG programs, or those produced by inductive synthesis could be significantly longer and more complex, but we believe we have successfully isolated the issues regarding the usability of the COMBILOG notation.

Some expressions, especially for cases of n-ary logic operators ($n > 2$), a layout problem arises. If it was not for the simplifications possible in Figure 4.11, the arguments of the base would have to be laid out in a different way, for example to radically reduce visual complexity. This problem is more apparent in the *atpos* example in Figure 4.13. To this end, we suggest the use of automatic layout algorithms as employed in [8]

and [12], but we leave such algorithms for VISUAL COMBILOG as future work.

Finally, the visualization we present provides a suitable tool to read COMBILOG programs. However, the effect was only particular to localized views of the program. Every operator of the expression tree can be visualized, but when we merge the visualizations to span the entirety of a predicate definition, we are prevented by the principle of compositionality. When arguments that cannot be bound together are displayed on the same diagram, this may convey the wrong implication to the user. Alternative ways of marking the incompatible arguments would be required. An alternative approach could be *lifting* the arguments, a transformation process similar to *lambda lifting* [54], but this kind of refactoring may result in significant automatic changes to the code structure. The effect of such changes require further usability analysis to find out to what degree the users may tolerate them.

For future work, alternative uses of these visualizations could be considered, such as for debugging, or tracing the data flow. Since VISUAL COMBILOG enables navigating through localized views of the program, it could be useful for debugging.

From the experiments, we concluded that the perceptional benefits of a diagrammatic notation are significant. But some practical issues persist surrounding day-to-day use. These issues make complete adaptation of a diagrammatic notation impractical, despite all human-centric benefits observed in isolation.

1. Diagrams are not as fluid as text on current operating systems. There is no operating system independent support for copy-and-pasting. With text it is possible to copy and paste the code into another application, performing some modifications on the text and then feeding it back into the original environment. There are no methods available for pipelining such diagrams. Vector graphics tools are not yet established enough to be adapted for such uses, and even if they did they would not be enough to transfer the semiotic information reflecting the relation between the visual elements.

2. Patterns of functionality such as *Find* and *Replace* for text manipulation are not readily-available for diagrams. Similar features would have to be implemented on a case-by-case basis and would only be available on the specific development environment on which they are implemented.

3. Tools surrounding text-based programming are not established for visual programming. While tools such as code validation; difference comparison; merging and version control are readily available and

configurable for almost every text-based language; their counterparts in visual programming would have to be implemented from first principles for a diagrammatic system.

Therefore we conclude that it is best to combine visual and textual notations in a way that makes use of the advantages of both. In the next chapter we present our work on the textual notation of COMBILOG in order to achieve some human-centric improvements in the textual domain as well.

5. Combilog with Name Projection: CNP

In Chapter 3, we identified some usability issues regarding COMBILOG. In this chapter, we introduce COMBILOG with Name Projection (CNP), a variant of COMBILOG intended to improve usability. To this end, we will first present a set of modifications to COMBILOG syntax, which results in the proposed CNP syntax. Secondly, we give a description of the syntax and informal semantics of CNP programs in Section 5.2, which is included in our recent work in [84] as well. Thirdly, we present the denotational semantics of CNP programs in Section 5.3. This includes a description of α-extensions, a specific form of relational extension we use, which permits compositionality to be maintained as argument names are introduced to the notation.

Finally, we present rewriting rules between COMBILOG and CNP, and demonstrate that they can be used to transform CNP programs to COMBILOG programs, and by extension to definite clause programs. We support these transformations by proving that the least fixpoint of a CNP program obtained by these transformations is isomorphic to the least fixpoint of the corresponding COMBILOG program.

5.1 Usability improvements over COMBILOG

In this section, we describe a list of changes to COMBILOG syntax. These modifications are based on the preceding Cognitive Dimensions of Notations analysis and informal experiments using COMBILOG syntax.

The changes mainly involve the *make* operator, but they also encompass the specific behaviour of the logic operators, since they need to comply with the new sort of expressions produced by *make* projections.

For reference, we shall start with the implementation of the *append* predicate in COMBILOG, without consulting to the recursion operators. Quoting from Chapter 2, Section 2.2.5:

$$
\begin{aligned}
append \leftarrow or(&and(make([1,2,3], const_{[]}),\\
&\quad make([3,1,2], id)),\\
&make([1,2,3],\\
&\quad and(make([3,4,5,1,2,6], cons),\\
&\qquad make([4,2,5,6,1,3], append),\\
&\qquad make([4,5,3,1,6,2], cons))))
\end{aligned}
$$

For convenience, let us also repeat the equivalent definition in PROLOG:

$$\begin{aligned} append(L1, L2, L) \leftarrow & const_{[]}(L1), id(L2, L). \\ append(L1, L2, L) \leftarrow & cons(X, T, L1), \\ & append(T, L2, Lmid), \\ & cons(X, Lmid, L). \end{aligned}$$

In the following sections we list the changes to the COMBILOG notation step by step, arriving at the final notation for CNP.

5.1.1 Names for arguments

The use of names as arguments, instead of indices, may help greatly to eliminate *hidden dependencies*. Let us first look at the part of the *append* predicate we visited earlier:

$$make([3, 4, 5, 1, 2, 6], cons)$$

The first argument of the make operator refers to arguments of the *cons* predicate by index. Let us assume that *cons* has names for its arguments, with respect to their original positions:

$$cons : \{a, b, ab\}$$

then in the *make* projection, we could refer to these arguments by name instead of indices 1, 2, and 3:

$$make([ab, 4, 5, a, b, 6], cons)$$

Now, the reader can immediately tell that the first argument of the new expression is bound to the same values as the *ab* argument of the *cons* predicate, without browsing into the definition of *cons*, eliminating a *hidden dependency*. The indices 4, 5, and 6 refer to unbound arguments, and since they do not have a name assigned in *cons*, they remain as indices for the time being. In the final notation, we omit these indices altogether.

This approach requires every argument in a predicate expression, and arguments of its sub-expressions, to be assigned names. This is easy to achieve for elementary predicates, by introducing argument names into their definitions. But as the original COMBILOG operator *make* reordered arguments to align them, we need a means to align the names, since the logic operators need an indication of which arguments to bind. For this, we use the $\mapsto$ symbol for renaming, as in $a \mapsto b$ to express renaming

argument a to b. It is crucial to rename the arguments that were in the same position, to the same name, in order to preserve the semantics of the predicate expression. If a projection preserves the name, as in $a \mapsto a$, we write it as only a, without the renaming symbol. In the example below, we look at a code fragment, the recursive case of the *append* predicate, which we shall refer to as $append_{rec}$:

$$
\begin{array}{l}
make([xs, ys, list], \\
\quad and(make([ab \mapsto xs, 4, 5, a \mapsto head_{xs}, b \mapsto tail_{xs}, 6], cons), \\
\qquad make([4, ys, 5, 6, xs \mapsto tail_{xs}, list \mapsto inlist], append), \\
\qquad make([4, 5, ab \mapsto list, a \mapsto head_{xs}, 6, b \mapsto inlist], cons)))
\end{array}
$$

Internally, it contains a logic operator and argument mappings between a *cons* predicate, a recursive call to the whole of the *append* predicate, and another *cons* predicate. To the outside context, it only exposes the arguments named *xs*, *ys*, and *list*, since the outer *make* operator only projects those arguments. Figure 5.1 displays a table with all the argument bindings in the example above.

Operationally, when this definition is used to append a list *xs* with another list *ys* to produce the resultant list *list*, the first *cons* unpacks *xs* into its $head_{xs}$ and $tail_{xs}$; the recursive call to *append* joins $tail_{xs}$ and *ys* to produce an intermediate list *inlist*; and the second *cons* constructs *list* from $head_{xs}$ and *inlist*.

Context	Arguments					
cons	ab			a	b	
append		ys			xs	$list$
cons			ab	a		b
and	xs	ys	$list$	$head_{xs}$	$tail_{xs}$	$inlist$
(final)	xs	ys	$list$			

Figure 5.1. Argument map for the recursive case of the *append* predicate. The table shows arguments that are bound to the same values, in the same column.

It is important to note that even though we have introduced identifiers for arguments, they are functionally different from variables. With variables, identical identifiers appearing in two or more discrete positions in the same scope would show a symbolic binding between the arguments. With identifiers in a projective configuration, this is not the case. This becomes more evident in a nested application of name such as the example below (which is not a part of the *append* predicate):

$$and(make([b \mapsto a, 4], cons),$$
$$\quad make([ab \mapsto a, b \mapsto b], cons))$$

For the *and* operator, the operand predicate expressions's arguments have only the names obtained after they're renamed, not their original argument names. For example, the second *make* expression has the argument names a and b only. In this expression, if b was a variable, b of the first *cons* would be bound to the value of b in the second *cons*; but instead it is bound to the value of ab in the second *cons*, through the renaming of both to a. Therefore, argument names have no scope in the sense bound variables do.

5.1.2 Unordered arguments

Our notation so far still includes a number of indices for unbound arguments, in order to form components of the same arity for the logic operators. The existence of names eliminates the need for arguments with fixed locations. Fixed locations for arguments are used to denote which arguments should be bound to the same values, as evident in Figure 5.1. But with argument names as a part of the expression, it is possible to use this meta-information to deduce the bindings. Eliminating the requirement for arguments in a specific order will further reduce both the *premature commitment* and *viscosity*.

Moving from arguments indexed by positions to arguments indexed by names, requires changing the behaviours of the logic operators. Instead of binding the arguments with the same location, they will now bind arguments with the same name. As a result, the arguments of a logic operator expression comprise all the arguments from its components, producing an *auto-expanding* effect which removes the manual expansion required by COMBILOG's logic operators. This further reduces the *hard mental operations* encountered by a programmer modifying expression arguments. We shall call these auto-expanding operators *ande* and *ore*, replacing *and* and *or*, respectively.

The auto-expanding logic operators also partially allow *progressive evaluation* which speeds up prototyping, as the programmer does not need to plan argument locations. Some degree of planning is still required to bind arguments by renaming, but even with an incomplete or invalid mapping, *auto-expanding* logic operators would lead to evaluable code that allows the progressive detection of errors.

Let us now remove the unbound indices, and convert to an unordered list for projected arguments:

$$\begin{array}{l} make(\{xs, ys, list\}, \\ \quad ande(make(\{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\}, cons), \\ \qquad make(\{ys, xs \mapsto tail_x, list \mapsto inlist\}, append), \\ \qquad make(\{ab \mapsto list, a \mapsto head_x, b \mapsto inlist\}, cons)) \end{array}$$

The use of unordered arguments reduces viscosity by allowing argument lists of source predicates to be modified or extended while keeping the dependent code intact. Let us imagine a library change where the signature of *cons* changes from $cons : \{a, b, ab\}$ to a quaternary variant $cons : \{head, tail, tailLength, list\}$. Both in COMBILOG and PROLOG, all the dependent code that uses the *cons* predicate would have to be refactored, while the CNP code that uses unordered arguments would continue to work as expected without requiring any modification. This is due to elimination of tight coupling to source predicates, specifically to arity of a source predicate and the order of its arguments.

Let us imagine a follow-up scenario, where the newly added *tailLength* argument of *cons* is removed, reverting its signature to $cons : [head, tail, list]$ again. In this case, all the dependent COMBILOG and PROLOG code that uses the quaternary variant would have to be altered again to reflect the change in argument number and locations. Dependent CNP code would not require any modifications and continue to work, as long as no new code making use of the *tailLength* argument had been introduced. We will refer to this feature of the CNP language as *change resilience*.

5.1.3 Use of the *make* operator

So far, we have used the original *make* operator from COMBILOG, but modified the way it projects arguments. There are two further aspects of this generalized projection operator that can be improved to reduce *premature commitment*. In the original form, the argument names come before the source predicate. To better reflect the thought process of a programmer, we can put the predicate identifier before the argument projection list. Instead of writing

$$make(\{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\}, cons)$$

we can write

$$make(cons, \{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\})$$

This form is better, but it still requires the programmer to determine the existence of a projection, by typing '*make*(' before the predicate identifier. To avoid this, we can omit an explicit call to the *make* operator by making it implicit. Whenever a list of projections appear in brackets after a predicate identifier or an expression, as in *cons* {...}, we can interpret it as an instance of the generalized projection operator *make*.

After removing the explicit *make* operator, the recursive append example $append_{rec}$ can be written as:

$$\begin{aligned} &ande(cons\ \{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\}, \\ &\quad append\ \{ys, xs \mapsto tail_x, list \mapsto inlist\}, \\ &\quad cons\ \{ab \mapsto list, a \mapsto head_x, b \mapsto inlist\})\ \{xs, ys, list\} \end{aligned}$$

The *make* operator has three essential behaviours regarding how it modifies arguments: reordering, cropping and expansion. The new implicit projection operation does not need to be able to perform expansion, since that behaviour is taken over by the auto-expanding logic operators. Moreover, because the predicate arguments are unordered in CNP, reordering of arguments is not possible. This results in a new operator that specializes in cropping and renaming. We shall call this new operator *proj*, short for *projection*, and use the prefix notation for formal treatments such as mathematical proofs and definitions. In code, we shall use the implicit use demonstrated in the example above.

This final form is the updated syntax we propose as a variant of COMBILOG, which we name *Combilog with Name Projection* (CNP). A table displaying how constructs compare in each language can be found in Figure 5.2. In the table, COMBILOG elementary predicates are given with their arities, as in *id/2*. The new *const* predicate has two arguments, A and C. C is the constant, as it also exists in the COMBILOG original, and A determines the name of the single argument of the predicate. For example, $const(five, 5)$ produces a predicate where the single argument is named *five*, and bound to the value 5.

With all the changes in place, we shall now give the implementation of the whole *append* predicate in CNP.

$$\begin{aligned} append = ore(&ande(const(nil, [])\ \{nil \mapsto xs\}, \\ &\quad id\ \{a \mapsto ys, b \mapsto list\}), \\ &ande(cons\ \{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\}, \\ &\quad append\ \{ys, xs \mapsto tail_x, list \mapsto inlist\}, \\ &\quad cons\ \{ab, a \mapsto head_x, b \mapsto inlist\})\ \{xs, ys, list\} \end{aligned}$$

When the definition of *append* is observed, the new syntax is not necessarily more compact, but the intention here is improving usability in terms of readability and modifiability, not necessarily terseness. In the

practical notation, in order to decrease premature commitment, we have chosen to write the projection items that project a name as it is, as in $a \mapsto a$ as a single a, as a form of syntactic sugar. This is only acceptable for the practical notation. For formal purposes, we always use the *proj* operator explicitly, and every projection item is written in the form $a \mapsto a$ or $a \mapsto b$. We shall refer to this as the *strict* notation. Considering this, the definition of *append* above would be written in the strict notation as:

$$\begin{aligned}
append = ore(&ande(proj(const(nil, []), \{nil \mapsto xs\}),\\
&\quad proj(id, \{a \mapsto ys, b \mapsto list\})),\\
&proj(ande(proj(cons, \{ab \mapsto xs, a \mapsto head_x, b \mapsto tail_x\}),\\
&\quad proj(append, \{ys \mapsto ys, xs \mapsto tail_x, list \mapsto inlist\}),\\
&\quad proj(cons, \{ab \mapsto list, a \mapsto head_x, b \mapsto inlist\})),\\
&\{xs \mapsto xs, ys \mapsto ys, list \mapsto list\}))
\end{aligned}$$

Let us give another example, which contains definitions of *isParent*, *isFather*, and *isGrandchild* predicates. Assuming the *parentage* and *isMale* predicates exist already, with the given signatures:

$$\begin{aligned}
parentage &: \{parent, child\}\\
isMale &: \{name\}
\end{aligned}$$

Then the predicates would be defined as:

$$\begin{aligned}
isParent &\leftarrow proj(parentage, \{parent \mapsto parent\})\\
isFather &\leftarrow ande(proj(parentage, \{parent \mapsto name\}),\\
&\qquad proj(isMale, \{name \mapsto name\}))\\
isGrandchild &\leftarrow proj(ande(proj(parentage, \{parent \mapsto gparent,\\
&\qquad child \mapsto parent\}),\\
&\qquad proj(parentage, \{parent \mapsto parent,\\
&\qquad child \mapsto child\})),\\
&\quad \{child \mapsto name\})
\end{aligned}$$

In the following sections, we will develop the semantics of CNP, and formulate the steps to transform from COMBILOG predicates to CNP predicates.

5.2 CNP semantics

In this section, we briefly describe the structure and syntax of CNP programs, before we move onto the denotational semantics in the next section.

Use	COMBILOG	CNP
Elementary predicates	$const(C)/1$ $id/2$ $cons/3$	$const(A, C) : \{A\}$ $id : \{a, b\}$ $cons : \{a, b, ab\}$
Logic operators	*and* *or*	*ande* *ore*
Recursion operators	*foldr* *foldl*	*foldr* *foldl*
Modifying arguments · Reordering · Renaming · Cropping · Expansion	 *make* – *make* *make*	 – *proj* *proj* *ande*, *ore*

Figure 5.2. A side-by-side comparison of COMBILOG versus CNP language constructs. Predicate or operator signatures are shown as p/m in COMBILOG where m is the arity, and as $p : \{a_1, \ldots, a_m\}$ in CNP where $\{a_1, \ldots, a_m\}$ is the set of argument names of p.

The overall structure of CNP programs is identical to COMBILOG. A CNP program consists of one or more predicate definitions. Each predicate definition is composed of a predicate identifier (p) and a body (φ_p) which is a predicate expression. An equals sign (=) is used to assign the body to an identifier, as shown:

$$p = \varphi_p$$

This notation contrasts with the use of the implication sign ($\leftarrow$) in definite clause programs and COMBILOG. The intention here is to reflect *extensional equivalence*. The extension of the predicate symbol p is equivalent to the extension of the body φ_p.

A predicate expression is constructed by the following rules:

- A predicate identifier is a predicate expression. This includes the identifiers for elementary predicates which are defined as a part of the CNP language, and the program predicates that are defined as a part of the program.
- A projection of a predicate expression is a predicate expression.
- A logical composition of two or more predicate expressions using logic operators *and* and *or* is a predicate expression.

Elementary predicates are part of the CNP language and provide primitive functionality. They are identical to the elementary predicates of COMBILOG, except the arguments have names. Program predicates are those defined as a part of the program. Projections are expressed through the *proj* operator, in the form of $proj(E, P)$ where E is a source predicate

expression and P is a set of projection items that can perform selection or renaming. Basic logic operators are *and* for conjunction and *or* for disjunction. Operators construct predicate expressions syntactically as a tree, and semantically as a graph since multiple leaves of the expression tree can refer to the same predicate, or even to the predicate that has the expression as a part of its body (recursion).

Every predicate expression has a set of names representing its arguments. In an ordinary logic predicate $p(X, Y)$, we would refer to the arguments by their position, as in 1st argument or 2nd argument, which are bound to the values of variables X and Y, respectively. CNP replaces these numeric positions with nominal positions. A nominal predicate signature is written as $p : \{a_1, a_2, \ldots\}$, where $a_1, a_2, \ldots$ are the names of arguments of p. This is in contrast to an ordinary predicate signature such as $p/2$, which only expresses that p has 2 arguments. Hence, we refer to the arguments of a CNP predicate expression by their name, as in 'argument a of E', where E is a predicate expression. This method of referring to arguments by names is akin to the *domain-unordered relations* described by Codd in [25].

At this point we should clarify that CNP is variable-free, just as COMBILOG, which means it needs a method for binding arguments of predicate expressions. This is established through the projection and logic operators by binding arguments that have the same name. We will give examples as we progress through the definitions.

Let us now define the elementary predicates of CNP, given by their nominal signatures:

$$\begin{aligned} \mathit{true} &: \{\} \\ \mathit{const}(A, C) &: \{A\} \\ \mathit{id} &: \{a, b\} \\ \mathit{cons} &: \{a, b, ab\} \end{aligned}$$

The predicate *true* has no arguments and is always *true*.

The predicate $\mathit{const}(A, C)$ has two meta-variables, where C is a ground term, and has one argument named A. It provides a way to introduce constants into a program through the definition of a unary predicate expression which is *true* only for the single value C, such as:

$$\mathit{isNil} = \mathit{const}(\mathit{nil}, [])$$

The *id* predicate represents the identity, taking two arguments: a and b. It is *true* if and only if the given two values for arguments a and b are identical.

The *cons* provides a standard way to work with lists and other data structures. It has three arguments: a, b, and ab. It is *true* when the

value for the *ab* argument is the list construction of those given for the arguments a and b. It is akin to the conventional *cons*) function found in LISP. The list construction is semantically denoted with the list construction function $(\cdot)$, but a programmer working with the CNP language only has access to this function through the *cons* predicate, as CNP does not have constructs for dealing with function terms directly.

The projection operator takes a source predicate expression, and a set of projection items, and produces a new predicate expression where arguments are bound to values of arguments from the source. Every projection item relates to an argument from the source, either in the form of a selection or a renaming. Projection items are mappings in the form of $a \mapsto b$ where a is an argument name from the source, and b is the corresponding new argument name in the resulting predicate expression. Hence, projection items where $a = b$ perform *selection*, and those where $a \neq b$ perform *renaming*.

A projection operation is written as follows, where $\langle a_i, b_i \rangle$ are the projection items and E is the source expression:

$$proj(E, \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\})$$

In this form, the *proj* operator can perform the *cropping* and *renaming* but not the *expansion* behaviour of the *make* operator found in COMBILOG. This *expansion* behaviour is subsumed by the logic operators in CNP, as explained later.

For the *proj* operator, we formally use the *strict notation*, meaning we write explicitly $a_i \mapsto b_i$ for every projection item i, even if $a_i = b_i$. For every semantic and formal treatment this is necessary in order to keep the projection set semantically equivalent to a function. But as a form of syntactic sugar, we define an auxiliary *practical notation* for *proj*, in which we omit one of the names if $a_i = b_i$, and omit the projection operator name. For example, a projection operation written as follows in the strict notation,

$$proj(E, \{a \mapsto c, b \mapsto b\})$$

would be written as follows in the practical notation:

$$E\ \{a \mapsto c, b\}$$

The practical notation is reserved for practical uses only, such as plain text representations. For this reason, we use the practical notation for the user study in the next chapter, Chapter 6. But throughout this chapter, only the strict notation is used.

The logic operators provided by CNP are the *and* operator for conjunction, and the *or* operator for disjunction. The logic operators both

take a list of predicate expressions as operands, and produce a new predicate expression. Both operators are multiary (n-ary, where $n \geq 2$). In the remaining sections, as we define denotational semantics for multiple logic operators, we only deal with binary variants, but devising multiary variants in terms of nested applications of the binary ones is trivial.

Logic operators accept operands that are predicate expressions with different signatures. This contrasts with COMBILOG, where logic operators only accept operands with equal number of arguments. In COMBILOG, due to the variable-free form, this is necessary for establishing a binding between the operand expression arguments. Instead in CNP, argument names act as a clue for deducing a binding scheme. The argument names of the composed predicate expressions encompass the argument names of each component (*auto-expanding*). For every operand expression, each argument is bound to the value of the corresponding argument in the composed predicate expression. This establishes indirect bindings between operand expressions, when they have arguments sharing the same name. When argument names of the operands are not appropriate for the intended binding scheme, projection can be applied to one or more of the operands. This is the main method of binding arguments in CNP, and significantly it is done without using variables. The denotational semantics of projection and logic operators is given in section 5.3.

Multiary logic operators of CNP are written as *ande* and *ore*:

$$ande(E_1, ..., E_n)$$
$$ore(E_1, ..., E_n)$$

The *e* suffix in the operator names implies auto-expanding behaviour. In virtually every logic composition of COMBILOG, arguments of the operands must be reordered, cropped or expanded through the use of the *make* operator. As shown in the Chapter 3, this significantly hinders the usability of the language. Through the improvements suggested in Section 5.1, CNP eliminates most involved cases by eliminating the necessity to introduce argument names for operands manually.

Due to the new expressiveness gained from argument names, *auto-expanding* logic operators form only a single class of many possible name-aware logic operator classes. Variant logic operators can be derived which apply alternative argument binding schemes.

CNP provides name-aware counterparts to the list recursion operators found on COMBILOG. These operators, namely *foldr* and *foldl*, are fixpoint combinators that eliminate the need for explicit recursion for single recursion and data structures constructed using *cons*. In order to keep these operators simple, and to avoid further introducing meta-variables to the operators, the argument names of the operator and the operands are predetermined, and are as follows:

$$
\begin{aligned}
\mathit{foldr}(P, Q) &: \{a0, as, b\} \\
\mathit{foldl}(P, Q) &: \{a0, as, b\} \\
&\mathit{where} \\
&\quad P : \{a, b, ab\} \\
&\quad Q : \{a, b\}
\end{aligned}
$$

Intuitively, the argument *a0* is the initial term, *as* is the input list, and b is the resulting term from the folding operation. The higher-order arguments P and Q are predicate expressions, where Q is the base case, and P is the recursive case. *foldr* passes the value of *a0* to Q's a to get the first running value. In contrast, *foldl* passes the second to last running value to Q's a to obtain the result of folding. In the most common case, the base case is the identity predicate ($Q = id$), which does not modify the passed value. The recursive case P takes an a and b and produces an ab. The functionality is narrated here in one direction, but both fold operators and their operand expressions are multi-directional. Both fold operators in CNP are identical to their COMBILOG counterparts except the introduction of argument names. We also include the variant *foldr2*, which omits the argument *a0* and initializes the running value through a unary base case $Q : \{b\}$.

We have given an overview of CNP programs, predicate definitions, predicate expressions and the operators of CNP. In the next section, we give a description of the abstract syntax tree of CNP programs.

5.2.1 CNP Syntax

The abstract syntax of CNP programs is given in Figure 5.3. A program consists of one or more (denoted with the symbol +) predicate definitions in the form of *PredicateId* = *Expression*. An *Expression* which is the body of a predicate, often referred to as a *predicate expression* can be either one of the operators of the language (*proj*, *ore*, *ande*, *foldr*, *foldl*), a predicate identifier, or an elementary predicate. The logic operators are treated in their binary versions here and in most other parts of this document, under the assumption that their multiary variants, as well as nullary and unary ones can be defined straightforwardly.

The elementary predicate *const* is parametric, where N is the argument name and C is the constant. The parameterization is for abstraction purposes only. When used in a program, the parameters N and C always have to be ground, as CNP is variable-free. Similarly, in a projection item in the form of $A \mapsto B$, A and B are ground argument names in a valid CNP expression.

$$
\begin{aligned}
\mathit{Program} &\to \mathit{PredicateDefinition}+ \\
\mathit{PredicateDefinition} &\to \mathit{PredicateId} = \mathit{Expression} \\
\mathit{Expression} &\to \mathit{proj}(\mathit{Expression}, \{\mathit{ProjectionItem}+\}) \\
\mathit{Expression} &\to \mathit{ore}(\mathit{Expression}, \mathit{Expression}) \\
\mathit{Expression} &\to \mathit{ande}(\mathit{Expression}, \mathit{Expression}) \\
\mathit{Expression} &\to \mathit{foldr}(\mathit{Expression}, \mathit{Expression}) \\
\mathit{Expression} &\to \mathit{foldl}(\mathit{Expression}, \mathit{Expression}) \\
\mathit{Expression} &\to \mathit{PredicateId} \\
\mathit{Expression} &\to \mathit{true} \\
\mathit{Expression} &\to \mathit{const}(A, C) \\
\mathit{Expression} &\to \mathit{id} \\
\mathit{Expression} &\to \mathit{cons} \\
\mathit{ProjectionItem} &\to (A \mapsto B)
\end{aligned}
$$

Figure 5.3. Productions for CNP grammar, where + signifies *one or more*, A, B, and N stand for argument names, and C a constant term.

We have explained the general features and the syntax of CNP. In the next section we will describe the denotational semantics of the language.

5.3 Denotational semantics

In this section we give the denotations of the CNP language constructs, including elementary predicates; the projection operator; basic logic operators and finally the extended logic operators. These denotations differ from ordinary relational denotations, since CNP predicate expressions contain argument names. Ordinarily, a relational extension would be defined as a set of tuples, each consisting of a sequence of terms. The terms are identified by their positions, as in $\langle t_1, \ldots, t_n \rangle$. This strict definition of a relational extension is not sufficient for expressing the denotations in CNP, as it is necessary to establish a flow of argument names between operators. For this reason, we define a new kind of relational extension we will call an α-extension.

Similarly to an ordinary relational extension, an α-extension consists of a set of α-tuples, akin to a *record* found in *Pascal* and many other languages. An α-tuple slightly differs from an ordinary tuple in the way that the elements of an α-tuple are located by a *name*: a first-order term that consists of an alphanumeric character sequence. Such names are purely *data*, or *constants*, and do not possess a lexical scope. The

operators of CNP are defined over α-extensions, and they provide the only way to modify and bind arguments of predicate expressions through the names embedded in the α-extension. Hence, the argument names are not variables and variable binding is not applicable to CNP expressions.

In the following sections, after formally defining α-extensions, we will describe the transformations between α-extensions and ordinary relational extensions, and show that these transformations form an isomorphism, thus establishing that α-extensions are isomorphic to ordinary relational extensions with respect to name maps that associate argument names to numeric argument positions. Finally, we will revisit some fundamental concepts of *Logic Programming* such as Herbrand interpretations and a fixpoint definition of CNP programs.

We shall make a general note about the notation. Making use of argument names in the equations necessitates an explicit separation between them and bound variables of a formula. When we use lower case identifiers, such as *as* or *b*, these represent argument names. These should be considered as constants. The identifiers that start upper case, such as *A* or *Xs* stand for the bound variables in a formula.

5.3.1 Semantics of α-extensions

Predicate expressions in CNP carry explicit names for arguments at every step of composition. The concept of argument names is analogous to labels in a record data type, or relational field names similar to the one in Codd's relational model in [25, 26][1], specifically to the concept of *domain-unordered relations* from [25] where the domains of a relation are referred to by names instead of positions. For writing denotations it is thus necessary to use a specific definition of a relational extension that is both compatible with the names in CNP's predicate expressions and also bijectively convertible to the relational extensions in first-order logic. We will call these name-bearing extensions α-extensions, short for *name-associative extensions*, denoted as $[\![E]\!]$ where E is a CNP predicate expression. The argument names associated with a predicate expression are determined by a *name map* (α), which will be defined in the next section. When it is useful to be explicit about the name map, we write it as a subscript to the expression, as in E_α.

In the equations in following sections, we refer to the Herbrand Universe of 'the' program (H). This refers to the Herbrand universe of a corresponding definite clause program as presented in Chapter 2. A CNP program is transformable to a COMBILOG program, which in turn is transformable to a definite clause program, which would contain the same terms, and hence has the same Herbrand Universe.

[1]Not to be confused with the concept of *Alpha expressions* defined in [26]

Conversions to and from α-extensions

In order to establish a formal definition of an α-extension, we need a tool for converting an ordinary tuple to an α-tuple. Consider the common form of an ordinary tuple with n elements:

$$t = \langle t_1, \ldots, t_n \rangle$$

The tuple t has the form of a sequence, which is equivalent to a function whose domain (K) is a continuous subset of natural numbers starting at 1, namely:

$$t : K \to H$$

where H is the Herbrand universe of the program. With terms $t_1, \ldots, t_n \in H$, t_i as a function can be written as:

$$t(i) = \begin{cases} t_1, & \text{if} \quad i = 1 \\ \ldots, & \ldots \\ t_n, & \text{if} \quad i = n \end{cases}$$

Based on this understanding of a tuple as a function from a set K of indices to a set H of terms, we can define an α-tuple as a function from a set A of alphabetic names to a set H of terms:

$$t_\alpha : A \to H$$

The conversion between the two forms of tuples can be established by a bijective function α, whose domain is the domain of the α-tuple, that is, A, and whose range is the domain of t, that is, K:

$$\alpha : A \to K$$

Since A is a set of unique names, and K is a continuous subset of natural numbers starting at 1, a bijective name map α can always be defined, given as many names as there are indices in K.

Definition 5.3.1. *A name map α is a bijective function from a set of names A to a continuous subset K of natural numbers starting at 1. A name map is compatible with a tuple t if the range of the name map is identical to the domain of the tuple, and it is compatible for an α-tuple t_α if the domain of the name map is identical to the domain of the α-tuple.*

A compatible name map α for a tuple $t : K \to H$ is a tool for converting it to an α-tuple, through function composition:

$$t_\alpha = t \circ \alpha$$

$$
\begin{aligned}
[\![\mathit{parentage}_\alpha]\!] = \Big\{ & \{\mathit{parent} \mapsto \text{`John'}, \mathit{child} \mapsto \text{`Mark'}\}, \\
& \{\mathit{parent} \mapsto \text{`Mary'}, \mathit{child} \mapsto \text{`Mark'}\}, \\
& \{\mathit{parent} \mapsto \text{`Nigel'}, \mathit{child} \mapsto \text{`Stefan'}\}, \\
& \{\mathit{parent} \mapsto \text{`Theodore'}, \mathit{child} \mapsto \text{`Stefan'}\} \Big\}
\end{aligned}
$$

where

$$\alpha = \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2\}$$

Figure 5.4. α-extension of the $\mathit{parentage}_\alpha$ predicate with two arguments: *parent* and *child*.

The reverse is also true, as the inverse of a name map is a tool for converting an α-tuple to an ordinary tuple:

$$
\begin{aligned}
t_\alpha \circ \alpha^{-1} &= t \circ \alpha \circ \alpha^{-1} \\
&= t \circ id_K \\
&= t
\end{aligned}
$$

This bijectivity is the basis for the isomorphism between ordinary extensions and α-extensions. An α-extension that is calculated by composing every tuple in an ordinary extension with a compatible name map is isomorphic to that ordinary extension.

Definition 5.3.2. *We define an α-tuple, shown as t_α, to be a set of name-value pairs where the names are unique. This qualifies an α-tuple as a functional relation.*

An example is given below, relating the name of a parent and the name of a child:

$$t_\alpha = \{\mathit{parent} \mapsto \text{`John'}, \mathit{child} \mapsto \text{`Mark'}\}$$

This definition of an α-tuple gives rise to the consequent definition of an α-extension, which is exemplified in Figure 5.4 for the *parentage* predicate.

The difference in the α-extension is that every α-tuple includes argument names. Also, the order in which argument name-value pairs appear is not important since α-tuple is defined as a set. The denotational difference between an ordinary extension is only the exchange of numeric argument positions with name positions. Yet, the implications are substantial for CNP, equipping it with a pipeline for name-aware CNP operators, allowing them to omit variables completely, while using a name-based notation.

We have presented the concept of α-extensions, which will be of use as we introduce the operators of CNP. We have also provided bijective transformation between α-extensions and ordinary relational extensions. In the following section, we will define constructs of CNP, starting with the elementary predicates.

5.3.2 Elementary predicates

We define four elementary predicates which are name-aware variants of their COMBILOG counterparts.

- The *true* predicate has no arguments, and is always true.
- The $const(A, C)$ predicate is true if its single argument named A has a value identical to C, where A and C are meta-variables in the formula above. In any valid CNP code, A would be an argument name, and C would be a constant term. Any predicate expression written in this way has a single α-tuple in its extension, which maps the given argument name A to the given constant term C.
- The binary identity predicate *id* is true if the a and b arguments have the identical value.
- The list construction predicate *cons* is true if the value of the ab argument is the result of applying the list construction function $\cdot$ to values of the arguments a and b.

The denotations of elementary predicates in CNP, given as α-extensions can be observed in Figure 5.5. Note that the definition of the elementary predicate *true* has $\emptyset$ for a name map, since it has no arguments. We have now defined the elementary predicates available in the language, therefore we can move on to defining the operators.

5.3.3 Projection operator

The projection operator is the primary tool for manipulating arguments of a predicate expression. It can eliminate or rename arguments from an expression, and establish argument bindings between a source expression S_α and the new predicate expression constructed by using *proj*.

The *proj* operator is written as $proj(S_\alpha, Ps)_{\alpha_c}$, where S_α is the source predicate expression, with an associated name map α, and Ps is a non-empty set of projection items. The resulting name map of the *proj* operation is α_c. Each projection item is in the form $A \mapsto B$, where A is

$$
\begin{aligned}
[\![\text{true}_{\emptyset}]\!] &= \{\{\}\} \\
[\![\text{const}(A, C)_{\alpha_1}]\!] &= \{\{A \mapsto C\}\} \\
[\![id_{\alpha_2}]\!] &= \{\{a \mapsto C, b \mapsto C\} \mid C \in H\} \\
[\![\text{cons}_{\alpha_3}]\!] &= \{\{a \mapsto X, b \mapsto Xs, ab \mapsto XXs\} \mid \\
&\qquad\qquad \langle X, Xs, XXs\rangle \in H^3 \wedge\ X \cdot Xs = XXs\}
\end{aligned}
$$

where

$$
\begin{aligned}
H &= \text{Herbrand universe of the program} \\
H^3 &= \text{Tertiary cartesian product of } H \\
\alpha_1 &= \{A \mapsto 1\} \\
\alpha_2 &= \{a \mapsto 1, b \mapsto 2\} \\
\alpha_3 &= \{a \mapsto 1, b \mapsto 2, ab \mapsto 3\}
\end{aligned}
$$

Figure 5.5. Denotations of CNP elementary predicates

an argument name and B is the new name for the argument. A rule of validity for the projection items is that any argument name from α can appear only once among the projection items as the first element A. The same rule of uniqueness applies also to any new name B, since it can only appear once as the second element of a projection item. This qualifies the projection items Ps as a bijective function $Ps : As \to Bs$. The domain of Ps is a non-empty subset of the names in the source expression S_α, or more specifically the domain of the name map α. With these restrictions in mind, we can write the denotation of the operator:

$$
\begin{gathered}
[\![\mathit{proj}\,(S_\alpha, Ps)_{\alpha_c}]\!] = \left\{t_\alpha \circ Ps^{-1} \mid t_\alpha \in [\![S_\alpha]\!]\right\} \\
\text{where } \alpha_c = \alpha \circ Ps^{-1}
\end{gathered}
$$

The resulting name map associated with application of *proj* is α_c, defined as the composition of the name map α and Ps^{-1}, which is undefined for argument names that do not appear in Ps as a new name. This guarantees that the arguments which are not explicitly projected are omitted in the new α-tuple. Since composition of two bijective functions is a bijection, it follows that α_c is also a bijection, and is also a valid name map.

As an example of how the *proj* operator produces a predicate expression from another source predicate expression, we can look at the α-extensions of the source and the result. Let us enumerate the α-extension of the

parentage predicate from earlier:

$$\begin{aligned}[\![\mathit{parentage}_{\alpha}]\!] = \Big\{ &\{\mathit{parent} \mapsto \text{‘John’}, \mathit{child} \mapsto \text{‘Mark’}\}, \\ &\{\mathit{parent} \mapsto \text{‘Mary’}, \mathit{child} \mapsto \text{‘Mark’}\}, \\ &\{\mathit{parent} \mapsto \text{‘Nigel’}, \mathit{child} \mapsto \text{‘Stefan’}\}, \\ &\{\mathit{parent} \mapsto \text{‘Theodore’}, \mathit{child} \mapsto \text{‘Stefan’}\}\Big\}\end{aligned}$$

where

$$\alpha = \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2\}$$

In order to define a new predicate *parentage2* using a projection of *parentage* by selecting the *parent* argument unmodified and renaming the *child* argument to *child-name*, then the instance of the *proj* operator should be written as:

$$\mathit{parentage2}_{\alpha_c} = \mathit{proj}(\mathit{parentage}, \{\mathit{parent} \mapsto \mathit{parent}, \mathit{child} \mapsto \mathit{child\text{-}name}\})_{\alpha_c}$$

since the projection in the example is

$$Ps = \{\mathit{parent} \mapsto \mathit{parent}, \mathit{child} \mapsto \mathit{child\text{-}name}\}$$

its inverse is defined as:

$$Ps^{-1} = \{\mathit{parent} \mapsto \mathit{parent}, \mathit{child\text{-}name} \mapsto \mathit{child}\}$$

and the name map of the *proj* operation is calculated as:

$$\alpha_c(A) = \alpha \circ Ps^{-1}(A)$$

or equivalently

$$\alpha_c(B) = \alpha(Ps^{-1}(B))$$

so that

$$\begin{aligned}\alpha_c(\mathit{parent}) &= \alpha(Ps^{-1}(\mathit{parent})) \\ &= \alpha(\mathit{parent}) \\ &= 1 \\ \alpha_c(\mathit{child\text{-}name}) &= \alpha(Ps^{-1}(\mathit{child\text{-}name})) \\ &= \alpha(\mathit{child}) \\ &= 2\end{aligned}$$

and therefore

$$\alpha_c = \{\mathit{parent} \mapsto 1, \mathit{child\text{-}name} \mapsto 2\}$$

and the resulting predicate $\mathit{parentage2}_{\alpha_c}$ have the following α-extension:

$$\begin{aligned}[\![\mathit{parentage2}_{\alpha_c}]\!] = \Big\{ &\{\mathit{parent} \mapsto \text{'John'}, \mathit{child\text{-}name} \mapsto \text{'Mark'}\},\\ &\{\mathit{parent} \mapsto \text{'Mary'}, \mathit{child\text{-}name} \mapsto \text{'Mark'}\},\\ &\{\mathit{parent} \mapsto \text{'Nigel'}, \mathit{child\text{-}name} \mapsto \text{'Stefan'}\},\\ &\{\mathit{parent} \mapsto \text{'Theodore'}, \mathit{child\text{-}name} \mapsto \text{'Stefan'}\}\Big\}\end{aligned}$$

5.3.4 Logic operators

CNP defines two logic operators: *ore* for disjunction, and *ande* for conjunction. The operators take two operands that are predicate expressions, and return another predicate expression. Both perform with the auto-expanding behaviour, and as a result, argument names in the resulting expression will be the union of argument names found in both of the operands. Here we will define binary versions of these operators, but it is trivial to devise the multiary versions in terms of the binary ones. The denotations of the binary auto-expanding logic operators are as follows:

$$\begin{aligned}[\![\mathit{ande}(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c}]\!] &= \{t_{\alpha_c} \in H_{\alpha_c} \mid (t_{\alpha_c} \supseteq t_{\alpha_1} \wedge t_{\alpha_c} \supseteq t_{\alpha_2}) \wedge\\ &\qquad t_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\alpha_2}]\!]\}\\ [\![\mathit{ore}(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c}]\!] &= \{t_{\alpha_c} \in H_{\alpha_c} \mid (t_{\alpha_c} \supseteq t_{\alpha_1} \vee t_{\alpha_c} \supseteq t_{\alpha_2}) \wedge\\ &\qquad t_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\alpha_2}]\!]\}\\ \text{where}\quad \alpha_1 &= \{a_1 \mapsto 1, \ldots, a_J \mapsto J\}\\ D_2 &= \mathit{Dom}(\alpha_2) - \mathit{Dom}(\alpha_1) = \{b_1, \ldots, b_K\}\\ \alpha_c &= \{a_1 \mapsto 1, \ldots, a_J \mapsto J, b_1 \mapsto J+1, \ldots, b_K \mapsto J+K\}\\ H^{J+K} &= (J+K)^{th} \text{ Cartesian product of Herbrand Universe}\\ H_{\alpha_c} &= \{U \circ \alpha_c \mid U \in H^{J+K}\}\end{aligned}$$

The name map α_1 contains J names, and the name map α_2 contains K names that are different to those in α_1, obtained through the set difference of domains of name maps, that is, $\mathit{Dom}(\alpha_2) - \mathit{Dom}(\alpha_1)$. As a result, the name map for the logic operation, that is, α_c, has $J+K$ names. H^{J+K} is a set of tuples each consisting of $J+K$ terms, each term being a member of the Herbrand Universe. By composing each member of H^{J+K} with α_c we obtain a set H_{α_c} of α-tuples, where each α-tuple maps a name from the domain of α_c to a member of the Herbrand Universe.

For an example, let us first quote the *parentage* predicate from earlier:

$$\begin{aligned}[\![\mathit{parentage}_{\alpha}]\!] = \Big\{ & \{\mathit{parent} \mapsto \text{`John'}, \mathit{child} \mapsto \text{`Mark'}\}, \\ & \{\mathit{parent} \mapsto \text{`Mary'}, \mathit{child} \mapsto \text{`Mark'}\}, \\ & \{\mathit{parent} \mapsto \text{`Nigel'}, \mathit{child} \mapsto \text{`Stefan'}\}, \\ & \{\mathit{parent} \mapsto \text{`Theodore'}, \mathit{child} \mapsto \text{`Stefan'}\}\Big\} \\ \text{where} \\ \alpha = \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2\}\end{aligned}$$

We shall also define a second predicate *teaching* which has two arguments, *child* and *teacher*:

$$\begin{aligned}[\![\mathit{teaching}_{\alpha}]\!] = \Big\{ & \{\mathit{teacher} \mapsto \text{`Dolores Umbridge'}, \mathit{child} \mapsto \text{`David'}\}, \\ & \{\mathit{teacher} \mapsto \text{`John Keating'}, \mathit{child} \mapsto \text{`Stefan'}\}\Big\} \\ \text{where} \\ \alpha = \{\mathit{teacher} \mapsto 1, \mathit{child} \mapsto 2\}\end{aligned}$$

and take the conjunction of the *parentage* predicate from earlier, and the new *teaching* predicate, using the *ande* operator:

$$\begin{aligned}\mathit{school\text{-}records}_{\alpha_c} &= \mathit{ande}(\mathit{parentage}_{\alpha_1}, \mathit{teaching}_{\alpha_2}) \\ \text{where} \\ \alpha_1 &= \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2\} \\ \alpha_2 &= \{\mathit{teacher} \mapsto 1, \mathit{child} \mapsto 2\}\end{aligned}$$

Let us calculate the denotation of this operation:

$$\begin{aligned}[\![\mathit{ande}(\mathit{parentage}_{\alpha_1}, \mathit{teaching}_{\alpha_2})_{\alpha_c}]\!] = \{t_{\alpha_c} \in H_{\alpha_c} \mid {} & (t_{\alpha_c} \supseteq t_{\alpha_1} \wedge t_{\alpha_c} \supseteq t_{\alpha_2}) \wedge \\ & t_{\alpha_1} \in [\![\mathit{parenting}_{\alpha_1}]\!] \wedge \\ & t_{\alpha_2} \in [\![\mathit{teaching}_{\alpha_2}]\!]\} \\ \text{where} \quad \alpha_1 &= \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2\}\ (J = 2) \\ D_2 &= \{\mathit{teacher}\}\ (K = 1) \\ \alpha_c &= \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2, \mathit{teacher} \mapsto 3\} \\ H^{J+K} &= H \times H \times H \\ H_{\alpha_c} &= \{U \circ \alpha_c \mid U \in H^{J+K}\}\end{aligned}$$

which results in the α-extension:

$$[\![\mathit{school\text{-}records}_{\alpha_c}]\!] =$$
$$\Big\{\{\mathit{parent} \mapsto \text{'Nigel'}, \mathit{child} \mapsto \text{'Stefan'}, \mathit{teacher} \mapsto \text{'John Keating'}\},$$
$$\{\mathit{parent} \mapsto \text{'Theodore'}, \mathit{child} \mapsto \text{'Stefan'}, \text{'teacher'} \mapsto \text{'John Keating'}\}\Big\}$$
$$\text{where}$$
$$\alpha = \{\mathit{parent} \mapsto 1, \mathit{child} \mapsto 2, \mathit{teacher} \mapsto 3\}$$

5.3.5 Recursion operators

The recursion operators are *foldr* and *foldl*, which are defined as a part of the CNP language. These are the counterparts to the recursion operators in COMBILOG, and operate the same way, except the addition of argument names.

The name maps for the resulting fold expressions are pre-determined, as well as the component predicates P and Q. As a result, any component predicate used with the folds has to comply with these pre-determined name maps. This is due to a design compromise for avoiding introduction of another higher-order argument (on top of P and Q) for indicating the roles of the arguments. Currently, argument *a0* refers to the initial value, argument *as* refers to the list, and *b* refers to the result of the folding. The associated name maps are as follows:

$$\alpha_{\mathit{foldr}} = \{a0 \mapsto 1, as \mapsto 2, b \mapsto 3\}$$
$$\alpha_P = \{a \mapsto 1, b \mapsto 2, ab \mapsto 3\}$$
$$\alpha_Q = \{a \mapsto 1, b \mapsto 2\}$$

The definitions of recursion operators do not precisely line up with their counterparts in other languages such as PROLOG or HASKELL. This is a result of the intentions behind the invention of COMBILOG, particularly the synthesis. The fold operators are designed to replicate the characteristic recursive predicate definitions iterating over lists in definite clause programs, which consist of a predicate for the recursive case (P) and a predicate for the base case (Q). This accommodates simpler predicate invention for the component predicates during program synthesis.

In their denotations below, the aggregate definitions *foldr* and *foldl* refer to the auxiliary definitions $\mathit{foldr}_0/\mathit{foldr}_{i+1}$ and $\mathit{foldl}_i/\mathit{foldl}_{i+1}$, respectively, where $i \geq 0$.

$$
\begin{aligned}
[\![foldr(P,Q)]\!] &= \bigcup_{i=0}^{\infty} [\![foldr_i(P,Q)]\!] \\
[\![foldr_0(P,Q)]\!] &= \{\{a0 \mapsto Y, as \mapsto [], b \mapsto Z\} \in H_{\alpha_f} \mid \\
&\qquad \{a \mapsto Y, b \mapsto Z\} \in [\![Q]\!]\} \\
[\![foldr_{i+1}(P,Q)]\!] &= \{\{a0 \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W\} \in H_{\alpha_f} \mid \\
&\qquad (\exists Z \in H \ s.t. \\
&\qquad\quad \{a0 \mapsto Y, as \mapsto Xs, b \mapsto Z\} \in [\![foldr_i(P,Q)]\!] \wedge \\
&\qquad\quad \{a \mapsto X, b \mapsto Z, ab \mapsto W\} \in [\![P]\!])\} \\
[\![foldl(P,Q)]\!] &= \bigcup_{i=0}^{\infty} [\![foldl_i(P,Q)]\!] \\
[\![foldl_0(P,Q)]\!] &= \{\{a0 \mapsto Y, as \mapsto [], b \mapsto Z\} \in H_{\alpha_f} \mid \\
&\qquad \{a \mapsto Y, b \mapsto Z\} \in [\![Q]\!]\} \\
[\![foldl_{i+1}(P,Q)]\!] &= \{\{a0 \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W\} \in H_{\alpha_f} \mid \\
&\qquad (\exists Z \in H \ s.t. \\
&\qquad\quad \{a \mapsto X, b \mapsto Y, ab \mapsto Z\} \in [\![P]\!] \wedge \\
&\qquad\quad \{a0 \mapsto Z, as \mapsto Xs, b \mapsto W\} \in [\![foldl_i(P,Q)]\!])\} \\
&\quad \textit{where} \\
&\quad H = \text{Herbrand universe of the program} \\
&\quad H^3 = \text{tertiary cartesian product of } H \\
&\quad H_{\alpha_f} = \{\{a0 \mapsto A_0, as \mapsto As, b \mapsto B\} \mid \langle A_0, As, B\rangle \in H^3\}
\end{aligned}
$$

There is a variant of the *foldr* operator, namely *foldr2* found in COMBILOG, which is also used here in Chapter 7 for program synthesis. This variant avoids the *foldr* argument *a0*, and obtains the initial value through a unary base case Q (as opposed to the binary one in *foldr*) with a single argument named b. Avoiding a redundant set-theoretic definition for *foldr2*, we can define it in terms of operators *foldr*, *ande*, and the elementary predicate *id*:

$$
\begin{aligned}
foldr2(P,Q) &= proj(foldr(P, ande(Q, id)), \{as \mapsto as, b \mapsto b\}) \\
&\quad \textit{where} \\
\alpha_{foldr2} &= \{as \mapsto 1, b \mapsto 2\} \\
\alpha_P &= \{a \mapsto 1, b \mapsto 2, ab \mapsto 3\} \\
\alpha_Q &= \{b \mapsto 1\}
\end{aligned}
$$

With constructs of CNP defined, in the next section we move on to describing the construction of a fixpoint semantics of a CNP program.

5.4 Fixpoint semantics

The meaning of a logic program is defined model-theoretically, as visited in Chapter 2. Here we extend the related concepts to CNP, in order to construct the meaning of CNP programs. Equivalence of a least fixpoint of a definite clause program and its model-theoretical meaning is an established result [120]. Through the earlier work presented in Chapter 2, the least fixpoint of a COMBILOG program is shown to coincide with that of a corresponding definite clause program. Here we extend this result by proving that the least fixpoint of a CNP program coincides with the fixpoint of a corresponding COMBILOG program, modulo introduction/removal of argument names. This result by extension connects the fixpoint semantics of CNP programs with definite clause programs.

Towards this result, we first define extension maps, the structure employed in COMBILOG as a counterpart to interpretations in First-order Logic. Using extension maps as a domain, we will construct an immediate consequence operator, and through it establish the fixpoint semantics as the least fixpoint of a power function that iterates the immediate consequence operator. Differently to COMBILOG, in CNP, extension maps relate predicate symbols to their α-extensions. For a program P_{cnp} with m predicate definitions, the structure of an extension map $\mathcal{E}$ is as follows:

$$\mathcal{E} = \bigcup_{i=1}^{m} \{p_i \mapsto e_{i\alpha}\}$$

where p_i stands for the i^{th} predicate symbol, and $e_{i\alpha}$ the α-extension related with that predicate.

Analogous to the COMBILOG extension maps, the lookup of the extension of a predicate p in an extension map $\mathcal{E}$ is given as:

$$[\![p]\!]_{\mathcal{E}} = \mathcal{E}(p)$$

The immediate consequence operator is a function that computes the direct logical consequence of a given extension map:

$$T^{cnp}_{P_{cnp}}(\mathcal{E}) = \bigcup_{i}^{m} \{p_i \mapsto [\![\varphi_i]\!]_{\mathcal{E}}\} \tag{5.1}$$

where p_i refers to the ith predicate symbol, φ_i to the ith predicate body, and $[\![\varphi]\!]_{\mathcal{E}}$ denotes the α-extension of the body φ with regard to the given

extension map $\mathcal{E}$. The extension of a compound expression is calculated using the denotations of the composition operators where the extensions of the components are looked up in the given extension map.

The immediate consequence operator yields only one step of implication. Therefore we define a power function that iterates the immediate consequence operator. In the base step, the initial extension map associates every predicate symbol in a program with the empty set. Then in each subsequent step, the new extension map grows by incorporating the newly discovered consequences.

The power function $\uparrow$ of the immediate consequence operator $T^{cnp}_{P_{cnp}}$ is given as:

$$\begin{aligned} T^{cnp}_{P_{cnp}} \uparrow 0 &= \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\} \\ T^{cnp}_{P_{cnp}} \uparrow (i+1) &= T^{cnp}_{P_{cnp}}(T^{cnp}_{P_{cnp}} \uparrow i) \\ T^{cnp}_{P_{cnp}} \uparrow \omega &= \bigcup_{j=1}^{m} \{p_j \mapsto \bigcup_{i=0}^{\infty} ((T^{cnp}_{P_{cnp}} \uparrow i)(p_j))\} \end{aligned}$$

Note that in the last equation, extensions for each predicate symbol in the program have to be looked up in each step separately, before their union is calculated. It follows that the fixpoint semantics of a CNP program P_{cnp} is given as:

$$T^{cnp}_{P_{cnp}} \uparrow \omega$$

In Chapter 2, we presented a similar treatment of fixpoint semantics of COMBILOG programs. The difference is that the extension maps in CNP map predicate symbols to α-extensions, while the COMBILOG extension maps map them to ordinary extensions as relations. For this reason, the fixpoint semantics of CNP programs has to be defined separately.

In the next section we look into the isomorphic equivalence of fixpoint semantics of a COMBILOG program and the corresponding CNP program.

5.4.1 Equivalence to Combilog programs

As described in the earlier section, the meaning of a CNP program is determined as the least fixpoint of the iterations of the immediate consequence operator through the power function $\uparrow$, which relies on the denotational semantics defined in the Section 5.3. In this section, we present the transformation steps that obtain a CNP program from a given COMBILOG program, and we show that the meaning of a COMBILOG program and the meaning of a corresponding CNP program obtained through the given transformation steps are isomorphic, modulo introduction or removal of

argument names, in the sense described under α-extensions in Section 5.3.1,

The recursion operators *foldr* and *foldl* and elementary predicates are excluded from this proof as their denotations are readily near-identical to their counterparts in COMBILOG.

In both languages, a program consists of predicate-identifier/body assignments. In COMBILOG, a predicate body conforms to a canonical form characterized by an outer disjunction, followed by *make* operations for each operand:

$$or(make([\ldots], P_1), \ldots, make([\ldots], P_\gamma))$$

where every P_i is a conjunction with β_i operands, each containing a *make* operator and a predicate identifier q_j:

$$\begin{aligned} P_1 &= and(make([\ldots], q_{1,1}), \ldots, make([\ldots], q_{1,\beta_i})) \\ \ldots &\quad \ldots \\ P_\gamma &= and(make([\ldots], q_{\gamma,1}), \ldots, make([\ldots], q_{\gamma,\beta_i})) \end{aligned}$$

The canonical form of a COMBILOG predicate body is given below. In the formula, the $\prod$ symbol is used as the multiary aggregation of the *and* operator, and the $\sum$ symbol is used as the multiary aggregation of the or operator. The number of operands to the *or* operator is γ, and the number of operands to the i^{th} *and* operator is β_i. The arity of each *and* operation is relevant to the formula, and is denoted by $\#_i$. Similarly, the arity of the single *or* operator is n. Operands of each *and* operator are denoted with a predicate identifier $q_{i,j}$, referring to the i^{th} *and* operations j^{th} operand. These operands will sometimes be referred to as *components*. The canonical form is denoted as follows:

$$p^n \leftarrow \sum_{i=1}^{\gamma} make\left([\sigma_{i,1}, \ldots, \sigma_{i,n}], \prod_{j=1}^{\beta_i} make\left([\pi_{i,j,1}, \ldots, \pi_{i,j,\#_i}], q_{i,j}\right)\right)$$

The integer indices denoted by $\sigma_{i,k}$ and $\pi_{i,j,k}$ are index lists for the *make* operator. The construction of the canonical form was discussed in Chapter 2 related to Definition 2.2.5.

The canonical form in COMBILOG is based on the standard structure of definite clauses, but it is not necessarily followed in CNP programs. CNP predicate definitions have a more relaxed structure, which can contain only a predicate identifier, a conjunction, a disjunction, or a combination of these that is not necessarily arranged in the COMBILOG canonical form. Due to compositionality, any CNP predicate body can easily be transformed to a form equivalent to the canonical form. For example, in order to introduce a conjunction instead of a single predicate identifier p, the *ande* operator can be used with the second operand as *true*,

constructing $ande(p, true)$, preserving the original meaning of p. Or, if the CNP syntax tree is deeper than the depth allowed for by the COMBILOG canonical form, parts of the expression can be assigned to different predicate identifiers, by applying a simplification transformation such as Tseitin transformations [117]. Thus, we base this section on the canonical form found in COMBILOG, assuming any CNP definition can be trivially transformed to the same form.

Before we follow on to the transformation steps, it is necessary to set some restrictions on the form of COMBILOG expressions. These restrictions have been conventional but not explicitly stated in the earlier COMBILOG publications.

Restrictions

Let us define the two restrictions that apply to the transformation steps.

Restriction 1 The index list of a *make* operator should not contain any duplicate indices. Uses such as $make([1, 2, 1])$ are not considered valid, because the index 1 appears twice. The alternative is to use the elementary *id* predicate.

Restriction 2 The restriction is regarding the use of indices that are higher than the arity of the source, such as $make([1, 2, 3], q^2)$, in a way that introduces an unbound argument in the same position in every component of a logic operator. Since the arity of q is 2, the third index 3 introduces an unbound argument. For example, in a conjunction, when every component introduces an unbound argument on the same index, it results in a newly introduced argument in the head, that is also unbound:

$$s \leftarrow and(make([1, 2, 3], q^2), make([1, 2, 3], r^1))$$

The third argument of s is not bound to values of arguments in either q or r. This is not a useful operation, and omitting it allows us to simplify the transformation steps. An alternative for establishing the same result is to project a single argument of the *id* predicate.

After having clarified the restrictions, we can move onto presenting the transformation steps.

Transformation steps

The following steps *nominal* define the transformation steps departing from a COMBILOG expression in the canonical form given above, to the corresponding CNP expression, so that:

$$E_{cnp} = nominal(E_{comb})$$

Step 1. The component predicates $q_{i,j}$ are made α-extensional by introducing name maps $\theta_{i,j}$, which cover the arities of $q_{i,j}$, by mapping uniquely chosen names $c_{i,j,1}, \ldots, c_{i,j,o_{i,j}}$ to argument indices $1, \ldots, o_{i,j}$, where $o_{i,j}$ is the arity of the component $q_{i,j}$:

$$\theta_{i,j} = \{c_{i,j,1} \mapsto 1, \ldots, c_{i,j,o_{i,j}} \mapsto o_{i,j}\}$$

The new name-aware components $q_{i,j_{\theta_{i,j}}}$ are extensionally isomorphic to COMBILOG components $q_{i,j}$, since they only introduce names to arguments. The *and* operators are also assigned name maps that assign names to their argument positions, in the same way, each denoted with η_i:

$$\eta_i = \{b_{i,1} \mapsto 1, \ldots, b_{i,\#_i} \mapsto \#_i\}$$

The names $b_{i,k}$ can be arbitrarily assigned for the purposes of transformation, but practically they would be automatically calculated by the *ande* operation (to be introduced in Step 3) as a union of all argument names in all inner *proj* operations. Similarly, a name map α with uniquely chosen names $a_1, \ldots, a_n$ is assigned to the whole expression E_{comb}, where n is the number of arguments in E_{comb}, and α is:

$$\alpha = \{a_1 \mapsto 1, \ldots, a_n \mapsto n\}$$

As a result, the component predicates are replaced with their name-aware CNP counterparts:

$$q_{i,j} \Rightarrow q_{i,j_{\theta_{i,j}}}$$

Note that due to the extensional isomorphism between ordinary extensions and α-extensions, discussed under Section 5.3.1 and since the name maps assigned are compatible and valid, the following extensional isomorphisms ($\cong$) are known between every COMBILOG component $q_{i,j}$ and the corresponding CNP component $q_{i,j_{\theta_{i,j}}}$:

$$[\![q_{i,j}]\!] \cong \left[\!\!\left[q_{i,j_{\theta_{i,j}}}\right]\!\!\right]$$

Step 2. The *make* operators are replaced with *proj* operators. The index lists are replaced with projection sets by using inverses of name maps $\theta_{i,j}$ and α, denoted respectively as $\theta_{i,j}^{-1}$, and α^{-1}. The indices that are higher than the arity of the source component ($\pi_{i,j,k} > o_{i,j}$ and $\sigma_{i,k} > \#_i$) are omitted. The only use of such indices is introducing unbound arguments for aligning argument indices with other components (expanding). This is not a necessity in CNP. We shall refer to the projections of each component $q_{i,j}$ as *inner projection* i, j, and to the projection of each conjunction as *outer projection* i, j. The cardinality of projection sets are denoted as $f_{i,j}$ (inner projection i, j), and g_i (outer *projection* i), where

$f_{i,j} \leq \#_i$ and $g_i \leq n$. The projection sets $\{v_{i,j,1}, \ldots, v_{i,j,f_{i,j}}\}$ (of inner projection i, j) and $\{u_{i,1}, \ldots, u_{i,g_i}\}$ (of outer projection i) are defined as follows, in relation to the respective index lists $\{\pi_{i,j,1}, \ldots, \pi_{i,j,\#_i}\}$ and $\{\sigma_{i,1}, \ldots, \sigma_{i,n}\}$ of the *make* operators. For every projection of a component $q_{i,j}$:

$$\{v_{i,j,1}, \ldots, v_{i,j,f_{i,j}}\} = \{\theta_{i,j}^{-1}(\pi_{i,j,x}) \mapsto \eta_i^{-1}(x) \mid 1 \leq x \leq \#_i \wedge \pi_{i,j,x} \leq o_{i,j}\}$$

and for every projection of a conjunction indexed i:

$$\{u_{i,1}, \ldots, u_{i,g_i}\} = \{\eta_i^{-1}(\sigma_{1,x}) \mapsto \alpha^{-1}(x) \mid 1 \leq x \leq n \wedge \sigma_{1,x} \leq \#_i\}$$

For example, an index list such as the one in $make([3,1,2], q_{1,1}^3)$ is replaced with a projection set $\{c_{1,1,3} \mapsto b_{1,1}, c_{1,1,1} \mapsto b_{1,2}, c_{1,1,2} \mapsto b_{1,3}\}$, mapping names from $\theta_{1,1}$ to names from η_1. If the component predicate is binary ($q_{1,1}^2$), since the indices higher than the arity will be dropped, the resulting projection set would be $\{c_{1,1,1} \mapsto b_{1,2}, c_{1,1,2} \mapsto b_{1,3}\}$. Note that any projection set obtained this way is bijective, due to the Restriction 1 and the uniqueness of the argument names. After this step, any *make* expression is replaced by a *proj* operation, where name maps $\delta_{i,j}$ and ϵ_i are name maps associated with the respective inner and outer projections, calculated as a result of the projected names.

$$make([\pi_{i,j,1}, \ldots, \pi_{i,j,\#_i}], q_{i,j}^{o_{i,j}})^{\#_i} \Rightarrow proj(q_{i,j_{\theta_{i,j}}}, \{v_{i,j,1}, \ldots, v_{i,j,f_{i,j}}\})_{\delta_{i,j}}$$

$$make([\sigma_{i,1}, \ldots, \sigma_{i,n}], and(\ldots)^{\#_i})^n \Rightarrow proj(q_{i\theta_i}, \{u_{i,1}, \ldots, u_{i,g_i}\})_{\epsilon_i}$$

Step 3. The logic operator *and* is replaced with *ande*, or similarly *or* with *ore*. The operands of *ande* are *proj* operations, since the *make* operations have been replaced in the previous step. The name maps associated with the logic operations have been already assigned by Step 1 as η_i for each conjunction i, and α for the disjunction.

When these translation steps are applied, the CNP expression that conforms to the canonical form is obtained, where the symbols $\bigwedge$ and $\bigvee$ are the multiary aggregations of the auto-expanding logic operators *ande* and *ore*:

$$p_\alpha \leftarrow \bigvee_{i=1}^{\gamma} proj\left(\bigwedge_{j=1}^{\beta_i} proj\left(q_{i,j_{\theta_{i,j}}}, \{v_{i,j,1}, \ldots, v_{i,j,f_{i,j}}\}\right), \{u_{i,1}, \ldots, u_{i,g_i}\}\right)$$

In the following section we present the theorem of extensional isomorphism between COMBILOG and CNP programs.

Theorem of extensional isomorphism

Theorem 5.4.1: A COMBILOG program P and a CNP program P_N obtained through the transformation steps *nominal* are model-theoretically isomorphic ($\cong$), and equivalent modulo introduction/removal of argument names.

Proof: The model-theoretical equivalence is established through the least fixpoint of the power function of the immediate consequence operator. Inductively constructed, the induction base corresponds to the equality of the base cases of the power functions, which are equal by definition:

$$T_P^{comb} \uparrow 0 = \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\}$$

$$T_{P_{cnp}}^{cnp} \uparrow 0 = \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\}$$

The induction step is stated as:

$$\left(T_P^{comb} \uparrow i \cong T_{P_{cnp}}^{cnp} \uparrow i\right) \Rightarrow \left(T_P^{comb} \uparrow (i+1) \cong T_{P_{cnp}}^{cnp} \uparrow (i+1)\right)$$

which can be proven through establishing that given equivalent extension maps ($\mathcal{E}$ for a COMBILOG extension map, and $\mathcal{E}_\alpha$ for a CNP extension map), the corresponding immediate consequence operators will yield the equivalent deductions in each step:

$$(\mathcal{E} \cong \mathcal{E}_\alpha) \Rightarrow \left(T_P^{comb}(\mathcal{E}) \cong T_{P_{cnp}}^{cnp}(\mathcal{E}_\alpha)\right)$$

The implication follows through definitions of immediate consequence operators of COMBILOG found in Chapter 2 and the CNP counterpart from the Formula (5.1), given that the transformation steps *nominal* preserve the model of any predicate body in the canonical normal form. Even if the models of individual operators are not preserved identically, preservation of the model of any predicate body in the canonical normal form is sufficient, assuming every predicate body is in this form. The canonical form in COMBILOG contains nested operators, in the order *or*, *make*, *and*, *make*, and the component *q*. The form is two nested applications of what we shall refer to as *half-canonical form*, consisting of either *or*, *make*, *q*, or *and*, *make*, *q*. In Lemma 5.4.1.a, we show that any COMBILOG expression in the half-canonical form, with two components, when transformed to a CNP expression using the transformation steps *nominal*, preserves its model modulo introduction or removal of argument names. The lemma generalizes to the multiary form, where the logic operator has more than two arguments. It follows due to compositionality, if the half-canonical form preserves its model, its nested applications also preserve their model. Therefore, any predicate body in the canonical form, when transformed using the transformation steps *nominal*, preserves its model. □

Lemma 5.4.1.a: A COMBILOG half-canonical-form expression E_{comb}, and the corresponding CNP expression E_{cnp}, obtained by

$$E_{cnp} = nominal(E_{comb})$$

are given as using the *and* and *ande* as example operators:

$$E_{comb} = and(make([\pi_{1,1}, \ldots, \pi_{1,n}], q_1^{m_1})^n, make([\pi_{2,1}, \ldots, \pi_{2,n}], q_2^{m_2})^n)^n$$
$$E_{cnp} = ande(proj(q_{1\theta_1}, \{v_{1,1}, \ldots, v_{1,k}\})_{\delta_1}, proj(q_{2\theta_2}, \{v_{2,1}, \ldots, v_{2,l}\})_{\delta_2})_\alpha$$

where the arities and name maps are:

$$\begin{aligned}
m_1 &: \text{arity of } q_1 \\
m_2 &: \text{arity of } q_2 \\
n &: \text{arity of } E_{comb}, \text{and also of both } make\text{s} \\
\theta_1 &: \text{name map of } q_{1\theta_1} \\
\theta_2 &: \text{name map of } q_{2\theta_2} \\
\delta_1 &: \text{name map of } proj(q_{1\theta_1}, \{v_{1,1}, \ldots, v_{1,k}\})_{\delta_1} \\
\delta_2 &: \text{name map of } proj(q_{2\theta_2}, \{v_{2,1}, \ldots, v_{2,l}\})_{\delta_2} \\
\alpha &: \text{name map of } E_{cnp}
\end{aligned}$$

A CNP expression in the half-canonical-form obtained by the transformation steps *nominal* is extensionally isomorphic modulo introduction of argument names ($\cong$) to the corresponding COMBILOG expression:

$$t \in [\![E_{comb}]\!] \quad \textit{iff} \quad t_\alpha \in [\![E_{cnp}]\!]$$

Proof of lemma 5.4.1.a: We shall construct a proof by step-by-step unfolding every operator in E_{comb} and obtaining the extension of the whole expression, performing the same process for E_{cnp} as well. Since the CNP expression is transformed from the COMBILOG one through the transformation steps *nominal*, the extensions of the original components are the same. By calculating the extensions of both expressions, at the end, we shall show that the resulting extensions only differ in that one is the ordinary extension, and the other is the α-extension for the same relation.

Assume the tuples r and s in the extensions of $q_1^{m_1}$, and $q_2^{m_2}$:

$$\langle r_1, \ldots, r_{m_1} \rangle \in [\![q_1^{m_1}]\!]$$
$$\langle s_1, \ldots, s_{m_2} \rangle \in [\![q_2^{m_2}]\!]$$

due to the denotation of the *make* operator, the extensions of the *make*s are calculated as:

$$[\![make([\pi_{1,1}, \ldots, \pi_{1,n}], q_1^{m_1})^n]\!] = \{\langle r_{\pi_{1,1}}, \ldots, r_{\pi_{1,n}} \rangle \in H^n \mid \langle r_1, \ldots, r_{m_1} \rangle \in [\![q_1]\!]\}$$
$$[\![make([\pi_{2,1}, \ldots, \pi_{2,n}], q_2^{m_2})^n]\!] = \{\langle s_{\pi_{2,1}}, \ldots, s_{\pi_{2,n}} \rangle \in H^n \mid \langle s_1, \ldots, s_{m_2} \rangle \in [\![q_2]\!]\}$$

through which we obtain the tuples:

$$\langle r_{\pi_{1,1}}, \ldots, r_{\pi_{1,n}} \rangle \in [\![make([\pi_{1,1}, \ldots, \pi_{1,n}], q_1^{m_1})^n]\!]$$
$$\langle s_{\pi_{2,1}}, \ldots, s_{\pi_{2,n}} \rangle \in [\![make([\pi_{2,1}, \ldots, \pi_{2,n}], q_2^{m_2})^n]\!]$$

and through the *and* operator we obtain:

$$\begin{aligned}&[\![and(make([\pi_{1,1},\ldots,\pi_{1,n}],q_1^{m_1})^n, make([\pi_{2,1},\ldots,\pi_{2,n}],q_2^{m_2})^n)^n]\!]\\&= [\![make([\pi_{1,1},\ldots,\pi_{1,n}],q_1^{m_1})^n]\!] \cap [\![make([\pi_{2,1},\ldots,\pi_{2,n}],q_2^{m_2})^n]\!]\end{aligned}$$

that if the two tuples r' and s' are equal, then there is a tuple t:

$$\langle r_{\pi_{1,1}},\ldots,r_{\pi_{1,n}}\rangle = \langle s_{\pi_{2,1}},\ldots,s_{\pi_{2,n}}\rangle \Rightarrow t \in [\![E_{comb}]\!]$$

where

$$\langle t_1,\ldots,t_n\rangle = \langle r_{\pi_{1,1}},\ldots,r_{\pi_{1,n}}\rangle = \langle s_{\pi_{2,1}},\ldots,s_{\pi_{2,n}}\rangle$$

which, in relation to the original tuples r and s of the components, can be written as a function:

$$t(i) = \begin{cases} r_{\pi_{1,i}} & \text{if } \pi_{1,i} \le m_1 \wedge \pi_{2,i} \le m_2 \wedge r_{\pi_{1,i}} = s_{\pi_{2,i}} \\ r_{\pi_{1,i}} & \text{if } \pi_{1,i} \le m_1 \wedge \pi_{2,i} > m_2 \\ s_{\pi_{2,i}} & \text{if } \pi_{2,i} \le m_2 \wedge \pi_{1,i} > m_1 \end{cases} \tag{5.2}$$

where a case such as $\pi_{2,i} > m_2 \wedge \pi_{1,i} > m_1$ is not possible due to Restriction 2.

Next, let us consider the α-tuples of CNP component predicates introduced by Step 1 of the *nominal*:

$$\begin{aligned}&r_{\theta_1} \in [\![q_{1\theta_1}]\!]\\&s_{\theta_2} \in [\![q_{2\theta_2}]\!]\\&r_{\theta_1} = \{c_1 \mapsto r_1,\ldots,c_{m_1} \mapsto r_{m_1}\}\\&s_{\theta_2} = \{e_1 \mapsto s_1,\ldots,e_{m_2} \mapsto s_{m_2}\}\end{aligned}$$

and let us remember the *proj* operator:

$$[\![proj\,(S_\alpha, Ps)_{\alpha_c}]\!] = \left\{\tau_\alpha \circ Ps^{-1} \mid \tau_\alpha \in [\![S_\alpha]\!]\right\}$$
$$\text{where } \alpha_c = \alpha \circ Ps^{-1}$$

the two *proj* operations written by the Step 2 of *nominal*:

$$\begin{aligned}&proj(q_{1\theta_1}, \{\theta_1^{-1}(\pi_{1,i}) \mapsto \alpha^{-1}(i) \mid \pi_{1,i} \le m_1\})_{\delta_1}\\&proj(q_{2\theta_2}, \{\theta_2^{-1}(\pi_{2,i}) \mapsto \alpha^{-1}(i) \mid \pi_{2,i} \le m_2\})_{\delta_2}\end{aligned}$$

Note that since θ_1, θ_2, and α are valid name maps, hence bijective, we can invert the name maps:

$$\begin{aligned}\theta_1^{-1} &= \{1 \mapsto c_1,\ldots,m_1 \mapsto c_{m_1}\}\\\theta_2^{-1} &= \{1 \mapsto e_2,\ldots,m_2 \mapsto e_{m_2}\}\\\alpha^{-1} &= \{1 \mapsto a_1,\ldots,n \mapsto a_n\}\end{aligned}$$

which would help to simplify the projection sets as follows:

$$proj(q_{1\theta_1}, \{c_{\pi_{1,i}} \mapsto a_i \mid \pi_{1,i} \leq m_1\})_{\delta_1}$$
$$proj(q_{2\theta_2}, \{e_{\pi_{2,i}} \mapsto a_i \mid \pi_{2,i} \leq m_2\})_{\delta_2}$$

using which, we can calculate the sets of names B and D in the name maps δ_1 and δ_2:

$$B = \{a_i \mid \pi_{1,i} \leq m_1\}$$
$$D = \{a_i \mid \pi_{2,i} \leq m_2\}$$

and the name maps δ_1 and δ_2 would be:

$$\delta_1 = \{b_1 \mapsto 1, \ldots, b_k \mapsto k\}$$
$$\delta_2 = \{d_1 \mapsto 1, \ldots, d_l \mapsto l\}$$

Note that $B \subseteq A$ and $D \subseteq A$, where A is the domain of the name map α. Using these *proj* operations, we can calculate the following α-tuples belonging to the extensions of these operations:

$$r_{\delta_1} \in [\![proj(q_{1\theta_1}, \{c_{\pi_{1,i}} \mapsto a_i \mid \pi_{1,i} \leq m_1\})_{\delta_1}]\!]$$
$$s_{\delta_2} \in [\![proj(q_{2\theta_2}, \{e_{\pi_{2,i}} \mapsto a_i \mid \pi_{2,i} \leq m_2\})_{\delta_2}]\!]$$
$$r_{\delta_1} = \{a_i \mapsto r_{\pi_{1,i}} \mid \pi_{1,i} \leq m_1\}$$
$$s_{\delta_2} = \{a_i \mapsto r_{\pi_{2,i}} \mid \pi_{2,i} \leq m_2\}$$

after which we can move on to the *ande* operator. Let us remember the operator:

$$[\![ande(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c}]\!] = \{t_{\alpha_c} \in H_{\alpha_c} \mid (t_{\alpha_c} \supseteq t_{\alpha_1} \wedge t_{\alpha_c} \supseteq t_{\alpha_2}) \wedge t_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\alpha_2}]\!]\} \quad t_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\alpha_2}]\!]\}$$

where $\alpha_1 = \{a_1 \mapsto 1, \ldots, a_J \mapsto J\}$

$D_2 = Dom(\alpha_2) - Dom(\alpha_1) = \{b_1, \ldots, b_K\}$

$\alpha_c = \{a_1 \mapsto 1, \ldots, a_J \mapsto J, b_1 \mapsto J+1, \ldots, b_K \mapsto J+K\}$

$H^{J+K} = (J+K)^{th}$ Cartesian product of Herbrand Universe

$H_{\alpha_c} = \{U \circ \alpha_c \mid U \in H^{J+K}\}$

which would instantiate to the following, where R_{δ_1} and S_{δ_2} are the first and the second *proj* operators, respectively.

$$\begin{aligned}
[\![ande(R_{\delta_1}, S_{\delta_2})_\alpha]\!] = \{t_\alpha \in H_\alpha \mid (t_\alpha \supseteq t_{\alpha_1} \wedge t_\alpha \supseteq t_{\alpha_2}) \wedge \\
t_{\alpha_1} \in [\![R_{\delta_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\delta_2}]\!]\} \\
t_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge t_{\alpha_2} \in [\![S_{\alpha_2}]\!]\} \\
\text{where} \quad \delta_1 = \{b_1 \mapsto 1, \ldots, b_k \mapsto k\} \\
D_2 = \{d_1, \ldots, d_l\} - \{b_1, \ldots, b_k\} \\
\alpha = \{a_1 \mapsto 1, a_n \mapsto n\} \\
H^n = n^{th} \text{ Cartesian product of the Herbrand Universe} \\
H_\alpha = \{U \circ \alpha \mid U \in H^n\}
\end{aligned}$$

through which we can say, if there is a tuple $t_\alpha \in H_\alpha$ such that $t_\alpha \supseteq r_{\delta_1}$ and $t_\alpha \supseteq s_{\delta_2}$, it would map all the names in the domain of α to terms in r_{δ_1} and s_{δ_2}, which we can write as $t_\alpha = \{a_1 \mapsto t_1, \ldots, a_n \mapsto t_n\}$. The tuple belongs to the extension of the *ande* operation:

$$t_\alpha \in [\![ande(proj(q_{1\theta_1}, \{v_{1,1}, \ldots, v_{2,k}\})_{\delta_1}, proj(q_{2\theta_2}, \{v_{2,1}, \ldots, v_{2,l}\})_{\delta_2})_\alpha]\!]$$

where t_α is defined as follows, where $1 \leq i \leq n$:

$$t_\alpha(a_i) = \begin{cases} r_{\delta_1}(a_i) & \text{if } a_i \in B \wedge a_i \in D \wedge r_{\delta_1}(a_i) = s_{\delta_2}(a_i) \\ r_{\delta_1}(a_i) & \text{if } a_i \in B \wedge a_i \notin D \\ s_{\delta_2}(a_i) & \text{if } a_i \in D \wedge a_i \notin B \end{cases} \tag{5.3}$$

which is the conditional definition of the α-tuple $t_\alpha \in E_{cnp}$. We shall explain how this is isomorphic to the original ordinary tuple t. Through the definitions of name sets B and D above, we know that:

$$\begin{aligned}
a_i \in B &\iff \pi_{1,i} \leq m_1 \\
a_i \in D &\iff \pi_{2,i} \leq m_2
\end{aligned}$$

and, through the definition of r_{δ_1} and s_{δ_2}, we know that

$$\begin{aligned}
\pi_{1,i} \leq m_1 &\Rightarrow r_{\delta_1}(a_i) = r_{\pi_{1,i}} \\
\pi_{2,i} \leq m_2 &\Rightarrow s_{\delta_2}(a_i) = s_{\pi_{2,i}}
\end{aligned}$$

therefore, we can replace the conditions $a_i \in B$ with $\pi_{1,i} \leq m_1$, and under the new condition $\pi_{1,i} \leq m_1$, we can replace $r_{\delta_1}(a_i)$ with $r_{\pi_{1,i}}$. Similarly, we can replace the condition $a_i \in D$ with $\pi_{2,i} \leq m_2$, and under the new condition $\pi_{2,i} \leq m_2$, we can replace $s_{\delta_2}(a_i)$ with $s_{\pi_{2,i}}$, resulting with the definition:

$$t_\alpha(a_i) = \begin{cases} r_{\pi_{1,i}} & \text{if } \pi_{1,i} \leq m_1 \wedge \pi_{2,i} \leq m_2 \wedge r_{\pi_{1,i}} = s_{\pi_{2,i}} \\ r_{\pi_{1,i}} & \text{if } \pi_{1,i} \leq m_1 \wedge \pi_{2,i} > m_2 \\ s_{\pi_{2,i}} & \text{if } \pi_{2,i} \leq m_2 \wedge \pi_{1,i} > m_1 \end{cases}$$

which is the name-aware instance of the original COMBILOG tuple established earlier:

$$t(i) = \begin{cases} r_{\pi_{1,i}} & \text{if } \pi_{1,i} \leq m_1 \wedge \pi_{2,i} \leq m_2 \wedge r_{\pi_{1,i}} = s_{\pi_{2,i}} \\ r_{\pi_{1,i}} & \text{if } \pi_{1,i} \leq m_1 \wedge \pi_{2,i} > m_2 \\ s_{\pi_{2,i}} & \text{if } \pi_{2,i} \leq m_2 \wedge \pi_{1,i} > m_1 \end{cases}$$

The isomorphism between t_α and t can be observed here, with the only difference being in their domains, one mapping names a_i to terms in tuples r and s, while the other maps the indices i to the same terms, and we know, through the Step 1 of *nominal*, that the names in a_i correspond bijectively to indices i. This result would also be true for an expression in the half-canonical form using the *or* and *ore* operators, albeit the isomorphic tuple definitions would be different than the *and*/*ande* case presented here.

A point that needs to be clarified is regarding expansion. When one of the *make* s in the COMBILOG half-canonical-form introduce an unbound argument, expanding the component predicate, the corresponding *proj* operator omits projecting that argument. But due to the Restriction 2, each argument in the E_{comb} has to be bound to the value of an argument in at least one component predicate. This makes sure that, for any index i, if $\pi_{1,i} > m_1$, then necessarily $\pi_{2,i} \leq m_2$, and similarly, if $\pi_{2,i} > m_2$, then necessarily $\pi_{1,i} \leq m_1$. As a result, at least one of the projection sets will contain a projection from an argument name in the component predicates to a_i. This makes sure that even though some indices in the *make* index lists are dropped by the *proj* operators, the final E_{cnp} half-canonical-expression will contain as many arguments as the original E_{comb}. □

5.5 Execution of CNP programs

CNP programs are executed the same way as COMBILOG programs, through a meta-interpreter written in PROLOG, which is implemented in line with the semantics of CNP presented earlier.

The meta-interpreter mainly consists of the *cnp* predicate, which has two arguments: the first argument is a complex term that is the predicate expression in the object language (CNP), and the second term is a list that contains the object language arguments. The *cnp* predicate represents the application of the predicate expression to the given arguments, and only succeeds when the given CNP predicate expression succeeds for the given arguments. As with the COMBILOG meta-interpreter discussed earlier in Chapter 2, this implementation is in line with the earlier work on implementing higher-order constructs in logic programming [121]. The CNP meta-interpreter is given in Appendix C.

As an example, let us observe the implementation of the elementary predicates, given below. Note that the lower-case argument names (a, b, and ab) are simple constant terms (atoms), that represent the argument names embedded in the α-extensions, while the upper-case X and Y are PROLOG variables.

$$
\begin{aligned}
&cnp(true, []).\\
&cnp(const(A, C), [A : C]).\\
&cnp(id, [a : X, b : X]).\\
&cnp(cons, [a : X, b : Y, ab : [X|Y]]).
\end{aligned}
$$

The implementation also contains a minimal set of library predicates:

$$
\begin{aligned}
&cnp(isNil, [nil : []]).\\
&cnp(gt, [a : X, b : Y]) \leftarrow X > Y.\\
&cnp(gte, [a : X, b : Y]) \leftarrow X \geq Y.
\end{aligned}
$$

The operators in CNP are represented separate clauses of the *cnp* predicate. As an example, let us observe the implementation of the *foldr* operator:

$$
\begin{aligned}
&cnp(foldr(P, Q), [a0 : A0, as : [], b : B]) \leftarrow cnp(Q, [a : A0, b : B]).\\
&cnp(foldr(P, Q), [a0 : A0, as : [A|As], b : B]) \leftarrow\\
&\quad cnp(foldr(P, Q), [a0 : A0, as : As, b : Bmid]) \wedge\\
&\quad cnp(P, [a : A, b : Bmid, ab : B]).
\end{aligned}
$$

The logic operators *ande* and *ore* are defined up to 5 operands. The list recursion operators *foldr* and *foldl* are defined, as demonstrated above, as well as some variants of the *foldr* operator, that are *foldr2* and *natrec*. One variant of the *foldl* operator is also included, that is the *filter* operator. A dynamic *defPredicate* predicate is auxiliary, and maps predicate identifiers to bodies, for facilitating predicate lookup.

5.6 Summary

In this chapter, we revisited the issues identified in Chapter 3, and suggested improvements which can be summarized as using unordered names instead of argument positions, and moving some functionality (*expanding*) to the logic operators. In this manner, we established a new language, *Combilog with Named Projection* (CNP).

We defined the structure of CNP programs and presented the denotational semantics of language constructs, introducing a new sort of relational extension, α-extensions, which is a result of using names for argument positions in a tuple. We showed that the conversion of an ordinary

relational extension to an α-extension is straightforward and bijective, using a compatible name map.

The auto-expanding logic operators *ore* and *ande* were possible due to the α-extensions. These enabled us to use arguments' names as a hint for establishing bindings without requiring a strict alignment of arguments as it is done in COMBILOG, which has been leading to boilerplate code, as noted in Chapter 3. Moreover, the operators *ore* and *ande* pave the way for a wider range of name-aware operators. One of these may be *oro* and *ando*, which semantically perform the same operations but project only the unbound (*outer*) arguments from both operands. Similarly, the pair *ori* and *andi* would project only the bound (*inner*) arguments. This approach may be extended to a *schema matching* technique, where natural linguistic reasoning on argument names and even ontologies are explored.

In Section 5.4.1, we described the transformation of COMBILOG expressions to CNP expressions through a canonical normal form. Exploiting the nested structure of the canonical normal form, we moved onto using a half-canonical-form in order to be able to write a shorter proof for equivalence of COMBILOG and CNP program meanings. Using the transformation steps and the denotations of CNP operators, we showed that for every behaviour of the operators, the CNP program will result in an equivalent meaning, modulo the introduction of argument names.

Finally we described the execution of CNP programs through a meta-interpreter written in PROLOG.

This chapter concludes the semantics of the CNP language. In the next chapter, we will discuss and evaluate the usability of the CNP notation, and in the following chapter, we will present an application of CNP for program synthesis, as a demonstration of the persisting compositionality principle.

6. Usability of CNP syntax

In the earlier chapters, we have identified the issues regarding the usability of COMBILOG's original syntax, and addressed these issues while developing a new notation we called CNP. In this chapter, we design an empirical approach to evaluate if CNP objectively compares to PROLOG in basic usability factors, namely comprehensibility and modifiability.

Our choice of reference language is PROLOG because it represents argument binding via variables, which seems to be a common practice in the larger family of declarative languages. Ultimately we would like to find out if a variable-free representation can be close to one with variables in terms of usability. Moreover, the semantics of PROLOG programs are in line with definite clause programs, much like COMBILOG and CNP. Also, PROLOG is one of the most established languages in the family of *Logic Programming* languages, followed by MERCURY [110].

There are many factors that determine the absolute usability of a programming language. The list starts with concerns such as availability of libraries, community, documentation, learning curve, range of problems that the language is suitable for, and more factors on the general day-to-day utilization of a programming language. Then follows the core concepts of usability regarding the textual representation of the language. This is our main concern in this study. We intend to look at the notation of CNP, particularly how it compares to PROLOG in two different ways. First, we aim to measure comprehensibility. This amounts to checking whether predicate definitions in CNP are easier or harder to read and understand than their counterparts in PROLOG. Second, modifiability, in which we measure whether CNP predicate definitions are easier to modify.

It is important to delineate the extent of the study. Usability of programming languages can depend on a wide range of factors, but here we shall focus on comprehensibility and modifiability only. This short list could have been extended to composition of new predicates which is an important programming technique, and would have made for a more thorough analysis of CNP, but the design and execution of such a study would prove cumbersome. It would have required teaching the whole language from scratch including its semantics to the level where participants can solve problems in the entirely new language.

In the next sections, we will present a design for a user study. Then, we will describe the participants, and present the results. Finally, we will discuss the results as conclusions of the study.

6.1 Overview of the study

There are two factors we would like the study to measure: the duration that it takes to complete tasks in CNP versus PROLOG, and the rate of mistakes programmers commit while completing these tasks. Together they provide a basic measurement of usability of programming language notation.

The study was conducted a test that included open-ended questions regarding pieces of code in each notation. The test began with an introduction (a manual) to the notation, followed with an example of a question in that notation, and then presented the actual questions. Each question included a relevant piece of code. The expected answers were fixed within reasonable tolerance such as typographic errors. The content of the test is delivered via an on-line forms service which collected the answers in a private database.

In order to achieve more power with fewer participants, a within-subjects method is adopted, as opposed to a between-subjects study. To every participant, we have presented two tests, one for each notation, so the results are less affected by variations between participants individual abilities. This method may lead to a distortion in the results in the form of a learning effect. When a participant reaches the second test (whichever notation it is), the common concepts between the two notations are learned already in the first. Hence, the second notation unavoidably has an advantage simply because it is taken later in the process. Counter-balancing has to be implemented to smooth out this effect. The participants are split into two experiment groups that take the tests in a different order, so the benefit of the learning effect is shared equally for the two notations.

Figure 6.1 displays a time plan of how the two experiment groups have taken the tests. To counter familiarity effects that may come into action, we have named the PROLOG notation as *Notation X*, and the CNP notation as *Notation Y*.

Order	Group A	Group B
1	Introduction to study	
2	Pre-test questions	
3	Introduction to Notation X	Introduction to Notation Y
4	Questions in Notation X	Questions in Notation Y
5	Introduction to Notation Y	Introduction to Notation X
6	Questions in Notation Y	Questions in Notation X
7	Post-test questions	

Figure 6.1. Time plan of the study for groups A & B

In the following sections, we will present the different parts of the test in the order they appear.

6.2 Introduction to study

In the first part of the process, we have provided participants with brief introduction to what the study intends to achieve without revealing any information about which notation is the one under development. The first snippet of information on the study was as follows:

> *"This is a brief study looking into two different notations for relational argument binding. We will present two different notations, and analyse small chunks of code in both in order to find out how they compare in ease of reading and editing. The duration of this test is around 25 minutes and is designed to be uninterrupted. If you expect interruptions (such as phone calls), we kindly ask you to take it at a later convenience."*

After the basic instructions on how to do the test, the participant was presented with the pre-test questions.

6.2.1 Pre-test questions

A few questions were positioned before the actual test began in order to collect demographic information about the participants and also to filter unsuitable participants.

The first questions were about the age of the participant. Then, we posed two questions:

- Do you have experience writing Prolog programs?
- Do you have any reading disorders that affects your ability to read formulas?

Answering *yes* to one of these questions disqualified participants from partaking. This was not made obvious to the participant initially, as it may have introduced an incentive to answer *no*.

Then, we have posed two other questions in order to collect demographic information:

- Do you have experience writing programs in other languages? If yes, please specify:
  (open answer)

- Would you consider yourself primarily a programmer or a researcher? (multiple choice: Programmer / Researcher / Neither)

The answers to pre-test questions were collected together with the test answers and stored in the database.

6.3 Usability questions

The main test consists of 6 questions, of which 3 were targeted to reading and interpreting the code in a given notation, and the other 3 were targeted to reading and also modifying code in a given notation towards a specific purpose. We shall now discuss both two groups of questions in detail.

6.3.1 Questions measuring comprehension

In the earlier chapters, we have identified that the main problem with COMBILOG syntax was keeping track of argument indices generated by the *make* operator. We have observed that when accompanied with multiple levels of composition operations, such as

$$make(some\ logic\ operator(make(\ldots)))$$

this effect is enhanced, and as a result renders the use of COMBILOG's original notation difficult. The purpose of the questions in this first section is to observe this effect. As a result, we intend to find out whether the improvements we have made on COMBILOG syntax while building CNP have made it relatively usable compared to PROLOG.

There are fundamentally two types of operations available in both notations. The first type is related to argument binding. This contains all the argument modification operations such as argument renaming, reordering, introducing, or removing. A second group of operations is making use of logic operators (*and*, or *or*) for composing a predicate definition using given predicates as components. When a predicate is given as an existing predicate without its implementation, we refer to it as a *given predicate*. If the program fragment deals with the definition of a predicate, that is, the predicate body, we refer to it as a *defined predicate* The three questions in this section combine a number of operations from these two groups, in the context of a predicate definition.

1. The first question exclusively involves a predicate definition that has only one operation which is argument mapping while logic operations are involved.

2. The second question involves argument mapping in the context of a logic operator that defines a new predicate. This question makes use of both groups of operations.
3. The third question involves two instances of argument mapping, and a logic operation in the definition of a predicate. One argument mapping occurs between the two component predicates, and another occurs between the user-defined predicate and the components.

The predicate definitions to be used for these questions need to have come from actual programs instead of abstract mock-ups. We have chosen a common example of logic programming to begin with, and picked out parts of it that corresponded to levels of complexity that we have described. The code is a fragment from a program that defines family relations.

In Figure 6.2 we show the program fragment in PROLOG which we based our questions on. The significance of this fragment is that while it includes predicate definitions in all three levels of complexity we require, it is a commonly understood and relatively short bit of code. In the code fragment, the first two predicates `parentage` and `isMale` are given as existing predicates. The following three predicate definitions `isParent`, `isFather` and `isGrandchild` are the three predicate definitions we will use for the questions.

The same program fragment, written in COMBILOG and CNP is given in Figures 6.3 and 6.4, respectively. The COMBILOG code is given here for continuity, it was not used as a part of the study.

Eliminating irrelevant artifacts in syntax

One of the questions in the design of this study is whether to use real examples directly or not. Use of actual bits of code reflecting real problems results in an immediate judgment on participants' side, based on the inherent meaning of the problem, not necessarily the code displayed. Participants' existing domain knowledge may influence their performance in program comprehension tasks. For example, in a program representing family relations, during reading the definition of an *isFather* predicate, the user immediately makes a judgment about the meaning of the predicate from the context, rather than the specifics of the definition. Then the user may proceed to answer the question using judgment instead of interpreting the syntax. This would have reduced the power of the study by reducing the observed effect.

Moreover, we would rather not use the same problem for both CNP and PROLOG. If the same problem is used, the learning effect carried over to the second section, whichever it is, clouds the measurement of results further, even though the study is counter-balanced.

```
parentage(Parent, Child) :- ...
isMale(Name) :- ...
isParent(Parent) :- parentage(Parent, _).
isFather(Name) :- parentage(Name, _), isMale(Name).
isGrandchild(Name) :- parentage(GParent, Parent),
                      parentage(Parent, Name).
```

Figure 6.2. Base program fragment written in PROLOG

```
parentage/2
isMale/1
isParent = make([1], parentage)
isFather = and(make([1], parentage), isMale)
isGrandchild = make([3], and(make([1,2,3], parentage),
                             make([3,1,2], parentage)))
```

Figure 6.3. Base program fragment written in COMBILOG

```
parentage :: {parent, child}
isMale :: {name}
isParent :: {parent} = parentage{parent}
isFather :: {name} = and(parentage{parent->name},
                         isMale{name})
isGrandchild :: {name} = and(parentage{parent->gparent,
                                       child->parent},
                             parentage{parent, child})
                           {child->name}
```

Figure 6.4. Base program fragment written in CNP

For these reasons, we have chosen to use obfuscated forms of the code for different notations. The underlying semantics will be exactly the same, but the actual code presented in the test will be different for each notation.

As an example, let us look at the `isParent` predicate definition in plain PROLOG:

```
isParent(Parent) :- parentage(Parent, _).
```

Instead of this plain form, we have used an alpha-renamed, obfuscated form of the code, stripped from its inherent meaning coming from the choice of variable and predicate names:

```
r(A) :- p(A, _).
```

Similarly, the plain CNP code for the same predicate, given here:

```
isParent :: {name} = parentage{parent->name}
```

is replaced with the renamed, obfuscated form, that is:

```
p :: {b} = g{a->b}
```

Transforming the program fragments in this way allowed us to conserve the complexity of the examples, while avoiding introducing a distortion in results. It also provided a way of generating different versions of the problem with the same complexity for different sections of the test.

Another issue is the existence of slightly different implementations of the same syntax constructs that we do not intend to measure. One of these is the choice of prefix or infix operator placement preferences, as in "and(a, b)" versus "a and b" (or "a, b"). For both notations, we transformed the codes so that prefix operators are used consistently.

Choice of the predicate definition assignment operator also fits in this category. For this, we consistently use the equality symbol "`=`". In PROLOG, multiple predicate definitions are used to show the *or* logical operator, but our examples do not use this logic operator, and they do not include multiple definitions for any predicate, which makes it possible to use the "`=`" equality sign instead of the "`:-`" implication sign. By this way we obtain uniformity among the notations in use of these symbols.

In most questions, there are some predicate definitions that are given, and the only relevant information about their implementation is the number of arguments, and names to be able to refer to these arguments. For example, in the base program fragment given in Figures 6.2, 6.3, and 6.4, the definitions for the `parentage` and *isMale* predicates are omitted. In order to convey their argument count and argument names, we have written these as universal relations. In the PROLOG version, these are written as `parentage(Parent, Child)` in order to be able to refer to the arguments of the predicate by the name of a variable. In the COMBILOG and CNP examples, these are simply given as signatures without a body. Semantically, writing these PROLOG predicates as universal predicates would be incorrect, but it is a trade-off decision that had to be taken for simplifying the questions. In the CNP fragments, these signatures are written as `k :: {x , y}` for a predicate `k` has two arguments, `x` and `y`. For consistency, these signatures are written in every predicate definition for the CNP fragments, including those who have a body.

The transformations that we have presented are necessary to isolate the effect of two argument mapping notations. The notations after these transformations are admittedly different than the originals, but we claim they provide a better measurement of our main goal that is improving

usability of argument bindings. Hence, from now on, we should refer to the modified version of the PROLOG syntax as *Notation X*, and the modified version of the CNP syntax as *Notation Y*.

The final alpha-renamed (obfuscated), transformed forms of the program fragments in both notations are given in Figure 6.5 and Figure 6.6, respectively.

```
p(A, B).
q(C).
r(A) = p(A, _).
s(D) = and(p(D, _), q(D)).
t(D) = and(p(_, A), p(A, D)).
```

Figure 6.5. Final transformed program fragment written in notation X

```
k :: {x, y}
m :: {z}
n :: {x} = k{x}
q :: {z} = and(k{x->z}, m{z})
r :: {z} = and(k{x->w, y->x}, k{x, y}) {y->z}
```

Figure 6.6. Final transformed program fragment written in notation Y

Comprehension tasks

We have presented the base code used in the first three questions of the study. Now we can discuss the actual questions posed to the participants, in the presence of this base code. The questions are formed around argument binding, which is the main problem we intend to solve in the context of this work. Hence, the questions are formed around these fundamental tasks.

- How many argument does a predicate have?
- What are the arguments of a defined predicate?
- What are the arguments of a given predicate?
- Is a given argument of the defined predicate bound to the value of an argument in one of the component predicates? If so, which one?
- Is a given argument of one of the component predicates bound to the value of an argument of the defined predicate? If so, which one?
- Is a given argument of one of the component predicates bound to the value of an argument of another component predicate? If so, which one?

For every question, only the relevant lines of the program fragments were presented to the participant. Next, we will look at Questions 1, 2, and 3, individually.

Question 1: Argument binding
The first comprehension question includes a simple task: mapping arguments of an existing predicate onto arguments of an existing predicate. For this task, we choose the two predicates `parentage` and `isParent` from the program fragment, given below:

```
parentage(Parent, Child).
isParent(Parent) :- parentage(Parent, _).
```

Similarly, in CNP, the same program fragment is written as:

```
parentage :: {parent, child}
isParent :: {parent} = parentage{parent}
```

which corresponds to the CNP code written in the strict notation:

$$parentage : \{parent, child\}$$
$$isParent \leftarrow proj(parentage, \{parent \mapsto parent\})$$

After transforming both program fragments into the notations X and Y, using the rules defined in Section 6.3.1, the actual Question 1 is presented in Figure 6.7.

Question in notation X	Question in notation Y

Consider this code in X:
`p(A, B).`
`r(C) = p(C, _).`
 	
a. How many arguments does p have?
b. How many arguments does r have?
c. What are the arguments of p?
d. What are the arguments of r?
	Consider this code in Y:
	`k :: {x, y}`
	`n :: {x} = k {x}`
	
	a. How many arguments does k have?
	b. How many arguments does n have?
	c. What are the arguments of k?
	d. What are the arguments of n?

Answers: a. 2 b. 1 c. A and B d. C	Answers: a. 2 b. 1 c. x and y d. x

Figure 6.7. Question 1

Question 2: Argument binding with a logic operation
The second question involves two given predicates `parentage` and `isMale`, and the predicate definition `isFather`. Composition of the `isFather` predicate includes an argument mapping between the exposed argument `Name` and arguments of components, and also an unbound argument. Let us look at the original PROLOG subprogram for this question.

```
parentage(Parent, Child).
isMale(Name).
isFather(Name) :- parentage(Name, _), isMale(Name).
```

Let us also see the untransformed CNP program fragment consisting of the lines involved in this question:

```
parentage :: {parent, child}
isMale :: {name}
isFather :: {name} = and(parentage{parent->name},
                         isMale{name})
```

which corresponds to the following CNP code in the strict notation:

$$\begin{aligned} parentage &: \{parent, child\} \\ isMale &: \{name\} \\ isFather &\leftarrow ande(proj(parentage, \{parent \mapsto name\}), \\ &\qquad proj(isMale, \{name \mapsto name\})) \end{aligned}$$

The code fragments after the transformations in Section 6.3.1 are given in Figure 6.8, along with the questions and the correct answers in both notations.
6.8.

Question 3: Two levels of argument binding with a logic operation
The last question of the comprehension section includes two levels of argument binding: first, a binding between the arguments of components in a conjunction, and second, a binding between an argument of a component and arguments of the predicate to be defined. The component predicates are `parentage` and `isGrandchild`. The PROLOG program including these two predicates is given below:

```
parentage(Parent, Child).
isGrandchild(Name) :- parentage(GParent, Parent),
                      parentage(Parent, Name).
```

Question in notation X	Question in notation Y

Consider this code in X:
`p(A, B).`
`q(C).`
`s(D) = and(p(D, _), q(D)).`
 	
a. What are the arguments of s?
b. Which argument(s) of s are bound to which argument(s) of p?
c. Which argument(s) of s are bound to which argument(s) of q?
d. p.B is bound to what other argument(s)?
	Consider this code in Y:
	`k :: {x, y}`
	`m :: {z}`
	`q :: {z} = and( k {x→z}, m {z} )`
	
	a. What are the argument(s) of q?
	b. Which argument(s) of q are bound to which argument(s) of k?
	c. Which argument(s) of q are bound to which argument(s) of m?
	d. k.y is bound to which argument of m?

Answers: a. D b. D to A c. D to C d. None	Answers: a. z b. z to x c. z to z d. None

Figure 6.8. Question 2

Similarly, the CNP program is:

```
parentage :: {parent, child}
isGrandchild :: {name} =
  and(parentage{parent, child},
      parentage{parent->gparent,
                child->parent}) {child->name}
```

which corresponds to the following CNP program in the strict notation:

$$\begin{aligned} &parentage : \{parent, child\} \\ &isGrandchild \leftarrow proj(ande(proj(parentage, \{parent \mapsto gparent, \\ &\qquad child \mapsto parent\}), \\ &\qquad proj(parentage, \{parent \mapsto parent, \\ &\qquad child \mapsto child\})), \\ &\qquad \{child \mapsto name\}) \end{aligned}$$

When both programs are transformed using rules we listed in section 6.3.1, we arrive at two program fragments in notation X and Y, which are presented below in the context of the actual question in the test in Figure 6.9.

6.3.2 Questions measuring modifiability

The second part of our survey focuses on use cases where a programmer needs to modify the notation in order to change the meaning of a program, or a part of a program. Among all the possible actions one can perform to modify a part of a program, those involved in this study are the ones relevant to composing arguments. The following 3 questions of the study aim to measure task time and difficulty of three common actions that heavily involve alterations to argument bindings.

In the questions in this part, we presented participants with an initial program fragment to begin with, and asked them to perform some specific modifications on this code for some given purpose. Similar to the earlier sections, the implementation of some parts of the programs are omitted, and these are given solely as signatures, as `flightRoute(A, B).` in PROLOG which is semantically would be a universal predicate (a predicate that's true for all given values), but here provided only as a means to convey the signature of a component predicate.

Now we will look at all three questions that involve modifications to argument bindings, and after that we will look at the actual questions used in the study.

Question in notation X	Question in notation Y

Consider this code in X, where p_1 and p_2 are two separate instances of the same predicate p. (assigned numbers so we can refer to them separately.)
 	
`p(A, B).`
`t(D) = and(`p_1`(_, F), `p_2`(F, D)).`
 	
When you consider the definition of predicate t,
 	
a. How many arguments does t have?
b. p_1.B is bound to which argument of t?
c. p_2.B is bound to which argument of t?
d. p_1.B is bound to which argument of p_2?
	Consider this code in Y, where k_1 and k_2 are two separate instances of the same predicate k. (assigned numbers so we can refer to them separately.)
	
	`k ::  {x, y}`
	`r ::  {z} = and(` k_1 `{x, y},`
	k_2 `{x→w, y→x} ) {y→z}`
	
	When you consider the definition of predicate r,
	
	a. How many arguments does r have?
	b. k_1.y is bound to which argument of r?
	c. k_2.x is bound to which argument of r?
	d. k_1.y is bound to which argument of k_2?

Answers: a. 1 b. None c. D d. A	Answers: a. 1 b. z c. None d. None

Figure 6.9. Question 3

Question 4: Extracting out a predicate definition

In the fourth question, a fragment including a predicate definition is provided. The predicate definition is intentionally long. The participant is asked to extract out parts of this definition as separate predicate definitions, and instead place calls by name to these new predicates in the body of the original predicate definition, effectively performing refactoring.

This task involves refactoring an expression, which in turn requires maintaining consistency of the argument mappings between the host expression and the one that is extracted out as a separate predicate.

The code fragment used in this question is a part of a path finding program. In PROLOG:

```
flightRoute(A, B).
trainRoute(A, B).
outRouteOpt(D) :- flightRoute(D, _),
                  trainRoute(D, _).
inRouteOpt(E) :- flightRoute(_, E),
                 trainRoute(_, E).
outInRouteOpt(D, E) :- outRouteOpt(D),
                       inRouteOpt(E).
```

The program above contains two given predicates, `flightRoute` and `trainRoute`, determining the flight and train routes available from a city `A` to a city `B`. The user-defined predicate `outRouteOpt(D)` determines that departing from city `D`, there are both flight and train options exist. Similarly, the user-defined predicate `inRouteOpt` determines that towards the city `E`, there are both flights and trains coming in. Consequently, the user-defined predicate composes a preliminary list of pairs of cities `D` and `E` where a passenger can depart from `D` with both a flight or a train, and can travel towards `E` with both flight or train, for the purpose of having a contingency plan. (Since it is not a complete program, it does not necessarily determine that a possible route exists between the two cities.) The same code in CNP can be written as:

```
flightRoute :: {x, y}
trainRoute :: {x, y}
outRouteOpt :: {a} = and(flightRoute {x->a},
                         trainRoute {x->a})
inRouteOpt :: {b} = and(flightRoute {y->b},
                        trainRoute {y->b})
outInRouteBoth :: {a, b} = and(outRouteOpt,
                               inRouteOpt)
```

The program fragment presented in the question is a modification of the code above. The definitions of `outRouteOpt` and `inRouteOpt` are inlined, resulting in the longer definition of `outInRouteOpt`. In PROLOG:

```
flightRoute(A, B).
trainRoute(A, B).
outInRouteOpt(D, E) :- flightRoute(D, _), trainRoute(D, _),
                       flightRoute(_, E), trainRoute(_, E).
```

and in CNP:

```
flightRoute :: {x, y}
trainRoute :: {x, y}
outInRouteBoth :: {a, b} =
  and(and(flightRoute {x->a}, trainRoute {x->a}),
      and(flightRoute {y->b}, trainRoute {y->b}))
```

which corresponds to the following CNP code in the strict form:

$$
\begin{aligned}
\mathit{flightRoute} &: \{x, y\} \\
\mathit{trainRoute} &: \{x, y\} \\
\mathit{outInRouteBoth} \leftarrow \mathit{ande}(&\mathit{ande}(\mathit{proj}(\mathit{flightRoute}, \{x \mapsto a\}), \\
&\quad \mathit{proj}(\mathit{trainRoute}, \{x \mapsto a\})), \\
&\mathit{ande}(\mathit{proj}(\mathit{flightRoute}, \{y \mapsto b\}), \\
&\quad \mathit{proj}(\mathit{trainRoute}, \{y \mapsto b\})))
\end{aligned}
$$

These two bits of code given above are the initial program fragment used in Question 4, before obfuscations. The questions for both notations after transformations given in Section 6.3.1 is given in Figure 6.10.

The code in notation X (Figure 6.10) question contains a predicate definition for predicate `s` that uses given predicates `p` and `r`. The two operands of the logic operator `and` are separate expressions consisting of logic operations and argument mappings. The answer in the figure contains the simplest form that the two new predicate definitions `s1` and `s2` can be written with no alpha renaming.

The code in notation Y (Figure 6.10) question also follows the same structure, asking the participant to refactor parts of `p` out. After extracting parts of p as separate predicate definitions, the correct answer in notation Y declares auxiliary predicates `p1` and `p2`, while calling both in the new definition of `p`. This is also the simplest form of code that accomplishes this task with minimum changes, with no alpha renaming. The variations with different argument names are also accepted as correct, as long as they are semantically equivalent.

<table>
<tr><td>Question 4 in (notation X)</td><td>Question 4 (notation Y)</td></tr>
<tr><td>Some predicate definitions may be too long. It is possible to take such definitions, and redefine parts of them in separate predicate definitions in order to use these smaller definitions. The idea is to define a complete predicate that is effectively the same as before. Consider the definition of s below:

<pre>p(A, B).
r(A, B).
s(D, E) = and(and(p(D, _), r(D, _)),
              and(p(_, E), r(_, E))).</pre>
How can we define two separate parts that compose s, and define s in terms of those two parts?</td><td>When some definitions are too long to be practical, parts of them can be extracted out as separate predicates, and given separate names. Predicate p given below is such an example in need of rewriting:

<pre>k :: {x, y}
m :: {x, y}
p :: {a, b} = and(and(k {x->a}, m {x->a}),
                  and(k {y->b}, m {y->b}))</pre>
Please write a new definition for p that uses separate definitions for its components.</td></tr>
<tr><td>Answer:

<pre>p(A, B).
r(A, B).
s1(D) = and(p(D, _), r(D, _)).
s2(E) = and(p(_, E), r(_, E)).
s(D, E) = and(s1(D), s2(E)).</pre></td><td>Answer:

<pre>k :: {x, y}
m :: {x, y}
p1 :: {a} = and(k {x->a}, m {x->a})
p2 :: {b} = and(k {y->b}, m {y->b})
p :: {a, b} = and(p1, p2)</pre></td></tr>
</table>

Figure 6.10. Question 4

Question 5: Inlining a predicate definition
This question contains the second task towards measuring modifiability of the notations. It involves replacing a call to a user-defined predicate, with its body in its definition, essentially performing the opposite task in question 4. The participant is given an initial fragment that includes a predicate definition containing a predicate call to another existing predicate. Then the participant is asked to replace the predicate call with its definition. This process involves rewriting the definition of a predicate while keeping the argument names in check so that an unintended binding or name capturing is avoided.

The base code in this question involves graphs, which are a common data structure in logic programming problems. Let us present the code first and discuss it subsequently. The code in question is written in PROLOG as follows, where `edge` is a binary predicate that succeeds if there is an edge connecting two given vertices:

```
edge(X, Y).
degreeTwo(Z).
trapCycleExists(V) :- edge(V, V2), edge(V2, V).
deadEndAhead(V) :- trapCycleExists(V), degreeTwo(V).
```

which translates to the CNP code:

```
edge :: {x, y}
degreeTwo :: {z}
trapCycleExists :: {v} = and(edge {x->v, y->z},
                             edge {x->z, y->v}) {v}
deadEndAhead :: {v} = and(trapCycleExists {v->z},
                          degreeTwo)
```

and which, in turn, corresponds to the CNP code given below in the strict notation:

$$
\begin{aligned}
\mathit{edge} &: \{x, y\} \\
\mathit{degreeTwo} &: \{z\} \\
\mathit{trapCycleExists} &\leftarrow \mathit{proj}(\mathit{ande}(\mathit{proj}(\mathit{edge}, \{x \mapsto v, y \mapsto z\}) \\
&\qquad\qquad \mathit{proj}(\mathit{edge}, \{x \mapsto z, y \mapsto v\})), \{v \mapsto v\} \\
\mathit{deadEndAhead} &\leftarrow \mathit{ande}(\mathit{proj}(\mathit{trapCycleExists}, \{v \mapsto z\}), \mathit{degreeTwo})
\end{aligned}
$$

In PROLOG the code above, the existing predicate `edge` determines if there is a directed path from a vertex `X` to a vertex `Y` in a graph. The second existing predicate `degreeTwo` determines if a given vertex `Z` is of degree 2, meaning has two edges connected to it. A cycle is a set of vertices beginning and ending at the same vertex. A trap cycle is a

cycle of length 2, practically forming a dead end for search purposes. The predicate `trapCycleExists` determines if a trap cycle exists at a given vertex `V`. The predicate `deadEndAhead` makes use of the predicate `trapCycleExists` and determines if the only cycle available at a vertex `V` is a trap cycle (since one of the 2 edges available must be the current location of the traverser).

In this program fragment, the participant is asked to replace the call to the predicate `trapCycleExists` with its definition.

The questions for both notations after obfuscations given in Section 6.3.1 are given in Figure 6.11.

For the code in notation X, the predicate s includes a call to predicate r, and the task is replacing this call with the definition of r. The task unavoidably requires alpha renaming, since simply replacing `r(B)` with `and(p(A, B), p(B, A))` would result in variable `B` in `s` name-capturing the encapsulated variable `B` in `r`. In order to avoid capture, the variable `B` in the definition of `r` needs to be given a fresh name (such as `C`). As a second step, the instances of variable `A` in `r` needs to be substituted with `B`. This solution is reflected in the given answer in the figure.

The given task for notation Y is identical to the one in X, which is modifying the definition of predicate `p`, replacing the call to predicate `n` with its definition. The participant can perform alpha renaming to rename arguments of `n` in order to compose an instance of it appropriate for the context it appears in the definition of `p`. A quicker solution is simply replacing the call to `n` with its definition, without modifying its arguments. Since the syntax of CNP (hence the syntax of notation Y) is purely compositional, and argument mappings are performed through projection, name-capturing is not an issue. This feature of the notation is not listed anywhere in the study, in order to avoid coaching towards this behavior. It is left to the participants to notice and use this feature, or perform alpha-renaming. Both solutions are accepted as correct as an answer.

Question 6: Debugging: correcting argument mismatch

The last question in this section is designed to measure participants success in noticing and correcting an argument binding which is faulty. Only for this question, the transformation steps listed in Section 6.3.1 does not include obfuscation of the predicate and argument names, since it would be impossible to hint at the error of argument mapping without giving the context. Because of this, it is necessary to use different program fragments for each notation. For both notations, the questions and answers are given in Figure 6.12.

For notation X, this code is a part of a program for deciding the color of a stone to be laid in a pavement. The rule that the program follows is that, the stones should be laid in alternating colors of white and black.

Question in notation X	Question in notation Y

There is more than one way to write some expressions. Consider this code in X:
 	
`p(X, Y).`
`q(Z).`
`r(A) = and(p(A, B), p(B, A)).`
`s(B) = and(r(B), q(B)).`
 	
How can we write effectively the same definition for `s`, without using `r`?
	It is possible to produce the same effect by writing a different expression. Consider this code in Y:
	
	`k :: {x, y}`
	`m :: {z}`
	`n :: {v} = and(k {x->v, y->z},`
	`k {x->z, y->v}) {v}`
	`p :: {v} = and(n {v->z}, m {z})`
	
	How can we write a definition for the same `p`, without using `n`?

Answer:
 	
`p(X, Y).`
`q(Z).`
`s(B) = and(and(p(B, C), p(C, B)), q(B)).`
	Answer:
	
	`k :: {x, y}`
	`m :: {z}`
	`p :: {v} = and(and(k {x->v, y->z},`
	`k {x->z, y->v}) {v->z},`
	`m {z})`

Figure 6.11. Question 5

The code above includes two given predicates, `white` and `order`. The predicate `white` determines if the given `Stone` is white. The predicate `order` connects different stones in a linked list. The user-defined predicate `black` determines if a given `Stone` should be black, depending on the stone that comes before it, according to the alternating colors rule.

In the question, the user is presented with this code, and the purpose of the code. It is also explained that there is something wrong with the code, and the participant is asked to fix the incorrect part. The error is an argument mismatch, within the arguments of the call to the `order` predicate. Instead of `order(Stone, PreviousStone)`, the corrected version should write `order(PreviousStone, Stone)`. The definition of the `order` predicate with the correct argument order is the hint.

For the notation Y, we have used a different code with the same level of complexity, and the same sort of existing argument error. The code fragment in the question is a partial program defining paternity relations between individuals. The existing predicate `isMale` determines if the given `person` is male. The second existing predicate `parentOf` determines if a given `parent` is actually the parent of a given `child`. The user-defined predicate `fatherOf` is intended to determine if a given `father` is actually the father of the given `child`. But the code, as suggested, includes an argument mapping mistake. The argument `father` of the `fatherOf` predicate is bound to the value of the `child` argument of the `parentOf` predicate, instead of the `father` argument. The participant is presented with the code, explained that there is something wrong with the code and asked to write the correct version, which should involve modifying the renaming of the arguments in order to correctly bind the *parent* argument of `parentOf` to the value of the `father` argument.

The Notation Y code in the question corresponds to the following CNP code in the strict form:

$$
\begin{aligned}
\mathit{isMale} &: \{\mathit{person}\} \\
\mathit{parentOf} &: \{\mathit{parent}, \mathit{child}\} \\
\mathit{fatherOf} &\leftarrow \mathit{ande}(\mathit{proj}(\mathit{parentOf}, \{\mathit{parent} \mapsto \mathit{parent}, \mathit{child} \mapsto \mathit{father}\}), \\
&\qquad\qquad \mathit{proj}(\mathit{isMale}, \{\mathit{person} \mapsto \mathit{father}\}))
\end{aligned}
$$

which, in order to be made correct, needs to be modified so that the first *proj* operator should instead be:

$$\mathit{proj}(\mathit{parentOf}, \{\mathit{parent} \mapsto \mathit{father}, \mathit{child} \mapsto \mathit{child}\})$$

6.4 Post-test questions

When participants completed the test parts for both notations, they were presented with a few questions regarding their impressions of both no-

Question in notation X	Question in notation Y

A partial program is presented below for deciding what colour of stone to lay in a black and white pavement. The program contains a decision predicate named `black` that determines whether to choose a black stone. The description of this predicate in English language would be "a stone should be black if the previous stone is white".
 	
`white(Stone).`
`order(Previous, Next).`
`black(Stone) = and(order(Stone, PreviousStone),`
`white(PreviousStone)).`
 	
The definition of the `black` contains a mistake. Please try to write the corrected version below.
	The code below is a part of a program that models family relations. It contains a predicate definition `fatherOf` which connects children in the database with their fathers. The definition of this predicate in English language would be "a father of a child is the parent who is also male."
	
	`isMale :: {person}`
	`parentOf :: {parent, child}`
	`fatherOf :: {father, child} =`
	`and(parentOf {parent, child->father},`
	`isMale {person->father} )`
	
	The definition of the `fatherOf` predicate contains a mistake. Please write the correct form below.

The answer should involve replacing the expression
 	
`order(Stone, PreviousStone)`
 	
with
 	
`order(PreviousStone, Stone)`
	The answer should involve replacing the expression
	
	`parentOf {parent, child->father}`
	
	with
	
	`parentOf {parent->father, child}`

Figure 6.12. Question 6

tations. The options available as an answer were: (a) Notation X, (b) Notation Y, and (c) No noticable difference. For group A, the options were listed in this order. For group B, the options for notation Y came first since they had taken the test for notation Y first.

These questions were posed after the study:

1. Which notation did you find easier to read?
2. Which notation did you find easier to modify?
3. Which notation would you choose, if you had to use one for a project?

We also included a field where we asked the participants to leave comments, notes and suggestions, if any.

6.5 Method

The study was executed on-line, by reaching to participants through e-mail. After an initial check, participants were placed in one of the two groups, Group A or Group B, randomly (introduced earlier in the overview of the study, section 6.1).

We reached out to participants who are either working programmers, or researchers using programming languages as a part of their research (as a programmer). We have placed equal number of programmers and researchers in both groups (A and B), and confirmed our initial assessment of their category in the pre-test questions.

There were a total of 20 participants, where 2 were female. The mean age of participants was 30.2 (Median 30, STD=4.18), while the ages ranged from 22 to 39.

The test described earlier was delivered through a form in *Google Forms*, a cloud-based, on-line data collection platform. The forms were accessible on-line through a link provided to each participant via e-mail. There were two forms created for counter-balancing groups A and B, differing only in the order the two notations appeared. Consequently, the data for the two forms were collected in two separate *Google Sheets* spreadsheets, which are also accessible on-line. This is the default form of data collection via *Google Forms*.

Timing points were included in the form, 3 for each notation. After the introduction to a notation is completed, the first time-stamp (t_1) was taken just before the first question started. At the end of the 3^{rd} question, the second time-stamp (t_2) was taken. The third time-stamp was taken at the end of the 6^{th} question. Because the first 3 questions are measuring notation comprehensibility and the last 3 notation modifiability, these

time stamps are sufficient to measure the two durations separately. The results were then downloaded and processed in *Microsoft Excel*.

One of the tests from Group A was regarded as invalid, as it showed that the participant spent a disproportionate amount of time working on the first 3 and the second 3 questions in notation X, respectively 2 minutes and 30 minutes. Upon post-study enquiry, it was revealed that the participant had taken a break, yet had not measured the duration. The result of this test is not included in the final results of the study.

6.6 Results

In this section, we will first look at some information obtained from the study, then we give our interpretation of those results which suggests that the usability of the two notations differ in some aspects but is generally comparable. In the test, two variables were measured: task time, and correctness.

6.6.1 Task time

We have taken two measurements for each notation, comprehension and modification. In the following sections we will report these separately first, then discuss combinations of these such as total task time and weighted total task time.

The total and weighted time is a measure that needs to be interpreted with caution. Simply adding up comprehension and modification durations gives us a total test time for each notation, but comparing these values between notations is not straightforward. There are two issues regarding this comparison. First, the questions we used for the comprehension (Q1, Q2, Q3) and modification (Q4, Q5, Q6) may have contrasting difficulties, and naturally different task times. Therefore it would be deceptive to simply compare total task times of the two notations. Second, we have no concrete information on relative contributions of these sort of tasks in the total usability of programming notations.

Comprehension time

Comprehension times are the duration participants took for the first 3 questions in each test. In the Figure 6.13, the comprehension measurement for each participant can be observed as t_{X_c} for notation X, and t_{Y_c} for notation Y.

Both populations followed normal distribution. Let us look at the mean values of comprehension task times for both notations:

$$M_{X_c} = 276sec \qquad M_{Y_c} = 345sec$$

The mean values show that the comprehension task in notation Y (based on CNP) is 25% slower than notation X (based on PROLOG), with a significance level $P < 0.05$.

Modification time
Modification times are the duration participants took while answering the last 3 questions of the test. Figure 6.13 displays the modifications times for the two notations as t_{X_m} and t_{Y_m}.

Both populations followed normal distribution. Let us look at the mean values of modification task times for both notations:

$$M_{X_m} = 468sec \qquad M_{Y_m} = 366sec$$

The mean durations show that in notation Y (based on CNP), modification tasks that we have tested took 22% less time to complete, with a significance level $P < 0.05$.

Total task time
As discussed earlier, the total task times for completing the whole test is a little more difficult to interpret. The comprehension questions and the modification questions were not necessarily equal in difficulty and task time. For this reason, we will report two kinds of total task time.

Observed total task time: The total task time participants took to complete all the 6 questions in the test. The mean values for both notations for observed total task times are:

$$M_{X_t} = 744sec \qquad M_{Y_t} = 711sec$$

The mean values suggest that notation Y may be slightly faster, but a *Student's t-test* did not confirm that the two populations were different with any significance. A Two-sample *Kolmogorov-Smirnov test* confirmed the null hypothesis that the two sets of total times were from the same distribution ($P = 0.99$). The observed total task times can be found in Figure 6.14 as t_{X_t} and t_{Y_t}.

Equally-weighted total task time: In order to normalize the results, we looked at mean times for comprehension task (M_c), modification task (M_m), and the mean of both values (M_{cm}) to calculate normalizing constants n_c and n_m. Finally, we calculated the means of equally-weighted total task times for each notation (M_{X_w} and M_{Y_w}). The mean task times for each part (regardless of notation) were:

$$M_c = 311sec \qquad M_m = 417sec \qquad M_{cm} = 364sec$$

Calculated normalization constants:

$$n_c = \frac{M_{cm}}{M_c} = 1.171 \qquad n_m = \frac{M_{cm}}{M_m} = 0.872$$

The equally-weighted total task times for both notations were obtained by individually multiplying the comprehension and modification task times, and taking their total. These values can be found in the Figure 6.14 as t_{X_w} and t_{Y_w}.

The mean values for equally-weighted total task times were found as:

$$M_{X_w} = 732sec \qquad M_{Y_w} = 723sec$$

Similar to observed total task times, equally-weighted total task times suggested little difference. The *Student's t-test* did not reject the null hypothesis that the two notations made no difference in equally-weighted total task times. A two-sample *Kolmogorov-Smirnov* test confirmed that the two sets of durations are from the same distribution ($P = 0.99$).

		in seconds			
Group	Participant	t_{X_c}	t_{X_m}	t_{Y_c}	t_{Y_m}
Group A	A1	180	960	240	600
	A2	180	360	180	420
	A3	420	360	240	120
	A4	300	780	240	420
	A5	180	360	240	240
	A6	540	480	660	360
	A7	600	600	600	480
	A8	300	1080	360	360
	A9	120	540	240	360
	A10	360	480	240	360
Group B	B1	240	240	540	240
	B2	120	360	180	300
	B3	360	420	300	240
	B4	300	240	360	240
	B5	300	540	420	600
	B6	180	420	120	240
	B7	120	240	480	600
	B8	120	120	180	180
	B9	420	420	660	540
	B10	180	360	420	420

t_{X_c}: comprehension task time in X
t_{X_m}: modification task time in X
t_{Y_c}: comprehension task time in Y
t_{Y_m}: modification task time in Y

Figure 6.13. Task completion times reported separately for comprehension and modification tasks, showing that the notation X is better in comprehensibility, while notation Y is better for modifiability.

		in seconds			
Group	Participant	t_{X_t}	t_{X_w}	t_{Y_t}	t_{Y_w}
Group A	A1	1140	1048	840	805
	A2	540	525	600	577
	A3	780	806	360	386
	A4	1080	1032	660	648
	A5	540	525	480	491
	A6	1020	1051	1020	1087
	A7	1200	1226	1080	1122
	A8	1380	1294	720	736
	A9	660	612	600	595
	A10	840	840	600	595
Group B	B1	480	491	780	842
	B2	480	455	480	473
	B3	780	788	540	561
	B4	540	561	600	631
	B5	840	822	1020	1015
	B6	600	577	360	350
	B7	360	350	1080	1086
	B8	240	245	360	368
	B9	840	858	1200	1244
	B10	540	525	840	858

t_{X_t}: observed total task time in X
t_{X_w}: equally-weighted total task time in X
t_{Y_t}: observed total task time in Y
t_{Y_w}: equally-weighted total task time in Y

Figure 6.14. Observed versus Equally-weighted total task times for notations X and Y, displaying statistically insignificant difference between the two.

6.6.2 Correctness

As a part of the study, we have looked at how correctly participants solved each question. In order to decide which answers to accept as correct, we followed the same principles, regardless of notation.

In comprehension questions, we accepted only exact answers, as the answers were mostly very short. Some typographic errors in capitalization were tolerated. In longer answers, some differences in articulation were also tolerated. For example, as an answer to the question "*Which argument(s) of q are bound to which argument(s) of k?*", answers such as "*z to x*", "*q.z is bound to k.x*", and "*q.z is bound to k.x renamed to k.z*" were all accepted as long as they were only differing in articulation and did not reflect a misunderstanding of the concepts.

In modification questions, some participants omitted context of the code in the answer, and only wrote the part they had modified. These answers were accepted, as long as the omitted part was not relevant to the required modifications. In some modifications, it was necessary for the participants to come up with new names for variables. We placed no importance on the choice of variable name, as long as the bindings established were correct.

Following the principles above, every answer was marked as either *correct* or *wrong*. Empty or indecipherable answers were regarded as false.

Comprehension questions

In the first 3 questions, we have failed to measure any significant difference in correctness between the two notations. Out of 12 sub-questions, the mean number of correct answers were 11.2 for *Notation X* and 11.3 for *Notation Y*. When calculated over three points for three questions (Every sub-question having 0.25 points), the mean number of correct answers were 2.79 for *Notation X* and 2.81 for *Notation Y*.

Modification questions

In the last 3 questions, we have measured significant difference in correctness between the two notations. Out of 3 questions, the mean number of correct answers were 1.7 for *Notation X* and 2.4 for *Notation Y*. This difference reveals a 42% increase in correct answers in *Notation Y*, with a significance level $P < 0.01$.

When analysed in detail, the highest contrast appeared in Question 5. In *Notation X*, only 3 people answered the question correctly, as opposed to 17 incorrect answers. In contrast, in *Notation Y*, there were 17 correct answers as opposed to 3 incorrect. Every participant who failed to answer Question 5 in *Notation Y* also failed to answer it on *Notation X*. Also, every participant who succeeded to answer it in *Notation X* correctly answered the same question in *Notation Y* as well. A *McNemar's test* also confirmed the significance of this difference with a level $P < 0.01$.

The modification task in Question 5 involved an alpha-renaming step during refactoring. In *Notation X*, this proved to be the problem for most participants who answered the question incorrectly.

Entire test

Looking at the entire test for correctness requires counting questions equally-weighted, as we did in the subsection 6.6.1. We gave each of the first 3 Questions 1 point, regardless of how many sub-questions they contained.

In total, out of 6 questions, mean number of correct questions were 4.4 for *Notation X*, and 5.2 for *Notation Y*. The difference reveals a 16% increase in correctness, with a significance level $P < 0.01$.

We also looked at success rate between the groups, in order to find out if there was an enhanced learning effect caused by the first notation that helped the grasping of the second notation. While participants in the group A scored a mean of 9.9 points out of 12, group B scored $9,2$, but the differences we not statistically significant.

6.6.3 Post-test questions

Participants answered to the post-text questions about their preferences after they have experienced both notations. Here are the number of participants and their choice of answers to each of these questions:

1. Which notation did you find easier to read?
   Notation X: 12
   Notation Y: 8
   No noticable difference: 0

2. Which notation did you find easier to modify?
   Notation X: 8
   Notation Y: 11
   No noticable difference: 1

3. Which notation would you choose, if you had to use one for a project?
   Notation X: 8
   Notation Y: 10
   No preference: 2

6.7 Conclusions

We have found that task times presented interesting results. While the results showed that notation Y (based on CNP) was 25% slower in comprehension tasks compared to notation X (based on PROLOG), it was also found to be 22% faster in modification tasks. There are no discernible difference between the two notations in total task time, looking at both the observed total task time and the equally-weighted total task time. This shows that for the kinds of tasks we measured, CNP is not significantly better or worse than PROLOG in terms of time requirements.

Measuring correctness, we found no evidence of difference between the two notations in comprehension tasks. In modification tasks, however, CNP showed significant promise with 42% improvement in correct answers.

In post-test questions, participants' opinion of the two notations seemed to be more correlated with the time they took in each one, rather than their success in providing correct answers. For example, in comprehension questions we failed to find any difference in correctness, but found that *Notation X* took significantly less time to interpret. Participants found *Notation X* easier to read compared to *Notation Y*, with a rate of 12 to 8. Similarly, we found that in modification tasks, *Notation Y* takes less time than *Notation X*. Participants found *Notation Y* easier to modify with a rate of 8 to 11.

In summation, considering the usability of programming languages depending on so many factors, we can only claim that overall CNP is not worse than PROLOG, and marginally better in some aspects. Considering CNP carries important features of COMBILOG such as compositionality and declarativity, and also considering the obvious difference in practicality of CNP notation compared to the original COMBILOG, we can conclude that CNP could be considered a successful improvement on COMBILOG.

7. Inductive Logic Programming with CNP

For a demonstration of the usability contributions in this thesis, it is natural to look back at the original motivation behind the development of COMBILOG. *Inductive Logic Programming* (ILP) is the primary intended use for COMBILOG, and the reason behind the fundamental properties of the language, particularly the compositionality, the variable-freeness, and the list recursion operators. It is a method for program synthesis from example input/output data, which manifest as ground instances in the context of Logic Programming. Compositionality of COMBILOG allows a simple but efficient technique for ILP, which will be referred to as *decompositional synthesis*.

The meta-interpreter is a meta-logic program representing the provability relation of an object-level logic program (T_O) (in COMBILOG or CNP) and an underlying logic theorem (T) as a predicate (*demo*). For a given T_O, the meta-interpreter *demo*(T_O, T) only succeeds if the underlying theorem logically follows. Since the meta-interpreter is bidirectional, it also functions in the reverse direction to compute an object-level logic program T_O for a given T.

As demonstrated in METAINDUCE [45], the meta-interpreter in the reverse direction can be exploited to synthesize an object-level logic program that logically entails a given set of example program input/output. Through the operators of the language, the given examples are decomposed recursively into operand predicates, until eventually elementary predicates are reached.

The simplicity of the method is thanks to the compositionality and variable-freeness of the language COMBILOG. Since these features are preserved in CNP, the same technique is also applicable here. This will be demonstrated by implementing a decompositional synthesizer that uses CNP as the target language. Since ILP is the motivating use for COMBILOG, it is also a substantiating application for CNP. Moreover, thanks to the better usability of CNP, synthesized programs are easier to comprehend, modify, and combine with hand-written fragments to devise more complicated programs than permitted by the synthesizer alone.

In the following sections we will demonstrate how two CNP programs are devised with this hybrid approach. The process of composing these programs is achieved in part via synthesis and in part via manual programming. This hybrid approach to ILP is not a new idea. User intervention has been considered in many applications when the strength of

the synthesis technique is not sufficient for the synthesis problem to be solved in one step [38, 57]. In the context of decompositional synthesis, it can be considered novel as the improved usability of CNP permits better interaction with the code.

Throughout the development of the sample programs, the usability improvements of CNP will be observed in the following two categories:
Confirmation: The synthesized program fragments need to be confirmed by the programmer as an acceptable implementation of the desired functionality. For this purpose, improved readability of the program representation is fundamental.
Intervention: Some programs are too complex to be synthesized in one step. As the number of arguments in the examples increases, or as the depth of the target predicate expression grows, the synthesis takes proportionally longer time due to combinatorial complexity, to the point of becoming infeasible as a programming methodology. For such cases the programmer needs to take a more active role by splitting the synthesis into fragments and combining the results into a coherent program. These interventions may involve manually composing program fragments. Modifiability of the program representation is crucial for the hybrid program development method described.

In the following sections, after a brief description of the concept of ILP and some relevant approaches to program synthesis, we will present the CNP synthesizer and decompositional synthesis. Finally, two sample programs will be devised with the described hybrid approach, through a user narrative. One program performs list reversal, and the other an insertion sort. The implementation of the CNP synthesizer described in the later section is given in the appendix.

7.1 Inductive Logic Programming

Inductive Logic Programming is a set of machine learning techniques that synthesize logic programs using ground examples of program input/output [73]. Using given positive and negative examples, a synthesizer can derive a logic program that is satisfied for the given positive examples, and is not satisfied for the given negative examples.

Formally, a logic program synthesis procedure infers a program H such that, given a background knowledge B, for positive examples E^+, $B \wedge H \models E^+$, and for negative examples E^-, $B \wedge H \not\models E^-$ [73]. For synthesizing a program H, the complexity of H and the relative necessary minimal sizes of B, E^+ and E^- hint at the strength of the synthesis technique at disposal.

Within the range of methodologies for logic program synthesis, there are a few general approaches. Here we shall briefly visit these approaches

and contrast them with the one employed in this chapter, before going onto the details.

An early group of methods in ILP depend on a search strategy narrowed by the dual of generalizations and specifications. Methods such as anti-unification, least general generalization [93], inverse resolution [72], and predicate invention [71] have been employed in various implementations of ILP [73, 78]. These approaches to synthesis have proven difficult for programs with recursive data types, primarily because of the termination issue [50].

A solution to synthesizing logic programs with recursion is found in MetaInduce [45]. This approach lists predetermined recursion schemes with predicate variables as parameters, and these parameters are replaced with predicate constants during synthesis. This approach is significantly different from the generate-and-test approaches found in most earlier work on ILP. Authors categorize recursive programs according to how many levels of nested recursion operators they contain, where *fold*-2 means 2 nested *fold* operations (including *naiveReverse*), and *fold*-1 means only one (for example, *append*). They show that they can generate *fold*-2 programs with the presented approach. Later they refine this work and present CombInduce, which uses combilog as the target language. combilog was devised to be variable-free and compositional, simplifying the synthesis and permitting meta-level constraints to produce procedurally acceptable programs [48, 49]. Moreover, combilog defines recursion operators which under the compositionality principle allow a precise decomposition of problems through decomposing given examples. CombInduce performs a top-down search guided by constraints, eventually reducing the synthesis problem into trivial instances corresponding to the elementary predicates of the language. This method requires only a few positive ground examples, and can synthesize up to two nested *fold* operators [50]. A comparison of various methods for logic program synthesis including CombInduce can be found in earlier work [11]. This general technique of using a meta-language and constraints to ensure desired properties of the synthesized program is found in multiple instances since the 1990's [22, 102], with more recent works as well producing successful results [74].

As a related methodology, *Deductive synthesis* is a method that primarily uses specifications to synthesize programs. An application of this method that focuses on recursive program schemes also defers to the problem reduction technique [57]. One of the presented examples is the insertion sort algorithm, which we will address in this chapter. The problem with their solution, as readily admitted by the authors, is that the specification required to synthesize insertion sort is as long as the synthesized program itself. The synthesis task for some of the examples takes up to 15 seconds, which may be considered infeasible for interactive

applications. This result can be attributed to the nature of deductive synthesis which requires extensive specifications, but the upside is that these specifications are conceptually closer to software requirements than the actual code. In contrast, our approach relies only on input/output examples, not on specifications. The input/output examples are closer to software test cases, and hints at possible applications under *test-driven development*.

A recent and interesting approach [81] is combining the *constructive type theory* with input/output examples, and a search on refinement trees, which is fundamentally similar to the method we refer to as decompositional synthesis. The method in fact proves quite successful, and authors present the synthesis statistics for a suite of algorithms, almost all finishing under a second. A shortcoming of this method is the number of required examples. For synthesizing a *fold* program the method requires 9 examples. In contrast, our approach requires only 1 or few examples for most cases, including the case we will present here, and still finishes under a second for all synthesis tasks, as will be demonstrated in the following synthesis section. Because the theorem provers based on the *constructive type theory* primarily focus on correctness, procedural aspects of synthesized programs such as complexity and termination have to be taken care of separately. In this particular example, syntactical restrictions on recursive expressions and data types are employed to ensure termination.

Another recent approach is using deep learning to reduce the search space. Balog et al. [7] use deep neural networks to predict possibilities of component functions and predicates being a part of a target program. Mapping each output node to a single component, after training, the probabilities are determined, and used to guide the search. The method is interesting and has a great potential to be very efficient since custom hardware (GPUs) can be used. Despite its advantages, the method is limited since the range of components it can predict is limited by the size of the neural network. Also, it is not a complete method of synthesis as it needs to be combined with a search algorithm.

A survey of the established methods for logic synthesis, namely deductive synthesis, constructive synthesis, and inductive synthesis can be found in [11, 35].

In the following section we present the CNP synthesis approach, including the decomposition constraints. Later in Section 7.3 we narrate two synthesis tasks focusing on a user's actions.

7.2 Synthesis of CNP programs

The CNP synthesizer is a Prolog program. The implementation is in line with the earlier implementation CombInduce [48, 49, 50], which performs a top-down search while decomposing the synthesis problem. Most of the synthesis constraints [48] are applied here as well, primarily the valence constraints for language operators, which we will discuss in the following section. The strengths of CombInduce are preserved in the new synthesizer through the semantic isomorphism between CNP and COMBILOG established in Chapter 5, and due to preserving principle of compositionality.

The decompositional synthesis algorithm performs an iteratively deepening tree search. The search tree corresponds to a predicate expression, while inner nodes correspond to operators and external nodes correspond to predicate identifiers (elementary or user-defined). The depth of the search tree is initially 1, it is iteratively increased up to a fixed depth, until a predicate expression that satisfied all the observables is found.

Every step of the search attempts to find a matching predicate expression for a set of ground examples. When $depth = 1$, it attempts to match with an elementary predicate or a user-defined predicate. When $depth = n$ where $n > 1$, it attempts to match with any operator that can produce a predicate expression with the target constraints. When an operator is found, the predicate arguments of the operator are synthesized through a search with $depth = n - 1$. The search is exhaustive for the given depth.

An example of the decompositional algorithm is visualized in Figure 7.1 for the *sum* predicate. Attempted operator assignments for $depth = 2$ are shown as hypotheses, where separate searches are initiated for their predicate arguments (P and Q). The examples for the predicate arguments are obtained through the semantics of the operator, which is *foldr2* in the successful search path.

The benefit of compositionality of the language semantics manifests itself at this step, as it provides a direct mapping between the extension of the operator and the extension(s) of its predicate arguments. The use of well-modedness constraints further reduces the search space by eliminating procedurally invalid non-terminating programs [48].

As the synthesis of CNP programs is mostly identical to that of COMBILOG programs from earlier work [11, 48, 49], it shall not be described in detail here, but the methodology is explained through an example. The Prolog source code of our CNP synthesizer is provided in Appendix D, as well as the CNP meta-interpreter in Appendix C.

In the following sections, first we describe the decomposition constraints that dictate how the search tree expands, then explain the de-

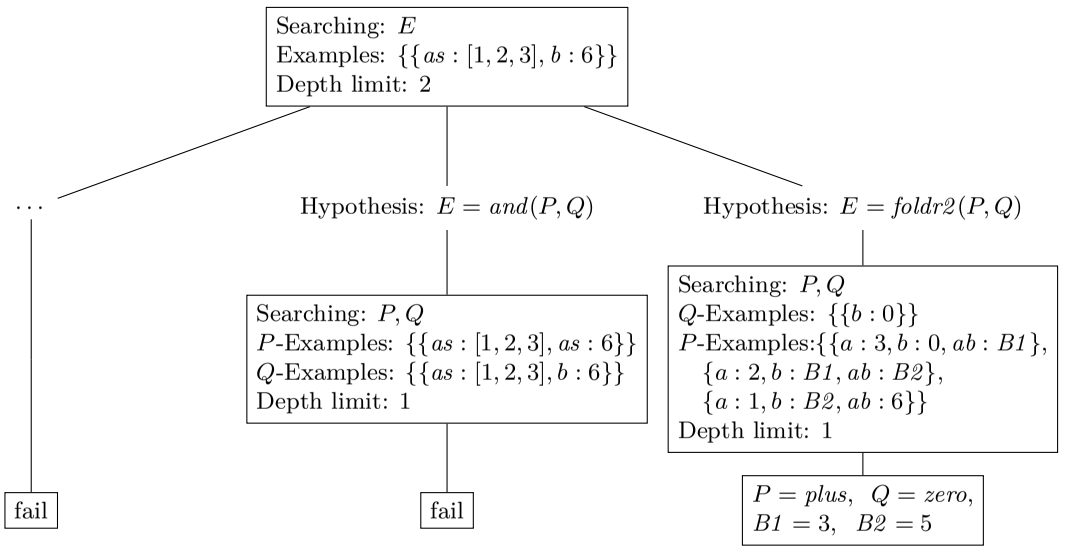


Figure 7.1. A visualization of decompositional synthesis of the *sum* predicate as a search tree. The left and middle branches show failing search paths, while the right branch shows a successful search path resulting with assignments $P = plus$ and $Q = zero$, which when substituted for the initial hypothesis constitute the predicate expression $E = foldr2(plus, zero)$. In the examples, the bound variables *B1* and *B2* are created by the synthesizer for connecting multiple subgoals. Natural numbers are represented as Arabic numerals to save space, as opposed to a Peano-style successor representation, where $0 = [], 1 = [[]], \ldots$. The component predicates *plus* and *zero* are assumed to be pre-defined, where *plus* is satisfied when argument *ab* is the sum of values of arguments *a* and *b*, and *zero* is satisfied when the value of argument *b* is 0 (or []).

compositional synthesis through an example where we discuss the intermediate steps that happen within the synthesizer.

7.2.1 Decomposition constraints

In decompositional synthesis, construction of the search tree is guided by a set of constraints. These constraints are adapted from the original CombInduce [48] to work with CNP which employs argument names. Some alterations were unavoidable due to the the different distribution of functionality among the operators of the languages.

Before discussing specific constraints, we shall introduce the concepts of *mode* and *valence*. A *mode* for a predicate p with n arguments is defined as a function from argument positions $\{1, \ldots, n\}$ to $\{in, out\}$ (or $\{+, -\}$), defining how arguments of a predicate should be used [4]. If a predicate guarantees the output arguments to be ground when the input arguments are ground, it is said to be *well-moded*. CombInduce restricts the synthesized programs to be only well-moded programs, and extends the concept of mode with types for variables, defining a *valence*, and uses valences during search to reduce search space [46, 48, 49]. As a result, CombInduce defines well-modedness constraints for elementary predicates as well as the operators. Here for CNP we use the concept of valence synonymously to a mode, omitting the type constraints. Also, because the arguments do not have positions but only names, a valence in CNP is defined as a function from argument names $\{a_1, \ldots, a_n\}$ to $\{in, out\}$.

The elementary predicates of CNP are associated with acceptable valences. Moreover, for every operator of the language, there are acceptable combinations of valences that map the valence of the operator expression to the valences of its operands. As a result, any predicate expression constructed following these constraints is well-moded.

In the following sections we cover these constraints, and move on to explaining decompositional synthesis.

Elementary predicates

The synthesis of elementary predicates is restricted to only a fixed range of valences consisting of a mode for each named argument. The synthesizer program contains entries for every acceptable valence for each of the elementary predicates. The valences for the elementary predicates of CNP are identical to those in COMBILOG, except for the introduction of argument names.

As an example, one of the acceptable valences for the *cons* predicate is $\{a : in, b : in, ab : out\}$, packing input values of arguments a and b into a single output value bound to argument ab. But a valence such as

Predicate	Valence
isNil	$nil : out$ $nil : in$
id	$a : in, b : in$ $a : out, b : in$ $a : in, b : out$
cons	$a : in, b : in, ab : out$ $a : in, b : out, ab : in$ $a : in, b : in, ab : in$ $a : out, b : in, ab : in$ $a : out, b : out, ab : in$

Figure 7.2. Valences for the elementary predicates.

$\{a : in, b : out, ab : out\}$ is not acceptable, as it would be a call that binds an infinite set of values to arguments b and ab.

The elementary predicate *true* is omitted from the synthesis as it is always *true* for any observable. The elementary predicate *const* for introducing constants is included with only one instance for the constant [] as the predicate $isNil = const(nil, [])$. A table of valences for the elementary predicates can be found in Figure 7.2.

Projection operator

The well-modedness constraints for the projection operator in CNP correspond to a subset of those applying to the *make* operator in COMBILOG and CombInduce [48]. While the *make* operator in COMBILOG can introduce unbound arguments, the *proj* operator in CNP cannot. This results in less complicated constraints for the *proj* operator compared to the *make* operator.

The single well-modedness constraint for the *proj* operator concerns the valence of the *proj* operation and the valence of its source expression. Every input argument of the source expression must be projected by the *proj* operator, as leaving them unbound would break well-modedness.

Logic operators

There are separate constraints applying to the two auto-expanding logic operators *ore* and *ande*.

For the synthesis of the disjunction operator *ore*, the valences of the operand predicates are required to be identical to the valence of the resulting predicate expression.

The conjunction operator *ande* requires a set of constraints including those of both the *and* and *make* operators in COMBILOG . This is due to the *ande* operator taking over some of the behaviour of the *make* operator

***ande(P,Q)* valence**	***P* valence**	***Q* valence**
$a : in$	$a : in$	$a : in$
$a : out$	$a : out$	$a : out$
$a : out$	$a : out$	$a : in$
$a : in, b : out$	$a : in$	$b : out$
$a : in, b : out$	$a : in, b : out$	$b : in$
$a : in, b : out$	$a : in, b : out$	$b : out$
$a : in, b : out$	$a : in, b : out$	$a : in$
$a : in, b : out$	$a : in, b : out$	$a : in, b : in$
$a : in, b : out$	$b : out$	$a : in, b : in$
$a : in, b : out$	$b : out$	$a : in, b : out$
...	...	...

Figure 7.3. Some example valence combinations for the *ande* operator.

in COMBILOG, and it can leave some arguments of its operators unbound between operands.

For a binary logic operation $ande(P_{\theta_1}, Q_{\theta_2})_\alpha$. The valence w of the *ande* operation, and the valences u and v respectively for components P and Q are defined as follows, by a set of rules.

$$w : Dom(\alpha) \to \{in, out\}$$
$$u : Dom(\theta_1) \to \{in, out\}$$
$$v : Dom(\theta_2) \to \{in, out\}$$

The composition $ande(P_{\theta_1}, Q_{\theta_2})_\alpha$ is well-moded, if for any argument name $a \in Dom(\alpha)$:

- if $v(a) = in$, then either $u(a) = out$ or $w(a) = in$.
- if $u(a) = in$, then $w(a) = in$.
- if $w(a) = out$, then either $u(a) = out$ or $v(a) = out$.

Some examples of these rules are displayed in the table in Figure 7.3. For a discussion of these constraints the reader is referred to [4, 49].

Recursion operators

The well-modedness constraints for the recursion operators in the CNP synthesizer are identical to those found in COMBINDUCE, except the introduction of names. A table of acceptable valence combinations for the *foldr* operator is given in Figure 7.4. The valence constraints related to variant *foldr2* is also identical to the COMBINDUCE one with the exception of argument names, therefore it is omitted here.

In the next section we present the decompositional synthesis through the discussion of an example synthesis task.

***foldr(P,Q)* valence**	***P* valence**	***Q* valence**
$a0 : in, as : in, b : out$	$a : in, b : in, ab : out$	$a : in, b : out$
$a0 : in, as : in, b : out$	$a : in, b : in, ab : out$	$a : out, b : out$
$a0 : out, as : in, b : out$	$a : in, b : in, ab : out$	$a : in, b : out$
$a0 : out, as : in, b : out$	$a : in, b : in, ab : out$	$a : out, b : out$
$a0 : in, as : in, b : out$	$a : out, b : in, ab : out$	$a : out, b : out$
$a0 : out, as : in, b : out$	$a : out, b : in, ab : out$	$a : out, b : out$

Figure 7.4. The table of acceptable valence combinations for the *foldr* recursion operator and its operands for the recursive case (*P valence*) and the base case (*Q valence*).

7.2.2 Decompositional synthesis

The CNP synthesizer is essentially a reverse implementation of the CNP meta-interpreter with operational improvements such as breadth-first search, depth limit, and pruning of not well-formed sub trees using the decompositional constraints presented earlier. When presented with an operator expression such as $op(P, Q)$, where operands P and Q are predicate expressions, the meta-interpreter evaluates P and Q first, then performs the operation over their values, and returns the resulting value as the result of the operator expression.

The synthesizer instead starts with a valence and a set of ground examples, and constructs a predicate expression E_{cnp} that is true for the given ground examples. The predicate expression E_{cnp} can be an elementary predicate, as long as it is true for the given ground examples, and it has a matching valence. It can also be an operation such as $ande(P, Q)$, where P and Q are component predicate expressions. In this case, synthesis sub-tasks are constructed for P and Q, where new valences are inferred through valence mappings established between the operator and its operands, and new ground examples are also constructed according to the original ground examples given for the *ande* operation. These synthesis sub-tasks may succeed, in which case a successful predicate expression is found; or they may not succeed, in which case the synthesizer moves to the next operation, until it exhausts every operation possible within the constraints. In this way, the synthesizer is analogous to a reverse meta-interpreter in the sense that instead of evaluating operands and combining their results, it first partitions the examples into operands and synthesizes each operand using the corresponding partition.

As an example of decomposition, let us consider a synthesis corresponding to the *append* predicate. The user states that this predicate should have three arguments: the first list (xs), the second list (ys), and the list that consists of the first and second lists appended together (zs). It is also required that when the xs and ys arguments are ground, the predicate should be able to bind the argument zs to a ground value. This synthesis

task is expressed by the following statement:

Synthesize a predicate R such that;
it has the valence: $\{xs : in, ys : in, zs : out\}$,
and it is true for the example(s): $\{\{xs : [1,2], ys : [3,4], zs : [1,2,3,4]\}\}$

which translates to the following goal in the incremental synthesizer:

$$\leftarrow synInc(R, \{xs : in, ys : in, zs : out\}, \{\{xs : [1,2], ys : [3,4], zs : [1,2,3,4]\}\}).$$

The incremental synthesizer initiates the synthesizer predicate *syn* with an increasing depth parameter. The synthesizer initiated at $depth = 1$ attempts to match the given constraints with the elementary predicates, which all fail for the given example, even though there are elementary predicates with the given valence. Then, at $depth = 2$ it attempts to find a combination of operators and elementary predicates, which also fails. At $depth = 3$, the synthesizer attempts the following assignment for R, which is:

$$R = proj(S, \{X \rightarrow xs, Y \rightarrow ys, Z \rightarrow zs\})$$

In the above expression, variables S, X, Y, and Z are not bound. They are due to be bound by the following sub-task for synthesizing S. When S is bound to a predicate expression, the source argument names X, Y, and Z will also be bound to the argument names of that predicate expression. For this purpose, the synthesizer devises a sub-task to synthesize S, with the same examples, same valence, but undetermined argument names (X, Y, and Z) and a decreased depth parameter, since one level of depth is allocated for the *proj* operator. This sub-task devised and executed by the synthesizer is as follows:

$$\leftarrow syn(S, 2, \{X : in, Y : in, Z : out\}, \{\{X : [1,2], Y : [3,4], Z : [1,2,3,4]\}\}).$$

This synthesis call tries different operators to match S, which fails until eventually it attempts to match it with the *foldr* operator, whose acceptable valences match with the ones the synthesizer is searching for. This results in the following substitution for S:

$$S = foldr(P, Q)$$

The argument names for the *foldr* operator are fixed by definition, which are *as*, *a0*, and *b*. The synthesizer attempts various combinations

of these argument names with X, Y, and Z, and eventually attempts assignment $\sigma = \{X = as, Y = a0, Z = b\}$. At this point, when the undetermined argument names are substituted for, and the assignment to S is placed in its context, the overall in-progress predicate expression for the initial goal is as follows:

$R = proj(foldr(P, Q), as \rightarrow xs, a0 \rightarrow ys, b \rightarrow zs)$.

At this stage, the operands of the *foldr* operator P and Q are unbound, and the synthesizer goes on to devise sub-tasks to synthesize these components. This involves inventing new examples for the operands, that are obtained through the semantics of the *foldr* operator, evaluated in inverse by the synthesizer. The single example for the synthesis of S was $\{X : [1, 2], Y : [3, 4], Z : [1, 2, 3, 4]\}$, which after the substitutions for undetermined argument names becomes:

$\{as : [1, 2], a0 : [3, 4], b : [1, 2, 3, 4]\}$

The synthesizer decomposes this example through the semantics of the *foldr* operator, and obtains the following one example for Q and two examples in conjunction for P. The single example for the base case Q is:

$\{a : [3, 4], b : B0\}$

while the two examples for P are:

$\{a : 2, b : B0, ab : B1\}$,
$\{a : 1, b : B1, ab : [1, 2, 3, 4]\}$

The synthesizer then devises sub-tasks corresponding to Q and P using these examples, and executes them in conjunction:

$$\begin{aligned}\leftarrow\ & syn(Q, 1, \{a : in, b : out\}, \{\{a : [3, 4], B0\}\}) \ \wedge \\ & syn(P, 1, \{a : in, b : in, ab : out\}, \{\{a : 2, b : B0, ab : B1\}\}) \ \wedge \\ & syn(P, 1, \{a : in, b : in, ab : out\}, \{\{a : 1, b : B1, ab : [1, 2, 3, 4]\}\}).\end{aligned}$$

Due to the depth constraint being equal to 1, the synthesizer only attempts to match Q and P with elementary predicates or user-defined predicates and not with an operator. Using any composition operator would result in an expression depth larger than 1. The goal succeeds with substitutions $\sigma = \{Q = id, P = cons, B0 = [3, 4], B1 = [2, 3, 4]\}$, with matching valences and satisfied examples for Q and P. As a result, the substitutions for Q and P are applied, and the earlier hypothesis for

S becomes:

$S = foldr(cons, id)$

When S is placed in its context, the final predicate expression for the initial goal becomes:

$R = proj(foldr(cons, id), \{as \rightarrow xs, a0 \rightarrow ys, b \rightarrow zs\})$

which is the correct CNP implementation for the *append* predicate. Even though the synthesizer only guarantees that it terminates for one mode ($\{a : in, b : in, ab : out\}$), the resulting predicate is still a multi-directional predicate which can be called, for example, to find out every possible lists xs and ys for a given appended list zs.

In the following section the synthesis of some sample programs will be analysed from a usability perspective, omitting most of the details presented in this section.

7.3 Sample synthesis scenarios

In this section we present two sample programs developed by a hybrid approach with some of the code being synthesized and some being manually written.

The first example is the list reversal operation, which is a fold-2 expression, containing 2 levels of nested fold operators. The intention with this example is to show that the strength of the CNP synthesizer is at the same level as CombInduce[49]. The second example is the insertion sort algorithm. This example involves more steps than the first and demonstrates the feasibility of the hybrid synthesis in a more realistic example. In both examples, the development is narrated focusing on the user's plans and actions.

7.3.1 Synthesis of list reversal

In this section we will look at the task of synthesizing a predicate that reverses a list. The initial attempt by the user is through the use of a single example, which is reversing the list $[1, 2, 3]$ and obtaining the list $[3, 2, 1]$. This is represented by the example $\{as : [1, 2, 3], bs : [3, 2, 1]\}$, and the valence $\{as : in, bs : out\}$, that is, when the argument as is bound to a value, the predicate call should guarantee that the argument bs will be bound to a value.

In the following presentation of our examples, we consider the synthesizer as a front-end tool executed in the Prolog interpreter. We only

show the goals typed in by the user, the synthesis results calculated by the synthesizer, with no discussion in between as we did in the earlier section. As a result it is more intuitive to typeset the examples verbatim, since they are code.

In order to initiate the synthesis; the CNP synthesizer, the CNP meta-interpreter, and the CNP library modules are consulted into the PROLOG environment, and the synthesizer is called by the following goal:

```
?- synInc(P, [as:in, bs:out], [[as:[1,2,3], bs:[3,2,1]]]).
```

The `synInc` predicate is the entry point for the synthesizer, which performs depth-limited calls to the actual synthesizer predicate `syn`. This goal fails, as the synthesizer tries expression trees up to a maximum depth of 6. It cannot find a predicate expression that satisfies the given example, and exits. At this point, the user needs to intervene as the synthesizer is not strong enough to synthesize this predicate in one step.

A naive implementation of the list reversal involves iterating through a given list, and appending each element to the end of a list starting with the empty list. Even though the append predicate expression is rather compact in CNP, it can only append lists to lists, but not elements to lists. The user hypothesizes that the synthesis may succeed if it is provided with a library predicate that can pack an element in a singleton list. As a result the user tries to synthesize this predicate as a first step.

This predicate is exemplified by the named ground instance $\{a : 1, aList : [1]\}$, and the synthesis is performed via the goal:

```
?- synInc(P, [a:in, aList:out], [[a:1, aList:[1]]]).
```

which results in the synthesis of the following predicate expression as it is bound to the variable P:

```
P = proj(ande(cons, proj(isNil, [nil->b])), [a->a, ab->aList])
```

The user confirms that this is the desired definition, and manually adds it to the CNP library with the name *asList*. This is undertaken via the *defPredicate* predicate, which assigns predicate names to predicate bodies, and is utilized by the meta-interpreter as well as the synthesizer. In order to make this predicate available to the synthesizer, for every valence of the predicate that is known to terminate, a separate instance of the *valencePredicate* predicate is stated:

```
defPredicate(asList,
  proj(ande(cons, proj(isNil, [nil->b])), [a->a, ab->aList])).
```

```
valencePredicate(asList, [a:in, aList:out]).
```

With this entry made to the CNP library, the user tries the earlier goal again:

```
?- synInc(P, [as:in, bs:out], [[as:[1,2,3], bs:[3,2,1]]]).
```

This succeeds and returns with the predicate expression:

```
P = proj(foldr2(
      proj(foldr(cons, proj(asList, [a->a, aList->b])),
        [a0->a, as->b, b->ab]),
      proj(isNil, [nil->b])), [as->as, b->bs])
```

The predicate expression consists of a projection of a *foldr2* operation, with a projection of the *isNil* predicate for the base case, and a *fold* operation for the recursive case. This *foldr* expression is similar to the implementation of *append*, which is $foldr(cons, id)$, but instead of *id*, uses a projection of the *asList* predicate as the base case, which lets it append a single element to a list, interpreting the single element as a singleton list, through the use of the *asList* predicate. The outer operation *foldr2* iterates through the given list, and at every step it appends the new element to the end of a list, effectively obtaining a reversed list. Due to the improved usability of CNP, the user can comprehend this code fragment relatively easily, and confirm it as the acceptable implementation for the intended purpose, adding it to the library:

```
defPredicate(reverse,
  proj(foldr2(
    proj(foldr(cons, proj(asList, [a->a, aList->b])),
      [a0->a, as->b, b->ab]),
    proj(isNil, [nil->b])), [as->as, b->bs])).
```

After being added to the library, the *reverse* predicate is now available through the CNP meta-interpreter. A goal that uses the predicate, as in,

```
?- cnp(reverse, [as:[a,b,c,d], bs:Bs]).
```

succeeds with the following answer:

```
Bs = [d,c,b,a].
```

The predicate can also be used in the reverse direction, by giving a reversed list:

```
?- cnp(reverse, [as:As, bs:[d,c,b,a]]).
```

which succeeds with the following answer:

```
As = [a,b,c,d].
```

As a result, the user has successfully synthesized a fold-2 predicate (a predicate that has two nested fold operators [45]) without writing any CNP code, but by intervening with the synthesis process through dividing the problem and confirming the results. The improved usability of CNP is essential for these interventions.

7.3.2 Synthesis of insertion sort

The next example is the hybrid synthesis of an insertion sort algorithm in CNP. In its entirety, the algorithm is too complex to be synthesized in a single pass of the synthesizer. This requires a meta-algorithm to be devised by the user, and its fragments to be synthesized or hand-written, depending on how complex the fragment needs to be. The resulting program is a hybrid between a hand-written program and a synthesized one.

The meta-algorithm of insertion sort devised by the user is assumed to be as follows:

1. Write a predicate `insertOrd` that can insert a given integer into an ordered list in the correct position, in such a way that after insertion the list preserves the same order.
   a) First, split the list into two lists, one that holds the integers that are less in value than the one being inserted, and one with integers with values greater than or equal to it.
   b) Second, concatenate the two lists and the new integer in such a way that the new integer goes in between the lists, thereby creating a new list that preserves the order.
2. Using the `insertOrd` predicate, write a predicate `iSort` that takes a list of integers, and inserts every integer in that list into a new list using the `insertOrd` predicate, starting from an empty list.

First, let us look at the hybrid synthesis steps for writing the `insertOrd` predicate.

In order to extract the elements that satisfy a specific predicate from a list, CNP offers an operator `filter(P)`, which is not included in synthesis but provided as an operator in the meta-interpreter. This operator takes a source predicate expression `P`, gives a predicate expression that has two arguments, a list (`as`) that contains all the elements, and a second list (`bs`)

that contains only the elements that satisfy the given predicate `P`. `P` has three arguments: `a`, `param`, and `b`. When a given value for `a` is accepted, the argument `b` is assigned to that value, otherwise it is assigned to *nil*. The `param` argument can be used to pass a supporting value.

For synthesizing the `filter` operation, the user first needs to synthesize its predicate arguments that can filter the integers less than or greater than or equal to a given integer.

The synthesized predicate should assign the argument `b` to the value of `a` if the value of `a` is less than the value of the `threshold` argument. This is exemplified by three cases, one where `a` is less than `threshold`, one where `a` is equal to `threshold`, and one where `a` is greater than `threshold`. For the latter cases, the argument `b` should be *nil*. These ground examples lead to the following synthesis goal:

```
?- synInc(P, [a:in, threshold:in, b:out],
          [[a:4,threshold:5,b:4],
           [a:4,threshold:4,b:[]],
           [a:4,threshold:3,b:[]]]).
```

which results in the suggested expression:

```
P = ore(ande(proj(gte, [a->a, b->threshold]),
             proj(isNil, [nil->b])),
        ande(id,
             proj(gt, [a->threshold, b->b])))
```

In the first case of the `ore` operator, the synthesizer came up with an `ande` expression which composes a projection of the elementary predicate `gte` that is satisfied when argument `a` is greater than or equal to the argument `threshold`, and a projection of the `isNil` predicate which is satisfied when the argument `b` is the term ([]), effectively assigning the value of `b` to `nil` if `a` is greater than or equal to `threshold`.

The second case of the `ore` operator, uses a projection of the elementary predicate `gt` with its arguments inverted as a *less than* predicate that is satisfied when argument `a` is less than argument `b`, in conjunction with the `id` predicate (satisfied when `a` and `b` are identical), effectively assigning the argument `b` to the value of `a` if the value of `a` is less than the value of `threshold`.

As this is the correct implementation for the desired behaviour, the user confirms this implementation and adds it to the library with the name `ltPass`:

```
defPredicate(ltPass, ore(ande(proj(gte, [a->a, b->threshold]),
                              proj(isNil, [nil->b])),
```

```
                        ande(id,
                             proj(gt, [a->threshold, b->b]))))).
```

which can be tested via the following goals:

```
?- cnp(ltPass, [a:5, threshold:6, b:B]).
B = 5

?- cnp(ltPass, [a:6, threshold:6, b:B]).
B = []

?- cnp(ltPass, [a:7, threshold:6, b:B]).
B = []
```

The new predicate `ltPass` can be used to hand-write a predicate expression that filters only the elements lower than a given threshold:

```
defPredicate(filterLt,
             filter(proj(ltPass, [a, threshold->param, b]))).
```

which can also be tested as follows:

```
?- cnp(filterLt, [as:[1,2,3,4,5], param:3, bs:Bs]).

Bs = [1, 2]
```

The user repeats the same procedure for synthesizing a `gtePass` predicate. It is subsequently used to write a `filterGte` predicate that extracts the elements greater than or equal to a given `param` value from a list. These actions result in the following predicate definitions:

```
defPredicate(gtePass,
             ore(ande(proj(gt, [b->a, a->threshold]),
                      proj(isNil, [nil->b])),
                 ande(id,
                      proj(gte, [b->threshold, a->b])))).

defPredicate(filterGte,
             filter(proj(gtePass, [a, threshold->param, b]))).
```

In order to append the lists extracted by the `filterLt` and `filterGte` predicates, the user also provides a synthesized `append` predicate, whose synthesis was described earlier:

```
defPredicate(append,
             proj(foldr(cons, id), [as->xs, a0->ys, b->zs])).
```

The predicates defined so far are sufficient for composing the planned `insertOrd` predicate, which inserts a given element into an ordered list in the correct order, returning a new ordered list that contains the new element. Due to the synthesizer not being strong enough for the task, because of the relatively high number of arguments involved, the user writes the following definition for `insertOrd` by hand:

```
defPredicate(insertOrd,
 proj(ande(proj(filterLt, [as, param->elem, bs->listLt]),
           proj(filterGte, [as, param->elem, bs->listGte]),
           proj(cons, [a->elem, b->listGte, ab->newListGte]),
           proj(append, [xs->listLt, ys->newListGte, zs->bs])),
      [as, elem, bs])).
```

The definition above uses the `filterLt` and `filterGte` predicates to split a given list into two (`listLt` and `listGte`) by a given threshold in the `elem` argument. It prepends the `elem` itself to the `listGte` and obtains `newListGte` using the elementary predicate `cons`. Finally it appends the `listLt` and `newListGte` lists to obtain the list in the `bs` argument, containing the list after the insertion of `elem`. This hand-written predicate works as follows, when called through the CNP meta-interpreter:

```
?- cnp(insertOrd, [as:[2,4,6], elem:5, bs:Bs]).

Bs = [2, 4, 5, 6] .
```

For this hand-written predicate the user also adds the `valencePredicate` instance that makes it available to the synthesizer:

```
valencePredicate(insertOrd, [as:in, elem:in, bs:out]).
```

At this point, the user has completed the first step of writing the insertion sort algorithm. The second step is solely a synthesis task. The goal below attempts to synthesize a predicate with two arguments `list` and `sorted`, with a single named ground example $\{list : [5,1,2], sorted : [1,2,5]\}$, given as follows:

```
?- synInc(P, [list:in, sorted:out],
          [[list:[5,1,2], sorted:[1,2,5]]]).
```

Predicate	Method
`ltPass`	Synthesized
`gtePass`	Synthesized
`filterLt`	Hand-written
`filterGte`	Hand-written
`append`	Synthesized
`insertOrd`	Hand-written
`iSort`	Synthesized

Figure 7.5. The list of predicates involved in the insertion sort algorithm

which results in the following synthesized predicate expression:

```
P = proj(foldr2(proj(insertOrd, [elem->a, as->b, bs->ab]),
                proj(isNil, [nil->b])),
         [as->list, b->sorted])
```

The synthesized predicate expression conforms with the recursive nature of the second step in the user's plan, using the specific fold variant `foldr2` operator. It iterates through the list given in the `list` argument, passing every element to the `insertOrd` predicate, inserting them onto a list which stays ordered due to the behavior of the `insertOrd` predicate. Consequently, the user confirms this expression as the correct implementation and adds it to the library, giving it the name `iSort`:

```
defPredicate(iSort,
  proj(foldr2(proj(insertOrd, [elem->a, as->b, bs->ab]),
              proj(isNil, [nil->b])),
       [as->list, b->sorted])).
```

The new predicate can be tested through the CNP meta-interpreter:

```
?- cnp(iSort, [list:[50,10,10,20,30], sorted:S]).

S = [10, 10, 20, 30, 50]
```

With the synthesis of the final `iSort` predicate, the insertion sort meta-algorithm initially planned by the user is implemented through a hybrid method involving both synthesized and hand-written fragments. The list of predicates in the Figure 7.5 reveals the pattern this mixed method entails. The equivalent PROLOG code can be found in Appendix E.

7.4 Conclusion

In this chapter, meta-interpretative inductive synthesis is presented as an example application of COMBILOG. This technique was the original motivation behind COMBILOG, since establishing a variable-free language that is semantically equivalent to definite clause programs significantly amplifies the strength and simplicity of the technique. As CNP is a usability improvement over COMBILOG, performing the same kind of synthesis directly in CNP shows that CNP is equally capable for this type of work.

Moreover, the improved usability of CNP allows a method presented here as *hybrid synthesis*, where some parts of the program are synthesized and some parts are hand-written by the user. To wit, let us observe the following code fragments, which are the synthesized predicate expressions for the `ltPass` predicate in the insertion sort example presented earlier, given in COMBILOG and CNP to compare. In COMBILOG:

```
P = or(and(make([1,2,3],gte),
           make([2,3,1],isNil)),
       and(make([1,3,2],id),
           make([2,1,3],gt))).
```

and the same code fragment synthesized in CNP:

```
P = ore(ande(proj(gte, [a->a, b->threshold]),
             proj(isNil, [nil->b])),
        ande(id,
             proj(gt, [a->threshold, b->b])))
```

The CNP code is easier to verify as the correct implementation. Argument names help to establish a mental connection between arguments, reducing the need to memorize predicate signatures and the number of arguments being dealt with. In order to read the COMBILOG code, the user needs to memorize which arguments are mapped to which index, and memorize the arities of each predicate. The increased usability of CNP enables users to confirm the synthesized predicate with greater ease. Thus, it contributes fundamentally to *hybrid synthesis*.

The increased usability makes it easier to write fragments by hand as well. If we consider the definition of the `insertOrd` predicate, the difference in difficulty between writing the COMBILOG fragment and the CNP fragment can be observed. If hand-written in COMBILOG, the `insertOrd` predicate definition is as follows:

```
make([1,2,3], and(make([2,1,4,3,5,6], filterLt),
                  make([2,1,4,5,3,6], filterGte),
```

```
                make([1,4,5,6,2,3], cons),
                make([4,5,3,1,6,2], append))).
```

While composing this expression by hand, the user needs to plan the indices ahead, and go back-and-forth between the indices and the predicate names. This applies also for expanding the components of the `and` operator, since the number of arguments needs forward planning. The outer `make` operator used for cropping needs to be written last.

In contrast, the equivalent hand-written CNP code from the insertion example analyzed earlier, is as follows:

```
proj(ande(proj(filterLt, [as, param->elem, bs->listLt]),
          proj(filterGte, [as, param->elem, bs->listGte]),
          proj(cons, [a->elem, b->listGte, ab->newListGte]),
          proj(append, [xs->listLt, ys->newListGte, zs->bs])),
     [as, elem, bs])
```

As well as eliminating the argument indices, the CNP code reduces the number of times the user needs to travel back-and-forth within the expression. The argument names are not as fluid as indices, which allows the choice of an arbitrary name for a bound argument through the `proj` operator. The auto-expanding `ande` operator reduces repetitive code as the user need not deal with the unbound arguments. As a result, the CNP predicate expression is much easier to write and modify.

These usability concerns are revisited here as a brief reminder of the usability analysis of COMBILOG found in Chapter 3 and the usability study of CNP found in Chapter 6. The improvements contribute to program synthesis by making it available for practical use. Although the framework for decompositional synthesis invented with COMBILOG is theoretically useful as a programming tool, uptake has been limited due to low usability. The CNP language addresses many of these problems and increases the potential for adoption. CNP facilitates a user-oriented decompositional synthesis approach referred to here as *hybrid synthesis*, where improved readability leads to simpler validation of synthesized code and improved modifiability leads to the straightforward composition of hand-written code. As noted earlier, hybrid synthesis, or interactive synthesis is not a new idea, but the novel concept here is facilitating hybrid *decompositional* synthesis.

8. Conclusion

In the earlier chapters, we identified the problem of usability in COMBILOG, and addressed it in two distinct ways. Issues were identified in Chapter 3 through a usability evaluation of the COMBILOG notation, which led the way to the solutions presented throughout the thesis. One of these was based on visualizing COMBILOG code through VISUAL COMBILOG, and one was the creation of a more usable textual iteration of COMBILOG, that is, CNP.

Both of these solutions are intended to equip COMBILOG for the original purpose behind its invention. Namely, establishing a compositional logic programming language that is equivalent to definite clauses, which can be used as a meta-language for inductive program synthesis. Here let us revisit and emphasize how each of these can be utilized in relation to program synthesis.

In Chapter 4 a system was designed that can be used to visualize any COMBILOG program as is, and can be used to modify a COMBILOG program by modifying the visualization. This approach assumes the user would like to keep COMBILOG as the target language of synthesis, yet would like to deal with reading and editing synthesized programs more easily. VISUAL COMBILOG supports this objective with a user-friendly visual notation, and an accompanying parser/generator that establishes the link between COMBILOG and VISUAL COMBILOG. As a proof of concept, a prototype was implemented with mirrored-editing feature which allows a programmer to edit COMBILOG code on one side, through modifying a *Visual Combilog* diagram on the other. The editor continuously updates the mirrored representations on-the-fly, and the user is able to use whichever notation they prefer for the task at hand.

The results of the user study in Chapter 4, show that when COMBILOG code is accompanied with the visual notation, it is 46% faster to interpret and users make 69% fewer mistakes while doing so. As a demonstration of this effect, Figures 8.1 and 8.2 display the COMBILOG code and the corresponding VISUAL COMBILOG diagram for the `append` predicate. Let us assume the code in Figure 8.1 is synthesized by the COMBILOG synthesizer. Using our VISUAL COMBILOG editor, this code can be visualized, and interpreted through visualization to confirm it as the correct implementation or not. Moreover, by interacting only with the diagram, the user can modify the COMBILOG code (as the changes on the diagram are continuously reflected on the textual code) and commit to the

```
append <- or(and(make([1,2,3], isNil),
                 make([3,1,2], id)),
             make([1,2,3], and(make([3,4,5,1,2,6], cons),
                               make([4,2,5,6,1,3], append),
                               make([4,5,3,1,6,2], cons))))
```

Figure 8.1. COMBILOG code for the `append` predicate

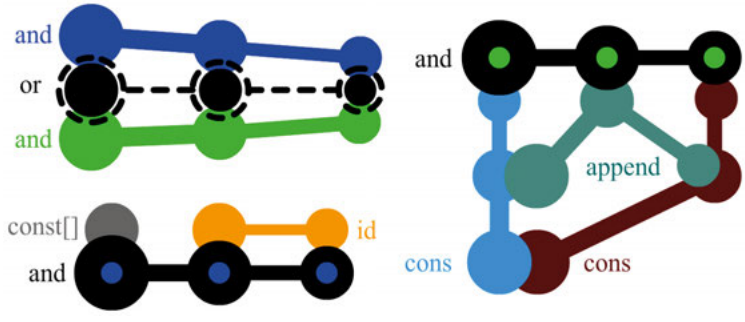


Figure 8.2. VISUAL COMBILOG diagram for the *append* predicate in three partitions. Top-left graphs shows the top level composition of the predicate as a disjunction, with one base case and one recursive case, which are separately given at the bottom-left and on the right, respectively.

new version when the correct implementation is arrived at. The usability improvements brought by VISUAL COMBILOG have a direct effect on usability of COMBILOG as a target language for program synthesis.

Although VISUAL COMBILOG significantly improves usability, there are still some obstacles to its adoption. Diagrams are not regarded first-class currency in current systems, which restricts their use to specific applications, limiting the potential for tool support. Moreover, the design of VISUAL COMBILOG was subject to some constraints inherent to COMBILOG, such as the fixed argument order. The visual notation could not depart too far from the concepts of COMBILOG, so as not to break the on-the-fly compatibility between the two. For example, if a minor change in the diagram resulted in a significant one in its COMBILOG dual, it would alienate the user by invalidating the mental map they constructed. Such limitations prompted exploration of alternative approaches: one of which was a more usable iteration of the textual COMBILOG notation.

Chapter 5 presents CNP. This new textual representation was designed to address the issues identified in Chapter 3. Chapter 5 then gives the formal semantics of CNP, and a proof of the isomorphism between the semantics of COMBILOG and CNP. Most importantly, the two essential features: the variable-freeness and compositionality of COMBILOG were

preserved in CNP. This isomorphism suggests that CNP should be able to replace COMBILOG in virtually all its uses, including program synthesis. To investigate any usability improvements in CNP, in Chapter 6 a user study compared the usability of CNP to a common notation for logic programming that uses variables (PROLOG). The results showed that while CNP was on average 25% slower than the PROLOG in comprehension tasks, it was also 22% faster in modification tasks. When both tasks were measured together, there was no statistically significant difference among the two. The results also showed that CNP has a significant advantage in the outcome of modification tasks: a 42% improvement in correct answers and 52% fewer mistakes were recorded. This reveals that CNP can be just as usable as PROLOG for the tasks in the study, which were designed around interpreting and altering relational argument binding.

After confirming the usability improvements in CNP, we exemplified its use as a target language for inductive program synthesis using the same technique as COMBILOG. This application, presented in Chapter 7, reaffirms that CNP can replace COMBILOG for its original motivating application. The strength of the synthesizer was the same as for COMBILOG, namely that it could synthesize programs up to two levels of nested *fold* operators with as few as only one input/output example. The improved usability of CNP is demonstrated through some synthesis examples that require a hybrid approach, where some fragments of a program are synthesized and some are hand-written. This hybrid approach is bolstered by the introduction of CNP, as comprehending and modifying COMBILOG is more difficult. A hybrid synthesis approach in COMBILOG would potentially involve translation to another language that is more usable, which may lead to reverse translation issues in order to be able to feed the edited fragments back into the synthesizer as library predicates. On the other hand, CNP fragments can be synthesized, read, edited, and fed back into the synthesizer as library predicates as they are. Chapter 7 presented a demonstration of synthesizing an insertion sort algorithm by this hybrid approach. A fragment of this algorithm can be seen in Figure 8.4, where the `gtePass` predicate is synthesized in both COMBILOG and CNP. The function of this predicate is passing through only the values greater than or equal to a given threshold value.

To summarize, we analysed the usability issues of COMBILOG, and presented two distinct solutions. We measured the improvements brought by each of these using empirical methods. Both approaches were implemented as usable sofware tools. Finally we demonstrated how the second approach (CNP) can, in fact, replace COMBILOG in its original use case. Of course, the usability of a programming language cannot be entirely determined by a limited number of specific tasks. The community, available libraries, usability of the compiler/interpreter, and distinctive uses for the language are all contributing factors. We should admit that our studies

```
gtePass = or(and(make([2,1,3],gt),
                 make([2,3,1],isNil))),
             and(make([1,3,2],id),
                 make([1,2,3],gte))).
```

Figure 8.3. Synthesized `gtePass` predicate example in COMBILOG

```
gtePass = ore(ande(gt {b->a, a->threshold},
                   isNil {nil->b}),
               ande(id,
                    gte {b->threshold, a->b})).
```

Figure 8.4. Synthesized `gtePass` predicate example in CNP, with the *proj* operator used implicitly

focus on a small number of simple examples, while real-world cases are likely to be much more complex. Regardless, when CNP is compared to its departure point of COMBILOG, it is safe to claim that CNP makes the uses of COMBILOG more accessible, and is a considerable candidate for the practical use of *Compositional Relational Programming*.

8.1 Future work

Some directions of work were identified as interesting in the course of this study but not pursued. Here is a list of some of those ideas, to make them available for further discussion and future research.

In Chapter 4 we presented a split-view editor for a side-by-side use of COMBILOG and VISUAL COMBILOG. An integrated interpreter and synthesizer in the editor would be the next step. Such an integrated tool would make further usability studies feasible, particularly those that evaluate VISUAL COMBILOG for program synthesis. The VISUAL COMBILOG notation is based on COMBILOG, but an iteration of it can be designed for CNP. VISUAL CNP would be significantly different to VISUAL COMBILOG due to the lack of argument order, introduction of argument names, and the auto-expanding logic operators. It could potentially improve usability of CNP programs further.

The CNP language is implemented as a meta-interpreter written in PROLOG. This implementation can be wrapped by a stand-alone interpreter that accepts the plain CNP programs without the meta-predicates and constructs imposed by the host environment. As well as an interpreter, a user-facing tool orchestrating the hybrid synthesis would be needed for the next step of user studies to measure the usability of CNP

for actual synthesis tasks. As an extension of such a tool, the synthesis could be improved yet further by letting the user type in a suggestive partial expression tree such as $fold(\ldots plus, \ldots)$ without regard to argument mappings, or the specific variant of the *fold* operator. The missing parts would be filled in by the synthesizer. Similar approaches were suggested in earlier work [38] and CNP could be considered a viable medium for such interactive synthesis.

In Chapter 5 we defined two extended logic operators *ande*/*ore*, and suggested some variants that may also make use of the argument names. Automatic schema matching is an already established method in the context of databases and data integration [100]. A thorough analysis of automatic schema matching methods in the context of *Logic Programming* could reveal a more suitable set of extended operators for CNP, further improving usability of the language. Empirical methods could be utilized for inventing new operators as well. A recent study has applied data mining techniques to open source code repositories in order to identify potential operators for COMBILOG with promising results [63]. A similar approach could be explored to identify new operators for CNP.

The operators *foldr*/*foldl* and their variants are the only recursion operators available in CNP. Exploring further operators for the language, and also separately for the synthesizer is advisable. The earlier work on COMBILOG refers to recursion schemes such as *divide and conquer* [47, 109], and argue for identifying further higher-order operators. The feasibility of such operators for efficient synthesis needs to be explored separately. For example, the *filter* operator is used in hand-written fragments of the hybrid synthesis examples in Chapter 7. This operator is inherently inefficient to synthesize, as it does not have a one-to-one mapping between elements in the input list and those in the output list. Decomposing synthesis examples for such an operator is combinatorially expensive. On the other hand, the synthesis complexity of a *map* operator is identical to that of the *fold*s, and it is straightforward to handle as a specific case.

Most functional languages benefit from having a static type system. It helps with identifying type-related misunderstandings at compile-time, and also accommodates working with algebraic forms of programs. CNP currently does not have a strong type system other than the separation of concepts inherited from First-Order Logic. The CNP synthesizer hints at fragments of a type system in the form of predicate valences and well-modedness constraints. It would be a worthwhile effort to explore the possibility of a strong type system for CNP , in particular as a way to formalize some aspects of the synthesizer. Moreover, a static type system could potentially improve the synthesis performance by including explicit types in the input/output examples.

In terms of usability, we have only performed quantitative studies. By design, these studies are likely to be too focused on specific tasks. Performing exploratory qualitative studies of programmer experience, particularly while dealing with other synthesis tasks could expose novel questions. Such results could be used to construct mental models for identifying more specific areas for further quantitative usability studies.

References

[1] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.

[2] John R Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2007.

[3] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.

[4] Krzysztof R Apt and Elena Marchiori. Reasoning about prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6:743–765, 1994.

[5] E. J. G. Arias, J. Lipton, J. Marino, and P. Nogueira. First-order unification using variable-free relational algebra. *Logic Journal of IGPL*, 19(6):790–820, 2011.

[6] John Bacon. The completeness of a predicate-functor logic. *The Journal of Symbolic Logic*, 50:903–926, 12 1985.

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[8] O. Banyasad and P.T. Cox. An automatic layout algorithm for lograph. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 139–146, September 2004.

[9] Omid Banyasad and Philip T. Cox. Implementing lograph. Technical report, Faculty of Computer Science, Dalhousie University, 2001. Technical report CS 2001-05.

[10] Jonas Barklund. *Parallel Unification*. PhD thesis, Computing Science Department, Uppsala University, Computing Science Dept., Box 520, SE-75120, Uppsala, Sweden, 1990.

[11] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. Synthesis of programs in computational logic. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic (Lecture Notes in Computer Science 3049)*, pages 30–65. Springer-Verlag, 2004.

[12] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235 – 282, 1994.

[13] Jacques Bertin. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin press, 1983.

[14] Alan F Blackwell. Dealing with new cognitive dimensions. In *Workshop on Cognitive Dimensions: Strengthening the Cognitive Dimensions Research Community., University of Hertfordshire*, 2000.

[15] Marat Boshernitsan and Michael Sean Downes. Visual programming languages: A survey. Computer Science Division (EECS), University of California, December 2004. Report No. UCB/CSD-04-1368.
[16] Polly S. Brown and John D. Gould. An experimental study of people creating spreadsheets. *ACM Trans. Inf. Syst.*, 5(3):258–272, 1987.
[17] Felice Cardone and J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.
[18] S. Ceri and G. Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Transactions on Software Engineering*, SE-11(4):324–345, April 1985.
[19] J. R. Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, 2004.
[20] J. R. Cheney and C. Urban. Nominal logic programming. *ACM Transactions on Programming Languages and Systems*, 30(5):26:1–26:47, 2008.
[21] Henning Christiansen. *A complete resolution method for logical meta-programming languages*, pages 205–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.
[22] Henning Christiansen. Implicit program synthesis by a reversible metainterpreter. In *International Workshop on Logic Programming Synthesis and Transformation*, pages 90–110. Springer, 1997.
[23] Henning Christiansen. Automated reasoning with a constraint-based metainterpreter. *The Journal of Logic Programming*, 37(1–3):213 – 254, 1998.
[24] Steven Clarke. Evaluating a new programming language. In *13th Workshop of the Psychology of Programming Interest Group*, pages 275–289, 2001.
[25] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
[26] Edgar F Codd. *Relational completeness of data base sublanguages*. IBM Corporation, 1972.
[27] Marco Comini, Giorgio Levi, and Maria Chiara Meo. A theory of observables for logic programs. *Information and Computation*, 169(1):23 – 80, 2001.
[28] P.T. Cox and T. Pietrzykowski. Lograph: a graphical logic programming language. In *Proceedings IEEE COMPINT 85*, pages 145–151, 1985.
[29] H. B. Curry. Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, 52(3):pp. 509–536, 1930.
[30] Haskell B Curry and Robert Feys. Combinatory logic, volume i of studies in logic and the foundations of mathematics, 1958.
[31] Jason Dagit, Joseph Lawrance, Christoph Neumann, Margaret Burnett, Ronald Metoyer, and Sam Adams. Using cognitive dimensions: advice from the trenches. *Journal of Visual Languages & Computing*, 17(4):302–327, 2006.
[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.
[33] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless

dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75/2, pages 381–392. Elsevier, 1972.
[34] Augustus De Morgan. *Formal logic or, the calculus of inference, necessary and probable*. Taylor and Walton, 1847.
[35] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *The Journal of Logic Programming*, 19:321 – 350, 1994.
[36] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. A visual interface for drawing asp programs. In *CILC*, volume 598 of *CEUR Workshop Proceedings*. CEUR, 2010.
[37] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. Aspide: Integrated development environment for answer set programming. In *Proceedings of 11th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–330. Springer, 2011.
[38] Pierre Flener. Inductive logic program synthesis with dialogs. In Stephen Muggleton, editor, *Inductive Logic Programming: 6th International Workshop, ILP-96 Stockholm, Sweden, August 26–28, 1996 Selected Papers*, pages 175–198, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
[39] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. *From Frege to Gödel: A source book in mathematical logic*, 1931:1–82, 1879.
[40] Gottlob Frege. Sense and reference. *The Philosophical Review*, 57(3):pp. 209–230, 1948.
[41] Murdoch J Gabbay and Andrew M Pitts. A new approach to abstract syntax with variable binding. *Formal aspects of computing*, 13(3-5):341–363, 2002.
[42] Thomas R.G. Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
[43] Thomas R.G. Green. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2000)*, pages 21–28. New York: ACM Press, 2000.
[44] Thomas R.G. Green and M. Petre. Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131 – 174, 1996.
[45] Andreas Hamfelt and Jørgen Fischer Nilsson. Inductive metalogic programming. In *Proceedings Fourth International Workshop on Inductive Logic programming*, pages 85–96. Bad Honnef/Bonn GMD-Studien Nr. 237, 1994.
[46] Andreas Hamfelt and Jørgen Fischer Nilsson. Declarative logic programming with primitive recursive relations on lists. In M. Maher, editor, *Procseedings of the Joint International Conference and Symposium on Logic Programming 1996*, pages 230–242. MIT Press, London, 1996.
[47] Andreas Hamfelt and Jørgen Fischer Nilsson. Towards a logic programming methodology based on higher-order predicates. *Journal of*

New Generation Computing, 15(4):421–448, 1997.

[48] Andreas Hamfelt and Jørgen Fischer Nilsson. Inductive logic programming with well-modedness constraints. In E. Rached, editor, *Proceedings of the 8th International Workshop on Functional and Logic Programming*, pages 220–231. Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universit Joseph Fourier, Laboratoire Leibniz, Institut IMAG, 1999. UMR no 5522.

[49] Andreas Hamfelt and Jørgen Fischer Nilsson. Inductive synthesis of logic programs by composition of combinatory program schemes (lecture notes in computer science 1559). In P. Flener, editor, *Procs. Workshop on Logic Based Program Transformation and Synthesis*, pages 143–158. Springer-Verlag, 1999.

[50] Andreas Hamfelt, Jørgen Fischer Nilsson, and Nikolaj Oldager. Logic program synthesis as problem reduction using combining forms. *J. Automated Software Engineering*, 8(2):167–193, 2001.

[51] Andreas Hamfelt, Jørgen Fischer Nilsson, and Görkem Pacaci. Compositional logic programming - theory and applications. *Theory and Practice of Logic Programming*, 2016. (under consideration).

[52] Andreas Hamfelt, Jørgen Fischer Nilsson, and Aida Vitoria. A combinatory form of pure logic programs and its compositional semantics. *(unpublished draft*, 1998.

[53] Michael E Hansen, Andrew Lumsdaine, and Robert L Goldstone. Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 27–38. ACM, 2012.

[54] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture: Nancy, France, September 16–19, 1985*, pages 190–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[55] C.V. Jones. *Visualization and Optimization*. Operations Research/Computer Science Interfaces Series. Springer, 1996.

[56] Katja Kevic, Braden M. Walters, Timothy R. Shaffer, Bonita Sharif, David C. Shepherd, and Thomas Fritz. Tracing software developers' eyes and interactions for change tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 202–213, New York, NY, USA, 2015. ACM.

[57] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM.

[58] Donald E. Knuth. Computer programming as an art (turing award lecture). *Communications of the ACM*, 17(12):667–673, December 1974.

[59] Andrew J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A

practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015.

[60] Robert Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979.

[61] Robert Kowalski and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2(3):227–260, 1972.

[62] J. W. Lloyd. *Foundations of Logic Programming; (2Nd Extended Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[63] Tong Luo. Towards simpler argument binding: Knowledge gathering by mining logic program repositories. Master's thesis, Informatics and Media, Uppsala University, Uppsala, Sweden, 2016.

[64] Alan M MacEachren. *How maps work: representation, visualization, and design*. Guilford Press, 1995.

[65] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, April 1986.

[66] Shane Markstrum. Staking claims: A history of programming language design claims and evidence: A positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM.

[67] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[68] Conor McBride and James McKinna. Functional pearl: I am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.

[69] R. McPhee. *Compositional Logic Programming*. PhD thesis, Oxford University Computing Laboratory, Worcester College, Oxford University, 2000.

[70] Erik Meijer. The world according to linq. *Queue*, 9(8):60:60–60:72, August 2011.

[71] Stephen Muggleton. Predicate invention and utilization. *Journal of Experimental & Theoretical Artificial Intelligence*, 6(1):121–130, 1994.

[72] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the fifth international conference on machine learning*, pages 339–352, 1992.

[73] Stephen Muggleton and Luc de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629 – 679, 1994.

[74] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[75] Marc Najork and Simon M. Kaplan. The cube language. In *VL'91*, pages 218–224, 1991.

[76] Marc Najork and Simon M. Kaplan. A prototype implementation of the cube language. In *In Proceedings of the IEEE Workshop on Visual*

Languages, pages 270–272, September 1992.

[77] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM.

[78] Shan-Hwei Nienhuys-Cheng and Ronald De Wolf. *Foundations of inductive logic programming*, volume 1228. Springer Science & Business Media, 1997.

[79] Jørgen Fischer Nilsson. Combinatory logic programming. In M. Bruynooghe, editor, *Procs. of the 2nd Workshop on Meta-programming in Logic*, pages 187–204. K.U. Leuven, Belgium, 1990.

[80] Masayuki Numao and Masamichi Shimura. Combinatory logic programming. In M. Bruynooghe, editor, *Procs. of the 2nd Workshop on Meta-programming in Logic*, pages 123–136. K.U. Leuven, Belgium, 1990.

[81] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 619–630, New York, NY, USA, 2015. ACM.

[82] Görkem Pacaci and Andreas Hamfelt. Colour beads visual representation of compositional relational programs. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 131–134, San Jose, CA, USA, 2013. IEEE conference proceedings.

[83] Görkem Pacaci and Andreas Hamfelt. A visual system for compositional relational programming. In *Proceedings of the The 23rd European Japanese Conference On Information Modelling And Knowledge Bases (EJC)*, pages 235–243, Nara, Japan, 2013. IOS Press, 2014.

[84] Görkem Pacaci, Steve McKeever, and Andreas Hamfelt. Compositional relational programming with nominal projection and compositional synthesis. In *(under consideration for) Proceedings of PSI'17: 11th Ershov Informatics Conference*, 2017.

[85] Michael S Paterson and Mark N Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186. ACM, 1976.

[86] Giuseppe Peano. *Arithmetices principia: nova methodo (The principles of arithmetic, presented by a new method)*. Fratres Bocca, 1889.

[87] Charles S Peirce. Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of boole's calculus of logic. *Memoirs of the American Academy of Arts and Sciences*, 9(2):317–378, 1873.

[88] Charles S Peirce. On junctures and fractures in logic. *Writings of Charles S. Peirce*, 1884:391, 1879.

[89] Charles S Peirce. On the algebra of logic: A contribution to the philosophy of notation. *American Journal of Mathematics*, 7(2):180–196, 1885.

[90] Charles S Peirce. *Collected papers of Charles Sanders Peirce*, volume 5. Harvard University Press, 1974.
[91] A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbay and C. J. Hogger, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1994.
[92] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation (Elsevier)*, 186:165–193, 2003.
[93] Gordon D Plotkin. A note on inductive generalization. *Machine intelligence*, 5(1):153–163, 1970.
[94] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *International Journal of Man-Machine Studies*, 36(5):741 – 773, 1992.
[95] W. V. Quine. *Word and object*. MIT press, 1960.
[96] W. V. Quine. Variables explained away. *Selected Logic Papers, Random House, New York*, pages 227–235, 1966.
[97] W. V. Quine. *Algebraic Logic and Predicate Functors*. Indianapolis, Bobbs-Merrill, 1971.
[98] W. V. Quine. Predicate-functor logic. In E. Fenstad, editor, *Procs. Second Scandinavian Logic Symposium*, pages 309–315. North-Holland, 1971.
[99] W. V. Quine. Predicate-functors revisited. *Journal of Symbolic Logic*, 46:649–652, 1981.
[100] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001.
[101] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.
[102] T Sato. Meta-programming through a truth predicate. logic programming. In *Proc. of the Joint International Conference and Symposium on Logic Programming, ed. Apt, K*, pages 526–540, 1992.
[103] M. Schönfinkel. über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3-4):305–316, 1924.
[104] S. Seres. *Algebra of Logic Programming*. PhD thesis, Oxford University Computing Laboratory, Wolfson College, Oxford University, 2001. Doctoral dissertation.
[105] S. Seres, M. Spivey, and T. Hoare. Algebra of logic programming. In *International Conference on Logic Programming*, pages 184–199. Palgrave MacMillan, 1999.
[106] Sun-Joo Shin. *The logical status of diagrams*. Cambridge University Press, Cambridge [England], 1994.
[107] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.
[108] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann.

Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 378–389, New York, NY, USA, 2014. ACM.

[109] Douglas R. Smith. The design of divide and conquer algorithms. *Science of Computer Programming*, 5:37 – 58, 1985.

[110] Tiobe Software. Tiobe programming community index. http://www.tiobe.com, September 5, 2015.

[111] Andreas Stefik, Stefan Hanenberg, Mark McKenney, Anneliese Andrews, Srinivas Kalyan Yellanki, and Susanna Siebert. What is the foundation of evidence of human factors decisions in language design? an empirical study on programming language workshops. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 223–231, New York, NY, USA, 2014. ACM.

[112] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Procs. Second International Conference on Logic Programming*, pages 127–138. Uppsala University, 1984.

[113] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(03):73–89, 1941.

[114] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[115] Alfred Tarski and Steven R Givant. *A formalization of set theory without variables*, volume 41. American Mathematical Soc., 1987.

[116] Anne M. Treisman and Garry Gelade. A feature-integration theory of attention. *Cognitive Psychology*, 12(1):97 – 136, 1980.

[117] Grigorii Samuilovich Tseitin. On the complexity of proof in prepositional calculus. *Zapiski Nauchnykh Seminarov POMI*, 8:234–259, 1968.

[118] J.D. Ullman. *Principles of database systems*. Computer software engineering series. Computer Science Press, 1980.

[119] Johan van Benthem and Alice Ter Meulen. *Handbook of logic and language*. Elsevier, 1996.

[120] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

[121] D. H. D. Warren. Higher-order extensions to prolog: are they needed ? In D. Michie, editor, *Machine Intelligence 10*, pages 441–454. Ellis Horwood and Edinburgh University Press, 1982.

[122] Jeremy M. Wolfe. Visual search. *Current Biology*, 20(8):R346 – R349, 2010.

[123] Young Seok Yoon and Brad A. Myers. Supporting selective undo in a code editor. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 223–233, Piscataway, NJ, USA, 2015. IEEE Press.

[124] Andreas Zetterström. Visual compositional-relational programming. Master thesis, Uppsala University, Computer Systems Sciences, 2010.

Appendix A.
VISUAL COMBILOG user study Group A test

EVALUATION OF A VISUAL SYSTEM FOR RELATIONAL PROGRAMMING

We need your help to understand if a particular type of visual assistance improves understanding of certain programming expressions. This questionnaire is composed of 4 parts.

PART 1

INTRODUCING CONSTRUCTS OF CODE

Our programming examples deal with relations and composing new relations using a set of operators. A relation represents a connection between a number of arguments, and the number of values is called the 'arity' of that relation. For example, a relation R with three arguments is an arity-3 relation:

$$\boldsymbol{r}(\boldsymbol{A_0}, \boldsymbol{A_1}, \boldsymbol{A_2})$$

The first operator we deal with is the *make* operator, which can manipulate the arguments of a given relation to produce a new one. It takes an index list and a relation for arguments and produces a new relation where arguments are ordered with the given index list. For example:

$$\boldsymbol{newRelation} \leftarrow \boldsymbol{make}([\mathbf{2}, \mathbf{1}, \mathbf{0}], \boldsymbol{relationA}).$$

Assuming the *relationA* is an arity-3 relation, the statement above inverses the argument order of the *relationA.* To help understanding what the *make* operator does, another way of writing the statement above could be as follows:

$$\boldsymbol{newRelation}(\boldsymbol{A_2}, \boldsymbol{A_1}, \boldsymbol{A_0}) \leftarrow \boldsymbol{relationA}(\boldsymbol{A_0}, \boldsymbol{A_1}, \boldsymbol{A_2})$$

But with the use of *make* operator, we avoid explicitly displaying arguments but rely on an index list (such as [2, 1, 0]). This index list can also be used to introduce new arguments by placing the *underscore* sign '_'. Such an example follows:

$$\boldsymbol{newRelation} \leftarrow \boldsymbol{make}([\_, \mathbf{1}, \_], \boldsymbol{relationA}).$$

The example above creates a *newRelation* where the argument-0 and argument-2 (first and last arguments) are unbound, and the argument-1 is bound to argument-1 of the *relationA.* The *make* operator can also be used in a nested fashion. An example that does the same thing as above example can be written as follows:

$$\boldsymbol{newRelation} \leftarrow \boldsymbol{make}([\_, \mathbf{0}, \_], \boldsymbol{make}([\mathbf{1}], \boldsymbol{relationA})).$$

In the example above, the outer *make* operator refers to the middle argument of *relationA* as argument-0 since the inner *make* produces a temporary relation with only one argument, which has only one argument, hence argument-0. There are also logical operators *and* and *or* that you can combine with *make.* These operators take other expressions as input to combine them for a logical expression. Every argument in the new relation is bound to the arguments at the same index in every relation in that context. For example:

$$\boldsymbol{newRelation} \leftarrow \boldsymbol{and}(\boldsymbol{relationA}, \boldsymbol{relationB}).$$

In the example above, if we assume the *relationA* and *relationB* are arity-3, the *newRelation* also has to be arity-3, since logic operators only combine relations of the same arity to produce a new relation of that same arity. Argument-0 of *relationA* is bound to argument-0 of *newRelation* and argument-0 of *relationB.* Same goes for argument-1 and argument-2. Another example that is one step more complex. *relationA* is arity-2 and *relationB* is arity-3:

$$\boldsymbol{newRelation} \leftarrow \boldsymbol{make}\left([\mathbf{1}], \boldsymbol{or}\left(\boldsymbol{relationA}, \boldsymbol{make}([\mathbf{1}, \mathbf{0}], \boldsymbol{relationB})\right)\right).$$

The statement above creates a *newRelation* that is arity-1. And this single argument of *newRelation*, argument-0, is bound to argument-1 of *relationA* and argument-0 of *relationB*, since the inner *make* reverses the places of argument-1 and argument-0 of *relationB*.

EXERCISES WITH ONLY CODE

In the next section, we will show you some code and ask a few questions about its consequences. Please do not forget to write down the time.

Start time: . . / . . /. . (hh/mm/ss)

1. $\boldsymbol{newRelation} \leftarrow \boldsymbol{make}([\mathbf{0}, \mathbf{2}], \boldsymbol{relationA})$.

   Assuming ***relationA*** **is arity-3**, please answer these questions:
   a. How many arguments does ***newRelation*** have?
      Answer:
   b. Argument-0 of ***newRelation*** is bound to which argument of ***relationA***?
      Answer:
   c. Is any argument of ***newRelation*** bound to argument-2 of ***relationA***?
      Answer:

2. $\boldsymbol{newRelation} \leftarrow \boldsymbol{and}(\boldsymbol{make}([\mathbf{2}, \mathbf{1}], \boldsymbol{relationA}),\ \boldsymbol{make}([\mathbf{3}, \mathbf{0}], \boldsymbol{relationB}))$.

   Assuming ***relationA*** **is arity-3** and ***relationB*** **is arity-4**:
   a. How many arguments does ***newRelation*** have?
      Answer:
   b. Argument-0 of ***newRelation*** is bound to which argument of ***relationA***?
      Answer:
   c. Argument-0 of ***newRelation*** is bound to which argument of ***relationB***?
      Answer:

3. $\boldsymbol{newRelation} \leftarrow \boldsymbol{make}\big([\_, \mathbf{1}, \mathbf{0}], \boldsymbol{and}(\boldsymbol{make}([\mathbf{2}, \mathbf{1}], \boldsymbol{relationA}),\ \boldsymbol{make}([\mathbf{3}, \mathbf{0}], \boldsymbol{relationB}), \boldsymbol{relationC})\big)$.

   Assuming ***relationA*** **is arity-3,** ***relationB*** **is arity-4** and ***relationC*** **is arity-2**:
   a. How many arguments does ***newRelation*** have?
      Answer:
   b. Argument-0 of ***newRelation*** is bound to which argument of ***RelationA***?
      Answer:
   c. Argument-0 of ***newRelation*** is bound to which argument of ***RelationC***?
      Answer:
   d. Argument-2 of ***newRelation*** is bound to arguments of what other relations?
      Answer:
   e. Argument-1 of ***relationA*** is bound to which argument of ***relationB***?
      Answer:

Finish time: . ./. . /. . (hh/mm/ss)

PART 2

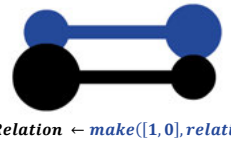
Warning: DO NOT go back to previous section to correct your answers.

In this part, we display a line of code and we also provide visuals corresponding to every line of code.

Relations are displayed by circles connected to each other with lines. A new relation that is being created will always be displayed in black in the middle, its arguments ordered from left to right. Solid circles (on the left, below) are used for ***and*** operator while dashed circles (on the right, below) represent ***or***.

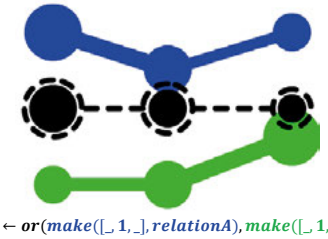
Other relations used in the expression will be displayed in different colours, largest circle of the same colour being argument-0 and the smallest circle being the last argument.

In the example above, the red relation can be written as $r(A_0, A_1, A_2)$ and blue relation as $b(A_1, A_0)$. Lastly, the circles that are touching are bound together. An example of this can be drawn as below:



$$newRelation \leftarrow make([1, 0], relationA)$$

In the example above, ***newRelation*** is created by inversing the arguments of ***relationA***. Argument-0 of ***newRelation*** is bound to argument-1 of ***relationA*** and argument-1 of ***newRelation*** is bound to argument-0 of ***relationA***. The next example is an ***or*** operation:



$$newRelation \leftarrow or(make([\_, 1, \_], relationA), make([\_, 1, \_], relationB))$$

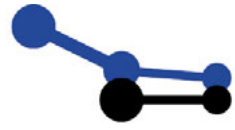
In this example, the ***newRelation*** will be an arity-3 relation. Argument-1 of ***newRelation*** is bound to argument-1 of ***relationA***. Argument-2 of ***newRelation*** is bound to argument-0 of ***relationB***.

In the next section, a line of code will be displayed alongside its visualization. Please answer the questions, and write down the time when you start and finish.

EXERCISES WITH VISUALS

Start time: . . / . . /. . (hh/mm/ss)

1. $\boldsymbol{newRelation} \leftarrow \boldsymbol{make}([\mathbf{1},\mathbf{2}],\boldsymbol{relationA})$.



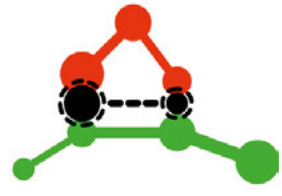
Assuming ***relationA*** **is arity-3**, please answer these questions:

d. How many arguments does ***newRelation*** have?
Answer:

e. Argument-0 of ***newRelation*** is bound to which argument of ***relationA***?
Answer:

f. Is any argument of ***newRelation*** bound to argument-0 of ***relationA***?
Answer:

2. $\boldsymbol{newRelation} \leftarrow \boldsymbol{or}(\boldsymbol{make}([\mathbf{0},\mathbf{2}],\boldsymbol{relationA}),\ \boldsymbol{make}([\mathbf{2},\mathbf{1}],\boldsymbol{relationB}))$.



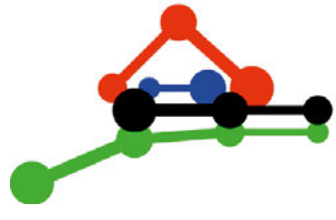
Assuming ***relationA*** **is arity-3** and ***relationB*** **is arity-4**:

d. How many arguments does ***newRelation*** have?
Answer:

e. Argument-0 of ***newRelation*** is bound to which argument of ***relationA***?
Answer:

f. Argument-0 of ***newRelation*** is bound to which argument of ***relationB***?
Answer:

3. $\boldsymbol{newRelation} \leftarrow$
$\boldsymbol{make}([\mathbf{1},\mathbf{0},\mathbf{2}],\boldsymbol{and}(\boldsymbol{make}([\mathbf{0},\mathbf{2},\_],\boldsymbol{relationA}),\ \boldsymbol{make}([\mathbf{2},\mathbf{1},\mathbf{3}],\boldsymbol{relationB}),\boldsymbol{make}([\mathbf{0},\mathbf{1},\_],\boldsymbol{relationC}))$.



Assuming ***relationA*** **is arity-3,** ***relationB*** **is arity-4** and ***relationC*** **is arity-2**:

f. How many arguments does ***newRelation*** have?
Answer:

g. Argument-0 of ***newRelation*** is bound to which argument of ***RelationA***?
Answer:

h. Argument-0 of ***newRelation*** is bound to which argument of ***RelationC***?
Answer:

i. Argument-2 of ***newRelation*** is bound to arguments of what other relations?
Answer:

j. Argument-1 of ***relationA*** is bound to which argument of ***relationB***?
Answer:

Finish time: . ./. . /. . (hh/mm/ss)

QUESTIONNAIRE

Please answer these questions:

"When accompanied with visuals, the code was easier to understand."

Strongly agree Agree Neutral Disagree Strongly Disagree

"When accompanied with visuals, I could interpret the code faster."

Strongly agree Agree Neutral Disagree Strongly Disagree

Are you familiar with any relational programming language? (such as Prolog):

Do you have a vision problem? Please specify:

Do you have a condition that makes it harder for you to read text? Please specify:

Do you have any issues perceiving colours? Please specify:

Your age:

Your gender (optional):

Please write down an e-mail address if you would like to be notified of the results:

Thank you for your time, your contribution is appreciated greatly.
Regards,
Görkem Pacaci

Appendix B.
COMBILOG interpreter

```
% Runs on: SWI−Prolog (MacOs 10.12, 64 bits, Version 7.3.20)
% Runs on: SICStus Prolog 4.2.0 (MacOs 10.12, x86_64−darwin−10.6.0)

:- use_module(library(lists)).
:- multifile defPredicate/2.

% elementary predicates
comb(true, []).
comb(isNil, [[]]).
comb(const(C), [C]).
comb(id, [X,X]).
comb(cons, [X,Y,[X|Y]]).

% natural number predicates given besides the
% elementary predicates of the language
comb(gt, [X, Y]) :- number(X), number(Y), X>Y.
comb(gte, [X, Y]) :- number(X), number(Y), X>=Y.
comb(asList, [X, [X]]).

comb(and(P, Q), Args) :- comb(P, Args),
                         comb(Q, Args).
comb(and(P, Q, R), Args) :- comb(P, Args),
                            comb(Q, Args),
                            comb(R, Args).
comb(and(P, Q, R, S), Args) :- comb(P, Args),
                               comb(Q, Args),
                               comb(R, Args),
                               comb(S, Args).

comb(or(P, Q), Args) :- comb(P, Args); comb(Q, Args).
comb(or(P, Q, R), Args) :- comb(P, Args);
                           comb(Q, Args);
                           comb(R, Args).
comb(or(P, Q, R, S), Args) :- comb(P, Args);
                              comb(Q, Args);
                              comb(R, Args);
                              comb(S, Args).

```

```
comb(make(I, P), Atarget) :- arity(P, PArity),
                             makebind(I, Asource, PArity, Atarget),
                            comb(P, Asource).

comb(foldr(_, Q), [Y, [], Z]) :- comb(Q, [Y, Z]).
comb(foldr(P, Q), [Y, [X|T], W]) :- comb(foldr(P, Q), [Y, T, Z]),
                                comb(P, [X, Z, W]).

% the binary variant of foldr
comb(foldr2(P, Q), Args) :-
  comb(make([2,3], foldr(P, and(make([1,2],Q),id))), Args).

comb(foldr(_), [[X|[]], X]).
comb(foldr(P), [[X|Xr], Y]) :-
  comb(foldr(P), [Xr,Z]),
  comb(P, [X,Z,Y]).

comb(natrec(_, Q), [Y, [], Z]) :- comb(Q, [Y, Z]).
comb(natrec(P, Q), [Y, [[]|T], W]) :- comb(natrec(P,Q), [Y,T,V]),
                                      comb(P, [Y,V,W]).
comb(natrec(P, Q), [Y, [[]|T], W]) :- comb(natrec(P,Q), [Y,T,V]),
                                      comb(P, [Y,V,W]).

comb(foldl(_, Q), [Y, [], Z]) :- comb(Q, [Y, Z]).
comb(foldl(P, Q), [Y, [X|T], W]) :- comb(P, [X, Y, Z]),
                                   comb(foldl(P, Q), [Z, T, W]).

% the defPredicate provides a library predicate, where predicate
% names are mapped to their bodies as predicate expressions.
% if the interpreter is called with a first term that is atomic,
% this clause does the library lookup for a predicate body,
% and if it finds one, executes it.
comb(Pred, Args) :-
  atomic(Pred),
  defPredicate(Pred, Body),
  comb(Body, Args).

% helper predicate for 'make'
% makebind(Indices, SourceArgs, AritySource, TargetArgs)
makebind([], S, ArityS, []) :- length(S, ArityS).
makebind([I|Ir], S, ArityS, [T|Tr]) :- I=<ArityS,
                                       nth1(I, S, T),
                                       makebind(Ir, S, ArityS, Tr).
makebind([I|Ir], S, ArityS, [_|Tr]) :- I>ArityS,
                                       makebind(Ir, S, ArityS, Tr).

% arity calculation is needed for the n-ary implementation of 'make'.
```

```
arity(true, 0).
arity(isNil, 1).
arity(const(_), 1).
arity(id, 2).
arity(gt, 2).
arity(gte, 2).
arity(cons, 3).
arity(and(P, Q), N) :- arity(P, N), arity(Q, N).
arity(and(P, Q, R), N) :- arity(P, N), arity(Q, N), arity(R, N).
arity(and(P, Q, R, S), N) :- arity(P, N), arity(Q, N),
                             arity(R, N), arity(S, N).
arity(or(P, Q), N) :- arity(P, N), arity(Q, N).
arity(or(P, Q, R), N) :- arity(P, N), arity(Q, N), arity(R, N).
arity(or(P, Q, R, S), N) :- arity(P, N), arity(Q, N),
                            arity(R, N), arity(S, N).
arity(make(I,_), N) :- length(I, N).
arity(foldr(_,_), 3).
arity(foldl(_,_), 3).
arity(foldr2(_, _), 2).
arity(natrec(_,_),3).
arity(foldr(_), 2).
arity(Pred, N) :-
  atomic(Pred),
  defPredicate(Pred, Body),
  arity(Body, N).
```

Appendix C.
CNP interpreter

```
% Runs on: SWI-Prolog (MacOs 10.12, 64 bits, Version 7.3.20)
% Runs on: SICStus Prolog 4.2.0 (MacOs 10.12, x86_64-darwin-10.6.0)

:- use_module(library(sets)).
?- use_module(library(lists)).
:- op(500, xfx, [->,:,/]).
:- multifile defPredicate/2.

% CNP elementary predicates
cnp(true, []).
cnp(const(N, C), [N:C]).
cnp(id, [a:X, b:X]).
cnp(cons, [a:X, b:Y, ab:[X|Y]]).

% these are not elementary predicates of the language,
% and can be defined using the elementary predicates above
% but for ease of use they are included here as a minimal library.
cnp(isNil, [nil:[]]).
cnp(gt, [a:X, b:Y]) :- number(X), number(Y), X>Y.
cnp(gte, [a:X, b:Y]) :- number(X), number(Y), X>=Y.

cnp(debug(Mess), [obj:Obj]) :-
  write(Mess), write(":"), writeln(Obj).

cnp(ande(A, B), Args) :-
  names(A, NamesA), names(B, NamesB),
  names(ande(A, B), NamesAB),
  subtract(NamesAB, NamesB, NamesOnlyA),
  subtract(NamesAB, NamesOnlyA, NamesBOrd),
  splitArgs(Args, NamesA, NamesBOrd, ArgsA, ArgsBOrd),
  reorderArgs(ArgsBOrd, NamesB, ArgsB),
  cnp(A, ArgsA),
  cnp(B, ArgsB).
cnp(ande(A,B,C), Args) :-
  cnp(ande(ande(A,B),C), Args).
cnp(ande(A,B,C,D), Args) :-
  cnp(ande(ande(A,B), ande(C,D)), Args).
cnp(ande(A,B,C,D,E), Args) :-
```

```
39   cnp(ande(ande(A,B), ande(C,D), E), Args).
40
41 cnp(ore(P, Q), Args) :-
42   names(P, NamesP),
43   names(Q, NamesQ),
44   names(ore(P, Q), NamesPQ),
45   subtract(NamesPQ, NamesQ, NamesOnlyP),
46   subtract(NamesPQ, NamesOnlyP, NamesQOrd),
47   splitArgs(Args, NamesP, NamesQOrd, ArgsP, ArgsQOrd),
48   reorderArgs(ArgsQOrd, NamesQ, ArgsQ),
49   (cnp(P, ArgsP); (cnp(Q, ArgsQ))).
50 cnp(ore(P,Q,R), Args) :- cnp(ore(ore(P,Q),R), Args).
51 cnp(ore(P,Q,R,S), Args) :- cnp(ore(ore(P,Q),ore(R,S)), Args).
52 cnp(ore(P,Q,R,S,T), Args) :- cnp(ore(ore(P,Q),ore(R,S),T), Args).
53
54 cnp(Q/Projs, Args) :-
55   cnp(proj(Q, Projs), Args).
56 cnp(proj(Q, Projs), Args) :-
57   names(Q, NamesQ),
58   namesInProj(Projs, SourceNames, _),
59   renameArgs(ArgsToProject, Projs, Args),
60   expand(ArgsToProject, SourceNames, NamesQ, ArgsQ),
61   cnp(Q, ArgsQ).
62
63 % foldr/foldl :
64 % - The arg. names for the base case are [a, b]
65 % - The arg. names for the recursive case are [a, b, ab]
66 % - The resulting predicates argument names are [a0, as, b]
67 cnp(foldr(_, Q), [a0:A0, as:[], b:B]) :-
68   cnp(Q, [a:A0, b:B]).
69 cnp(foldr(P, Q), [a0:A0, as:[A|As], b:B]) :-
70   cnp(foldr(P, Q), [a0:A0, as:As, b:Bmid]),
71   cnp(P, [a:A, b:Bmid, ab:B]).
72
73 cnp(foldl(_, Q), [a0:A0, as:[], b:B]) :-
74   cnp(Q, [a:A0, b:B]).
75 cnp(foldl(P, Q), [a0:A0, as:[A|As], b:B]) :-
76   cnp(P, [a:A, b:A0, ab:Bmid]),
77   cnp(foldl(P, Q), [a0:Bmid, as:As, b:B]).
78
79 % Constraints for using foldr2 :
80 % - The argument name for Q is only [b]
81 % - The argument names for P are [a, b, ab]
82 % - The resulting arguments are [as, b]
83 cnp(foldr2(_, Q), [as:[], b:B]) :-
84   cnp(Q, [b:B]).
85 cnp(foldr2(P, Q), [as:[A|As], b:B]) :-
```

```
86   cnp(foldr2(P, Q), [as:As, b:Bmid]),
87   cnp(P, [a:A, b:Bmid, ab:B]).
88
89 % Constraints for using natrec:
90 % - The argument names for Q are: a, b
91 % - The argument names for P are: a, b, ab
92 % - The resulting arguments are: a0, as, b.
93 cnp(natrec(_, Q), [a0:A0, as:[], b:B]) :-
94   cnp(Q, [a:A0, b:B]).
95 cnp(natrec(P, Q), [a0:A0, as:[[]|As], b:B]) :-
96   cnp(natrec(P,Q), [a0:A0, as:As, b:Bmid]),
97   cnp(P, [a:A0, b:Bmid, ab:B]).
98 % variant with binary recursive case and unary base case
99 cnp(natrec(_, Q), [a0:_, as:[], b:B]) :-
100  cnp(Q, [b:B]).
101 cnp(natrec(P, Q), [a0:A0, as:[[]|As], b:B]) :-
102  cnp(natrec(P,Q), [a0:A0, as:As, b:Bmid]),
103  cnp(P, [b:Bmid, ab:B]).
104
105 % filter :
106 % - The arg. names for the recursive case are [a, param, b]
107 % - The resulting predicates arg. names are [as, param, bs]
108 cnp(filter(_), [as:[], param:_, bs:[]]).
109 cnp(filter(P), [as:[A|As], param:Param, bs:Bs]) :-
110  cnp(P, [a:A, param:Param, b:[]]),
111  cnp(filter(P), [as:As, param:Param, bs:Bs]).
112 cnp(filter(P), [as:[A|As], param:Param, bs:[B|Bs]]) :-
113  cnp(P, [a:A, param:Param, b:B]),
114  dif(B, []),
115  cnp(filter(P), [as:As, param:Param, bs:Bs]).
116
117 % user predicates are defined with defPredicate(name, Body).
118 cnp(Pred, Args) :-
119  atomic(Pred), defPredicate(Pred, Body), cnp(Body, Args).
120
121 % cnp with unordered arguments.
122 % it's more efficient to implement this
123 % only as a user-facing separate predicate, as below.
124 cunp(E, UArgs) :- cnp(E, Args), permutation(Args, UArgs).
125
126 % projNames(Projs, SourceNames, NewNames)
127 namesInProj([], [], []).
128 namesInProj([A->B|Projs], [A|ARest], [B|BRest]) :-
129  namesInProj(Projs, ARest, BRest).
130 namesInProj([A|Projs], SourceNames, NewNames) :-
131  atomic(A),
132  namesInProj([A->A|Projs], SourceNames, NewNames).
```

```
133
134 renameArgs([], [], []).
135 renameArgs([A:V|SourceRest], Projs, [B:V|NewArgsRest]) :-
136   select(A->B, Projs, ProjsRest),
137   renameArgs(SourceRest, ProjsRest, NewArgsRest).
138 renameArgs(SourceArgs, [A|ProjRest], NewArgs) :-
139   atomic(A),
140   renameArgs(SourceArgs, [A->A|ProjRest], NewArgs).
141
142 %reorderArgs(Args, NamesOrd, ArgsOrd)
143 reorderArgs([], [], []).
144 reorderArgs(Args, [N|NamesOrd], [N:V|ArgsOrd]) :-
145   select(N:V, Args, ArgsRest),
146   reorderArgs(ArgsRest, NamesOrd, ArgsOrd).
147
148 % calculating the Names for a cnp expression E.
149 % names(E, Names).
150 names(true, []).
151 names(isNil, [nil]).
152 names(const(N, _), [N]).
153 names(id, [a, b]).
154 names(cons, [a, b, ab]).
155 % names for numeric operators
156 names(gt, [a, b]).
157 names(gte, [a, b]).
158 names(debug(_), [obj]).
159
160 names(Q/Projs, Names) :-
161   names(proj(Q,Projs), Names).
162 names(proj(_, []), []).
163 names(proj(_, [_->B|Projs]), [B|Names]) :-
164   names(proj(_, Projs), Names).
165 names(proj(_, [A|Projs]), [A|Names]) :-
166   atomic(A),
167   names(proj(_, Projs), Names).
168
169 names(ande(A,B), Names) :-
170   unionNames(A, B, Names).
171 names(ande(A,B,C), Names) :-
172   unionNames(ande(A,B), C, Names).
173 names(ande(A,B,C,D), Names) :-
174   unionNames(ande(A,B), ande(C,D), Names).
175 names(ande(A,B,C,D,E), Names) :-
176   unionNames(ande(A,B,C), ande(D,E), Names).
177
178 names(ore(A,B), Names) :-
179   unionNames(A, B, Names).
```

```
180 names(ore(A,B,C), Names) :-
181   unionNames(ore(A,B), C, Names).
182 names(ore(A,B,C,D), Names) :-
183   unionNames(ore(A,B), ore(C,D), Names).
184 names(ore(A,B,C,D,E), Names) :-
185   unionNames(ore(A,B,C), ore(D,E), Names).
186
187 names(foldr(_,_), [a0, as, b]).
188 names(natrec(_,_), [a0, as, b]).
189 names(foldr2(_,_), [as, b]).
190 names(foldl(_,_), [a0, as, b]).
191 names(filter(_), [as, param, bs]).
192
193 names(Pred, Names) :-
194   atomic(Pred),
195   defPredicate(Pred, Body),
196   names(Body, Names).
197
198 argNames([], []).
199 argNames([N|Nrest], [N:_|Arest]) :-
200   argNames(Nrest, Arest).
201
202 % unifySeqArgs(ArgsA, ArgsB, Args).
203 unifySeqArgs([], [], []).
204 unifySeqArgs([], [N:V|ArgsB], [N:V|Args]) :-
205   unifySeqArgs([], ArgsB, Args).
206 unifySeqArgs([N:V|ArgsA], [], [N:V|Args]) :-
207   unifySeqArgs(ArgsA, [], Args).
208 unifySeqArgs([N:V|ArgsA], [N:V|ArgsB], [N:V|Args]) :-
209   unifySeqArgs(ArgsA, ArgsB, Args).
210 unifySeqArgs([N:V|ArgsA], [Nb:Vb|ArgsB], [N:V|Args]) :- N\=Nb,
211 unifySeqArgs(ArgsA, [Nb:Vb|ArgsB], Args).
212
213 % splitArgs(Args, NamesA, NamesB, ArgsA, ArgsB).
214 % requires namesa and namesb in the same order
215 % as they are in args.
216 splitArgs([], [], [], [], []).
217 splitArgs([N:V|Args], [], [N|NamesB], [], [N:V|ArgsB]) :-
218   splitArgs(Args, [], NamesB, [], ArgsB).
219 splitArgs([N:V|Args], [N|NamesA], [], [N:V|ArgsA], []) :-
220   splitArgs(Args, NamesA, [], ArgsA, []).
221 splitArgs([N:V|Args], [N|NamesA], [N|NamesB], [N:V|ArgsA],
222           [N:V|ArgsB]) :-
223   splitArgs(Args, NamesA, NamesB, ArgsA, ArgsB).
224 splitArgs([N:V|Args], [N|NamesA], [Nb|NamesB], [N:V|ArgsA],
225           ArgsB) :-
226   N\=Nb,
```

```
227   splitArgs(Args, NamesA, [Nb|NamesB], ArgsA, ArgsB).
228 splitArgs([N:V|Args], [Na|NamesA], [N|NamesB], ArgsA,
229           [N:V|ArgsB]) :-
230   N\=Na,
231   splitArgs(Args, [Na|NamesA], NamesB, ArgsA, ArgsB).
232
233 unionNames(A, B, Names) :-
234   names(A, NamesA),
235   names(B, NamesB),
236   subtract(NamesB, NamesA, NamesBOnly),
237   append(NamesA, NamesBOnly, Names).
238
239 % expands the given Args to a new ExpandedArgs
240 % that contains an unbound arg for every new name.
241 % expand(Args, ArgNames, Names, ExpandedArgs)
242 expand(Args, ArgNames, [], []) :-
243   argNames(ArgNames, Args).
244 expand(Args, ArgNames, [N|Names], [N:V|EArgs]) :-
245   member(N:V, Args),
246   expand(Args, ArgNames, Names, EArgs).
247 expand(Args, ArgNames, [N|Names], [N:_|EArgs]) :-
248   \+member(N:_, Args),
249   expand(Args, ArgNames, Names, EArgs).
```

Appendix D.
CNP synthesizer

```
% Runs on: SWI−Prolog (MacOs 10.12, 64 bits, Version 7.3.20)
% Runs on: SICStus Prolog 4.2.0 (MacOs 10.12, x86_64−darwin−10.6.0)

:- use_module(library(lists)).
:- use_module(library(aggregate)).
:- use_module(library(clpfd)).
:- multifile valencePredicate/2.

% Entry point
synInc(Prog, ValenceProg, Examples) :-
  range(1, 6, Depth),
  write('Attempting depth='), writeln(Depth),
  syn(Prog, [], Depth, ValenceProg, Examples).

% Syn for a given depth
syn(Pred, _, Depth, ValencePred, Examples) :-
  Depth>=1,
  valencePredicate(Pred, ValencePred),
  testExamplesPos(Pred, Examples).

syn(ore(P,Q), [], Depth, ValenceOr, Examples) :-
  Depth>=2,
  SubDepth is Depth-1,
  ValenceP=ValenceOr, ValenceQ=ValenceOr,
  append(ExamplesP, ExamplesQ, Examples),
  ExamplesP\=[], ExamplesQ\=[],
  syn(P, ore, SubDepth, ValenceP, ExamplesP),
  syn(Q, ore, SubDepth, ValenceQ, ExamplesQ).

syn(ande(P, Q), _, Depth, ValenceAnd, Examples) :-
  Depth>=2,
  SubDepth is Depth-1,
  valenceAnde(ValenceAnd, ValenceP, ValenceQ),
  argNames(NamesP, ValenceP),
  argNames(NamesQ, ValenceQ),
  splitExamples(Examples, NamesP, NamesQ, ExamplesP, ExamplesQ),
  syn(P, ande, SubDepth, ValenceP, ExamplesP),
  syn(Q, ande, SubDepth, ValenceQ, ExamplesQ),
```

```
39   P\=Q.
40
41 syn(proj(Q,Projs), ParentOp, Depth, ValenceProj, Examples) :-
42   ParentOp\=proj,
43   Depth>=2,
44   SubDepth is Depth-1,
45   valenceProj(ValenceProj, Projs, 0, ValenceQ),
46   argNamesDifferent(ValenceQ),
47   unprojExsToVal(Examples, Projs, ValenceProj, ValenceQ, ExUnproj),
48   syn(Q, proj, SubDepth, ValenceQ, ExUnproj).
49
50 syn(natrec(P,Q), ParentOp, Depth, ValenceNatrec, Examples) :-
51   Depth>=2,
52   SubDepth is Depth-1,
53   dif(ParentOp, ore),
54   valenceNatrec(ValenceNatrec, ValenceP, ValenceQ),
55   syn_natrec(P, ValenceP, Q, ValenceQ, SubDepth, Examples).
56
57 syn(foldr2(P,Q), ParentOp, Depth, ValenceFoldr2, Examples) :-
58   Depth>=2,
59   SubDepth is Depth-1,
60   dif(ParentOp, ore),
61   valenceFoldr2(ValenceFoldr2, ValenceP, ValenceQ),
62   syn_foldr2(P, ValenceP, Q, ValenceQ, SubDepth, Examples).
63
64 syn(foldr(P,Q), ParentOp, Depth, ValenceFoldr, Examples) :-
65   Depth>=2,
66   SubDepth is Depth-1,
67   dif(ParentOp, ore), dif(ParentOp, foldr),
68   valenceFoldr(ValenceFoldr, ValenceP, ValenceQ),
69   syn_foldr(P, ValenceP, Q, ValenceQ, SubDepth, Examples).
70
71 syn_natrec(_, _, _, _, _, []).
72 syn_natrec(P, ValP, Q, ValQ, SubDepth,
73           [[a0:A0, as:[], b:B]|ExRest]) :-
74   syn(Q, natrec, SubDepth, ValQ, [[a:A0, b:B]]),
75   syn_natrec(P, ValP, Q, ValQ, SubDepth, ExRest).
76 syn_natrec(P, ValP, Q, ValQ, SubDepth,
77           [[a0:A0, as:[[]|As], b:B]|ExRest]) :-
78   syn_natrec(P, ValP, Q, ValQ, SubDepth,
79             [[a0:A0, as:As, b:Bmid]|ExRest]),
80   syn(P, natrec, SubDepth, ValP, [[a:A0, b:Bmid, ab:B]]).
81 % variant with binary recursive case
82 syn_natrec(P, ValP, Q, ValQ, SubDepth,
83           [[a0:A0, as:[[]|As], b:B]|ExRest]) :-
84   syn_natrec(P, ValP, Q, ValQ, SubDepth,
85             [[a0:A0, as:As, b:Bmid]|ExRest]),
```

```
86   syn(P, natrec, SubDepth, ValP, [[b:Bmid, ab:B]]).
87
88 syn_foldr2(_, _, _, _, _, []).
89 syn_foldr2(P, ValP, Q, ValQ, SubDepth, [[as:[], b:B]|ExRest]) :-
90   syn(Q, foldr2, SubDepth, ValQ, [[b:B]]),
91   syn_foldr2(P, ValP, Q, ValQ, SubDepth, ExRest).
92 syn_foldr2(P, ValP, Q, ValQ, SubDepth,
93            [[as:[A|As], b:B]|ExRest]) :-
94   syn_foldr2(P, ValP, Q, ValQ, SubDepth,
95              [[as:As, b:Bmid]|ExRest]),
96   syn(P, foldr2, SubDepth, ValP, [[a:A, b:Bmid, ab:B]]).
97
98 syn_foldr(_, _, _, _, _, []).
99 syn_foldr(P, ValP, Q, ValQ, SubDepth,
100          [[a0:A0, as:[], b:B]|ExRest]) :-
101   syn(Q, foldr, SubDepth, ValQ, [[a:A0, b:B]]),
102   syn_foldr(P, ValP, Q, ValQ, SubDepth, ExRest).
103 syn_foldr(P, ValP, Q, ValQ, SubDepth,
104          [[a0:A0, as:[A|As], b:B]|ExRest]) :-
105   syn_foldr(P, ValP, Q, ValQ, SubDepth,
106            [[a0:A0, as:As, b:Bmid]|ExRest]),
107   syn(P, foldr, SubDepth, ValP, [[a:A, b:Bmid, ab:B]]).
108
109 % tests elementary predicates for given examples.
110 testExamplesPos(_, []).
111 testExamplesPos(P, [E|Er]) :-
112   once(cnp(P, E)),
113   testExamplesPos(P, Er).
114
115 rangeList(Lo, Up, []) :- Lo>Up.
116 rangeList(Lo, Up, [Lo|Ns]) :-
117   Lo=<Up,
118   Lo1 is Lo+1,
119   rangeList(Lo1, Up, Ns).
120
121 range(Lo, Up, N) :- rangeList(Lo, Up, Ns), member(N, Ns).
122
123 %splitExamples(Examples, NamesP, NamesQ, ExamplesP, ExamplesQ).
124 splitExamples([], _, _, [], []).
125 splitExamples([E|Examples], NamesP, NamesQ, [Ep|ExamplesP],
126              [Eq|ExamplesQ]) :-
127   splitArgs(E, NamesP, NamesQ, Ep, Eq),
128   splitExamples(Examples, NamesP, NamesQ, ExamplesP, ExamplesQ).
129
130 % map unprojToValence to a list of examples.
131 unprojExsToVal([], _, _, _, []).
132 unprojExsToVal([E|Examples], Projs, ValenceProj, ValenceUnproj,
```

```
133               [Eun|ExamplesUnproj]) :-
134   unprojToValence(E, Projs, ValenceProj, ValenceUnproj, Eun),
135   unprojExsToVal(Examples, Projs, ValenceProj, ValenceUnproj,
136                  ExamplesUnproj).
137
138 % reverse-apply projections to a valence.
139 unprojToValence([], [], [], [], []).
140 unprojToValence(ArgsProj, Projs, ValenceProj,
141                 [Norig:IO|ValenceOrig], [Norig:V|ArgsOrig]) :-
142   select(Norig->Nproj, Projs, ProjsRest),
143   select(Nproj:V, ArgsProj, ArgsProjRest),
144   select(Nproj:IO, ValenceProj, ValenceProjRest),
145   unprojToValence(ArgsProjRest, ProjsRest, ValenceProjRest,
146                   ValenceOrig, ArgsOrig).
147 unprojToValence(ArgsProj, Projs, ValenceProj, [_:out|ValenceOrig],
148                 [_:_|ArgsOrig]) :-
149   unprojToValence(ArgsProj, Projs, ValenceProj, ValenceOrig,
150                   ArgsOrig).
151
152 % valences for elementary predicates.
153 valencePredicate(isNil, [nil:out]).
154 valencePredicate(isNil, [nil:in]).
155 valencePredicate(id, [a:in, b:in]).
156 valencePredicate(id, [a:out, b:in]).
157 valencePredicate(id, [a:in, b:out]).
158 valencePredicate(cons, [a:in, b:in, ab:out]).
159 valencePredicate(cons, [a:in, b:out, ab:in]).
160 valencePredicate(cons, [a:in, b:in, ab:in]).
161 valencePredicate(cons, [a:out, b:in, ab:in]).
162 valencePredicate(cons, [a:out, b:out, ab:in]).
163
164 % valences for numeric comparison mini-library
165 valencePredicate(gt, [a:in, b:in]).
166 valencePredicate(gte, [a:in, b:in]).
167
168 % valence combinations for the foldr operator.
169 %valenceFoldr(ValenceFoldr, ValenceP, ValenceQ)
170 valenceFoldr([a0:in, as:in, b:out],
171              [a:in, b:in, ab:out],
172              [a:in, b:out]).
173 valenceFoldr([a0:in, as:in, b:out],
174              [a:in, b:in, ab:out],
175              [a:out, b:out]).
176 valenceFoldr([a0:out, as:in, b:out],
177              [a:in, b:in, ab:out],
178              [a:in, b:out]).
179 valenceFoldr([a0:out, as:in, b:out],
```

```
180             [a:in, b:in, ab:out],
181             [a:out, b:out]).
182 valenceFoldr([a0:in, as:in, b:out],
183             [a:out, b:in, ab:out],
184             [a:out, b:out]).
185 valenceFoldr([a0:out, as:in, b:out],
186             [a:out, b:in, ab:out],
187             [a:out, b:out]).
188
189 % valence combinations for the natrec operator.
190 % valenceNatrec(NatrecValence, PValence, QValence)
191 valenceNatrec([a0:in, as:in, b:out],
192              [a:in, b:in, ab:out],
193              [a:in, b:out]).
194 % valence for the second variant with binary recursive case
195 % and unary base case.
196 valenceNatrec([a0:in, as:in, b:out], [b:in, ab:out], [b:out]).
197
198 % valence combinations for the foldr2 operator
199 % valenceFoldr2(Foldr2Valence, PValence, QValence)
200 valenceFoldr2([as:in, b:out], [a:in, b:in, ab:out], [b:in] ).
201 valenceFoldr2([as:in, b:out], [a:in, b:in, ab:out], [b:out]).
202 valenceFoldr2([as:in, b:out], [a:out, b:in, ab:out], [b:in] ).
203 valenceFoldr2([as:in, b:out], [a:out, b:in, ab:out], [b:out]).
204
205 % valence combinations for the ande operator are generated
206 % by the following predicate.
207 % % valenceAnde(ValenceAnde, ValenceP, ValenceQ).
208 valenceAnde([], [], []).
209 % only p
210 valenceAnde([N:IO|ValenceAnde], [N:IO|ValenceP], ValenceQ) :-
211   valenceAnde(ValenceAnde, ValenceP, ValenceQ).
212 % p and q
213 valenceAnde([N:IO|ValAnde], [N:IOp|ValP], [N:IOq|ValQ]) :-
214   valenceCompArg(IO, IOp, IOq),
215   valenceAnde(ValAnde, ValP, ValQ).
216 % only q
217 valenceAnde([N:IO|ValenceAnde], ValenceP, [N:IO|ValenceQ]) :-
218   valenceAnde(ValenceAnde, ValenceP, ValenceQ).
219
220 % supporting predicate for valenceAnde
221 valenceCompArg(in, in, in).
222 valenceCompArg(out, in, out).
223 valenceCompArg(out, out, in).
224 valenceCompArg(out, out, out).
225
226 % valence combinations for the proj operator are generated
```

```
% by the following predicate
valenceProj(ProjVals, Projs, UnboundMax, SourceVals) :-
  ((valenceHasOuts(ProjVals), AllowedModes=[in, out]);
   (\+valenceHasOuts(ProjVals), AllowedModes=[in])),
   valenceProj_(ProjVals, Projs, UnboundMax, AllowedModes,
               SourceVals).

valenceProj_([], [], _, _, []).
valenceProj_(ProjVals, Projs, N, AllowedModes,
            [_:out|SourceVals]) :-
  N >= 1,
  Nm is N-1,
  member(out, AllowedModes),
  valenceProj_(ProjVals, Projs, Nm, AllowedModes, SourceVals).
valenceProj_([B:IO|ProjVals], [A->B|Projs], UnboundMax,
            AllowedModes, SourceVals) :-
  valenceProj_(ProjVals, Projs, UnboundMax, AllowedModes,
              SourceValsRest),
  select(A:IO, SourceVals, SourceValsRest).

% supporting predicate for valenceProj
valenceHasOuts([_:out|_]).
valenceHasOuts([_:in|Vals]) :-
  valenceHasOuts(Vals).

% true if all argument names in a given atu are unique.
argNamesDifferent([]).
argNamesDifferent([A:_|Arest]) :-
  nameIsDiffToAllArgNames(A, Arest),
  argNamesDifferent(Arest).

% true if given name is different from the names in the
% second argument.
nameIsDiffToAllArgNames(_, []).
nameIsDiffToAllArgNames(A, [B:_|Arest]) :-
  dif(A, B),
  nameIsDiffToAllArgNames(A, Arest).
```

Appendix E.
Insertion sort code in Prolog

```
% Runs on: SWI−Prolog (MacOs 10.12, 64 bits, Version 7.3.20)
% Runs on: SICStus Prolog 4.2.0 (MacOs 10.12, x86_64−darwin−10.6.0)

gte(A, B) :- A>=B.
gt(A, B) :- A>B.
isNil([]).
id(A, A).
cons(A, B, [A|B]).

ltPass(A, Threshold, B) :-
  gte(A, Threshold),
  isNil(B).
ltPass(A, Threshold, B) :-
  gt(Threshold, A),
  id(A, B).

gtePass(A, Threshold, B) :-
  gt(Threshold, A),
  isNil(B).
gtePass(A, Threshold, B) :-
  gte(A, Threshold),
  id(A, B).

filterLt([], _, []).
filterLt([A|At], Param, Bs) :-
  ltPass(A, Param, []),
  filterLt(At, Param, Bs).
filterLt([A|At], Param, Bs) :-
  ltPass(A, Param, B), B\=[],
  filterLt(At, Param, Bt), Bs=[B|Bt].

filterGte([], _, []).
filterGte([A|At], Param, Bs) :-
  gtePass(A, Param, []),
  filterGte(At, Param, Bs).
filterGte([A|At], Param, Bs) :-
  gtePass(A, Param, B), B\=[],
  filterGte(At, Param, Bt), Bs=[B|Bt].
```

```
39
40 append(Xs, Ys, Zs) :-
41   isNil(Xs),
42   id(Ys, Zs).
43 append(Xs, Ys, Zs) :-
44   cons(X, Xt, Xs),
45   append(Xt, Ys, XYs),
46   cons(X, XYs, Zs).
47
48 insertOrd(As, Elem, Bs) :-
49   filterLt(As, Elem, ListLt),
50   filterGte(As, Elem, ListGte),
51   cons(Elem, ListGte, NewListGte),
52   append(ListLt, NewListGte, Bs).
53
54 iSort([], []).
55 iSort([E|Lt], Sorted) :-
56   iSort(Lt, SortedT),
57   insertOrd(SortedT, E, Sorted).
```