Compositional Relational Programming With Name Projection and Compositional Synthesis

Görkem Paçacı, Steve McKeever, Andreas Hamfelt

Department of Informatics and Media, Uppsala University

Abstract. CombInduce is a methodology for inductive synthesis of logic programs, which employs a reversible meta-interpreter for synthesis, and uses a compositional relational target language for efficient synthesis of recursive predicates. The target language, Combilog, has reduced usability due to the lack of variables, a feature enforced by the principle of compositionality, which is at the core of the synthesis process. We present a revision of Combilog, namely, Combilog with Name Projection (CNP), which brings improved usability by using argument names, whilst still staying devoid of variables, preserving the compositionality.

1 Introduction

Automated program synthesis is the task of generating programs that follow given specifications. There are various forms of synthesis, such as *deductive synthesis*, where the specifications are expressed in a formalized language, and *inductive synthesis*, where the specifications take the form of program input/output examples [1]. These two distinct approaches have competing qualities. For example, deductive synthesis guarantees the generated program will follow the specification, which means as long as the specification is correct, the generated program will also be correct. As a result, the usability of the specification language is of crucial value. It is also an issue that the specification language has to be as expressive as the target language, and often the specification itself is as long as the program to be generated [12]. In inductive synthesis, the input/output data examples are usually provided in a very simple form, so they require almost no formality. This makes inductive synthesis more versatile in this respect, but also the correctness of the generated program will rely on the completeness of the examples given. At the end it is up to the user (or programmer) to review, and confirm/deny the generated program. This brings the usability of the target language to focus, as it becomes a determining factor of the methods success.

Here we focus on the linguistic usability of CombInduce, a method for inductive synthesis of logic programs [11, 10]. It employs a reversible meta-interpreter, which is a technique developed through the 1990s [14, 9], and recently revisited by multiple works [13, 4]. The CombInduce approach is distinguished by its capability to synthesize two nested recursions at once, classified as a fold-2 program. Fold-2 programs include a fold operator as a recursive case of another fold operator, such as naive reversal of a list, or multiplication in Peano arithmetic.

CombInduce can synthesise these using only the standard elementary predicates providing the identity and list construction, and the general list recursion operators fold-left and fold-right. This is in contrast to most recent publications on inductive synthesis, which seem to focus on only one level of recursion [13, 4, 7]. Moreover, CombInduce generates mentioned fold-2 programs in under a second with mostly less than a handful examples, compared to some other methods that require tens of examples and are distinctly slower. On the other hand, some capabilities of CombInduce are still to be investigated, such as its strength in synthesizing complex non-recursive programs, and programs with negation.

The distinguishing recursive synthesis capability of CombInduce comes at a cost. The target language, namely Combilog, is required to be compositional[1], as well as being equivalent to definite clause programs in expressiveness. This means that the meaning of any operator of the language should be defined only in terms of meanings of its operands, isolating the meaning of an expression from the context it appears in. This leaves the language devoid of variables, decreasing its overall usability as a human-facing aspect of the synthesis. There are only a few notations that follow this principle. One of these is Quine's *Predicate-Functor Logic*[18], another is *Combinatory Logic* [19, 5, 6], but neither of these are intended to provide a reasonable level of usability or specialized recursion operators. As noted earlier, readability of the target language itself is crucial for the methods success, as the programmer needs to be able to comprehend and confirm the generated code as correct. If the notation is easy to modify, then when the synthesizer creates a close but incorrect candidate, the user can perform manual alterations.

In our earlier study, we identified the usability issues with Combilog to be related to the lack of variables as discussed above [17, 16]. We devised a visual language, *Visual Combilog*, which can mirror the textual Combilog code, and developed an editor that can view and edit Combilog code and Visual Combilog code side-by-side in real time, transforming back and forth as necessary. As a result of a user study involving 20 participants, we measured Visual Combilog to be significantly more usable compared to Combilog. We measured a 46% increase in speed and a 69% decrease in errors when the users were dealing with problems specifically devised to focus on argument binding [16].

As a continuation of our work on improving the linguistic usability of CombInduce, we present here a textual iteration of the Combilog language, namely, *Combilog with Name Projection* (CNP). In Section 2, we will reveal the specific problem that requires Combilog to be revised, and discuss why CNP is an improvement. We will continue by the formal semantics of CNP in Section 3. Section 4 will support our claims of usability with the results of a usability study, and Section 5 demonstrate how CNP is used for synthesis in place of Combilog.

[1] The concept of compositionality here refers to the *principle of compositionality* [21], where the meaning of an expression is defined as a function of meanings of its components only, and not to the concept of or-compositionality [3].

2 Compositional relational argument binding problem

In order to provide the mechanics for binding arguments of component predicates, Combilog devises the *make* operator which takes a source predicate and produces a new predicate where the arguments are bound to those in the source according to a given list of indices. To demonstrate, let us look at three uses of the *make* operator, in relation to how it's ordinarily achieved using variables. The first example reflects *cropping*, which eliminates arguments from a predicate:

$$\text{using variables: } p(X,Y) \leftarrow r(X,Y,\_) \qquad \text{using } make\text{: } p \leftarrow make([1,2],r)$$

Because the index list of the *make* operator refers only to the arguments 1 and 2, the predicate p has only two arguments bound to the respective arguments from r. The second example displays the case where the arguments in the new predicate do not appear in the same order as the source predicate. This use case of the *make* operator is referred to as *permutation* of arguments:

$$\text{using variables: } p(Y,X) \leftarrow r(X,Y,\_) \qquad \text{using } make\text{: } p \leftarrow make([2,1],r)$$

This is almost identical to the one in the first example, except the arguments appear in the switched order. The third argument of r is still cropped. The third and final example reflects the *expansion* use of the *make* operator. This is used to introduce new arguments that are unbound to the source predicate.

$$\text{using variables: } p(Y,X,\_) \leftarrow r(X,Y,\_) \qquad \text{using } make\text{: } p \leftarrow make([2,1,4],r)$$

The only difference in this example is the introduction of a third index 4, which yields a third argument in p which is not bound to an argument of r, since r does not have a 4th argument. For introducing new unbound arguments further higher indices can be used. A complete predicate definition shows the difficulty of comprehension and modification resulting from the use of *make* more clearly. Consider this implementation of the *append* predicate in Prolog:

$$append([], Ys, Ys).$$
$$append([X|Xsrest], Ys, [X|MidList]) \leftarrow append(Xsrest, Ys, MidList).$$

In order to be able to compare the code above to its Combilog equivalent, it is useful to first observe a version of it where the syntactic sugar specific to Prolog is removed. This version of the code is given below, where the list operator $[X|T]$ and the empty list constant [] are replaced by auxiliary predicates $cons(H,T,[H|T])$, and $const_{[]}$:

$$\begin{aligned} append(Xs, Ys, Zs) &\leftarrow const_{[]}(Xs), id(Ys, Zs). \\ append(Xs, Ys, Zs) &\leftarrow cons(X, Xsrest, Xs) \wedge \\ &\qquad append(Xsrest, Ys, MidList) \wedge \\ &\qquad cons(X, MidList, Zs). \end{aligned}$$

in Combilog the same predicate is written as:

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

The Combilog definition is significantly more difficult to comprehend and modify than the Prolog equivalent. This is the main issue addressed in this paper. The intention is to introduce a new syntax for Combilog without reducing expressiveness or breaking the compositionality principle. The principle can be expressed as follows. The meaning of every valid expression in a language: (1) Should be defined as a function of meanings of its components. (2) Should not depend on the context it appears. (3) Should not depend on what comes before or after it sequentially (excluding the name-called components of the expression that may happen to come before or after it). The principle is formally stated as:

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

Free variables may incorporate the context into the meaning of an expression, therefore the compositionality principle prohibits a general use of variables. Combilog programs contain no variables, and the constructs of the language are devised in this manner. The elementary components of Combilog programs are a limited number of predicates:

- *true* for logical truth
- $const_C$ for introducing a constant C, defined as $const_C(C)$.
- *id* for identity, defined as $id(X, X)$
- *cons* for working with lists, defined as $cons(H, T, [H|T])$.

Combilog programs are composed using the following operators.

- Logic operators *and* and *or*, which are correspond to set intersection and union, requiring their components to have equal number of arguments:

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

- The generalized projection operator, *make*:

$$make([\mu_1, \ldots, \mu_m], P)(X_{\mu_1}, \ldots, X_{\mu_m}) \leftarrow P(X_1, \ldots, X_n)$$

- The list recursion operators *foldr* and *foldl*:

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

Let observe how these operators behave through some examples:

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

In the Combilog code above, the predicate *isFather* is defined as the conjunction of *isMale* and *hasChildiren* predicates, all three predicates being unary. In contrast, in a language such as Prolog, using variables, the same could have been written as follows.

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

When arguments are bound in a non-trivial scheme, the *make* operator is required. Let us define the *daughterOf* predicate, which succeeds if *A is the daughter of B*, A being the first argument and B the second:

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

In this example, assuming the *isFemale* is unary, but *parentOf* is binary, it is necessary to expand the *isFemale* predicate to binary by adding a second, unbound argument. The index 2 in the *make* operation associated with *isFemale* does exactly this. Since there is no second argument in the original *isFemale* predicate, *make* defines a second argument but binds it to no argument of *isFemale*, introducing an unbound argument. The same expression could have been written using variables as:

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

As a final example of a Combilog program, let us demonstrate the recursion. The *append* example from earlier can be written using the *foldr* operator as:

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

Even though the definition of *append* with the *foldr* operator is quite simpler than without, the code involving a *make* operator renders any Combilog code relatively difficult to read and modify. In the next section, we present CNP which introduces argument names to overcome this usability problem.

3 Combilog with Name Projection (CNP)

CNP introduces the following changes to Combilog's syntax, which foremost includes the introduction of names for arguments. This allows a direct benefit since names also stand as a form of documentation as opposed to being only an identifier. There are also indirect benefits of using argument names, such as the possibility of defining operators that use argument names as a hint for schema matching between their operands' arguments. Here we summarize the overall changes CNP involves:

1. Numeric sequential argument positions (1st, 2nd, etc.) are replaced with nominal argument positions (*head*, *c*, etc.).
2. Elementary predicates are modified to include argument names.
3. The logic operators *and* and *or* are replaced with their name-aware variants which exhibit an auto-expanding behaviour, using arguments names of the components as clue.
4. The expansion use of the *make* operator is taken over by new auto-expanding logic operators (*ande* and *ore*) which accept components with any arity. This is made possible due to nominal argument positions, also.
5. A new *proj* operator is introduced, replacing the *make* operator. This new operator takes over the cropping, as well as introducing a *renaming* use.
6. The list recursion operators *foldl* and *foldr* are replaced with name-aware versions which introduce restrictions on argument names of the operands as well as fixed argument names for the resulting predicates.

These improvements are guided by a heuristic usability analysis using Green's *Cognitive Dimensions* [8], and a thorough discussion can be found in the relevant work of Paçacı [15]. They improve the usability significantly while preserving expressiveness and the compositionality principle. The resulting approach significantly resembles Codd's *Relational Algebra* [2], especially the *unordered relational domains* with the non-sequential nominal argument positions and *natural join* with the auto-expanding logic operators. The intention here is to perform logic programming rather than modelling and querying relational data. Let us observe the following examples of CNP syntax. In the examples, the argument names of a component predicate are given in a signature form $predName : \{name1, name2, \ldots\}$. Let us start with the *daughterOf* predicate:

$$parentOf : \{parent, child\}$$
$$isFemale : \{name\}$$
$$daughterOf \leftarrow ande\big(proj(parentOf, \{parent \mapsto parent, child \mapsto daughter\}),$$
$$proj(isFemale, \{name \mapsto daughter\})\big)$$

The two *proj* operators project the *parentOf* and *isFemale* predicates to produce anonymous predicates with argument names $\{parent, daughter\}$ and $\{daughter\}$, respectively. Then, the *ande* operator takes their conjunction, mapping identically named arguments, producing another anonymous predicate with arguments $\{parent, daughter\}$, which is finally assigned to the predicate name *daughterOf*. The second example is the recursive definition of the *append* predicate:

$$append \leftarrow foldr(cons, id)$$

This example is identical to that in Combilog, since it does deal with arguments. As it will be discussed later, the *foldr* operator in CNP has fixed argument names, therefore the anonymous predicate above has the argument names inherited from this operator: $\{as, a0, b\}$. In order to change these to a more conventional argument names for *append*, we can project it, and assign it to a predicate name:

$$append \leftarrow proj(foldr(cons, id), \{as \rightarrow xs, a0 \rightarrow ys, b \rightarrow zs\})$$

which has the argument names $\{xs, ys, zs\}$.

Analogous to Combilog programs, CNP programs consist of a set of predicate definitions in the form of $p \leftarrow \varphi$, where p is a predicate symbol and φ is a body. The body is a CNP expression, constructed from elementary predicates and composition operators (*proj*, *ande*, *ore*, *foldr*, *foldl*). In the following sections, we will we will present the formal semantics of these CNP program constructs.

3.1 Name-aware tuples and extensions

Before moving on to specific operators and their denotations, let us clarify a fundamental concept. In order to support the mechanics of argument binding with names, we shall adopt a special understanding of a relational tuple, namely, α-tuple, which is in line with a *record*. An ordinary tuple with k elements can be formalized as a function from a k-size subset of the natural numbers K to elements from the Herbrand Universe, $\tau : K \to H$. For example, for a given tuple $\tau = \langle t_1, \ldots, t_k \rangle$, applications of τ as a function are $\tau(1) = t_1, \ldots, \tau(k) = t_k$. Assuming the existence of a bijective name map α from a set of names A to the same subset of the natural numbers K, that is, $\alpha : A \to K$, then α is a compatible name map to transform an ordinary tuple τ to an α-tuple $\tau_\alpha = \tau \circ \alpha$. As a result, τ_α is obtained as a function from a set of names A to elements from the Herbrand universe. Given a usual tuple $\tau = \langle t_1, \ldots, t_k \rangle$ and a compatible name map $\alpha = \{a_1 \mapsto 1, \ldots, a_k \mapsto k\}$, the applications of τ_α as a function are $\tau_\alpha(a_1) = t_1, \ldots, \tau_\alpha(a_k) = t_k$. Since the name map α is bijective, it can also be used to obtain an ordinary tuple from an α-tuple, establishing the isomorphism between them. Consequently, the relational extensions discussed in the following sections should be read as sets of α-tuples, rather than sets of ordinary tuples, referred to as α-extensions.

In the following sections, we will give denotations of elementary predicates and operators of CNP, and we will finalize the semantics with the fixpoint semantics of CNP programs.

3.2 Elementary predicates

CNP includes a set of elementary predicates as counterparts to Combilog's elementary predicates, the only difference being that they denote α-extensions. The denotations of these elementary predicates are given below, with their associated

name maps given as subscripts to the predicate symbols.

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

where

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

Note that the Herbrand Universe is understood as that of a corresponding definite clause program. The *const* operator is parametric, where N is the name of the single argument of the predicate, and C is the constant the predicate should succeed for. CNP also defines an explicit instance of the *const* operator, namely *isNil*, for providing easy access to the empty list constant, defined as:

$$isNil = const(nil, [])$$

3.3 Projection operator

CNP replaces Combilog's *make* operator with the *proj* operator. The projection operator $proj(S_\alpha, P)$ produces a new predicate based on a given source predicate S_α, considering the projection map given as P. A valid projection map P is a map from a non-empty set of existing names $A \subseteq Dom(\alpha)$ from the name map of S to a set of new names B, formalized as a function as $P : A \to B$. P is not required to cover all names appearing in the name map of S, but has to have at least one mapping, and is required to be bijective (every old name $a \in A$ is mapped to exactly one new name, and every new name $b \in B$ is mapped by exactly one old name). The map entries where a name is mapped to itself by P are only *projection*, while the map entries that map names to new names are called *renaming*.

$$
\begin{aligned}
[\![proj(S_\alpha, P)_{\alpha_c}]\!] &= \{\tau_\alpha \circ P^{-1} \mid \tau_{\alpha_1} \in [\![S_\alpha]\!]\} \\
&\text{where } \alpha_c = \alpha \circ P^{-1}
\end{aligned}
$$

The resulting name map associated with application of *proj* is the composition of the name map of S and P^{-1}, containing only those names that are projected by P. The *proj* operator takes over most of the functionality of *make* except introduction of argument names, which is taken over by the logic operators, discussed in the next section.

3.4 Logic operators

The logic operators defined in CNP have the option to vary their behaviour by using argument names as hints to establish bindings between arguments of the component predicates. The auto-expanding logic operators *ande* and *ore* are introduced, which behave as if the component predicates are expanded with unbound arguments to cover all argument names of all the component predicates, taking their union. This helps to reduce the boilerplate code that emerges as a result of compulsory use of expanding the *make* operator to each operand of a logic operator, as discussed earlier. This functionally takes over the expanding use of the *make* operator in Combilog. Denotations of the auto-expanding logic operators are given below, where $Dom(\alpha)$ refers to the domain of a name map, which consists of the relevant argument names.

$$
\begin{aligned}
[\![ande(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c}]\!] &= \{\tau_{\alpha_c} \in H_{\alpha_c} \mid (\tau_{\alpha_c} \supseteq \tau_{\alpha_1} \wedge \tau_{\alpha_c} \supseteq \tau_{\alpha_2}) \wedge \\
&\qquad \tau_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge \tau_{\alpha_2} \in [\![S_{\alpha_2}]\!]\} \\
[\![ore(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c}]\!] &= \{\tau_{\alpha_c} \in H_{\alpha_c} \mid (\tau_{\alpha_c} \supseteq \tau_{\alpha_1} \vee \tau_{\alpha_c} \supseteq \tau_{\alpha_2}) \wedge \\
&\qquad \tau_{\alpha_1} \in [\![R_{\alpha_1}]\!] \wedge \tau_{\alpha_2} \in [\![S_{\alpha_2}]\!]\} \\
\text{where} \quad As &= Dom(\alpha_1) \cup Dom(\alpha_2) \\
H_{As} &= \{\{A\} \times H \mid A \in As\} \\
H_{\alpha_c} &= \{\{T_1, \ldots, T_n\} \mid T_1 \in H_{A_1} \wedge \ldots \wedge T_n \in H_{A_n} \wedge \\
&\qquad \{H_{A_1}, \ldots, H_{A_n}\} = H_{As}\}
\end{aligned}
$$

The set of argument names A in the expanded composition is established as the union of all names appearing in domains of component predicates' name maps. The set H_A is a set of sets, where each set contains a mapping for one of the names in A to every element of the Herbrand Universe. Each of these sets is referred as T_i in the definition of H_{α_c}. Every name-value pair t has a name from A as its first element, and an element of the Herbrand Universe as its second element. In this way, H_{α_c} is established as a set of α-tuples compatible with α_c, which is the name map associated with the composition. The resulting name map α_c maps the union of argument names in α_1 and α_2 to numeric indices, where names in α_1 are mapped to their original indices and the unique names in α_2 are mapped to the following indices, preserving their original order. For example, given $\alpha_1 = \{a \mapsto 1, b \mapsto 2\}$ and $\alpha_2 = \{b \mapsto 1, c \mapsto 2\}$, the resulting name map of the composition is $\alpha_c = \{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$.

The availability of argument names enables the implementation of a wider range of logic operators that apply various argument binding schemes. Besides *ande* and *ore*, operators such as *ando* and *oro* which bind every name-matching argument but returns a predicate with only the unmatched argument names can be implemented, resembling relation composition [20] but applying to multi-ary predicates as well as binary. Another alternative is the pair andl/orl, which return a predicate with only the arguments from the left-hand component. Extending these binary logic operators to multiary variants is straightforward.

3.5 Recursion operators

CNP defines list recursion operators *foldr* and *foldl*. These are the counterparts to their namesakes in Combilog, and operate the same way, modulo the addition of names for their arguments. In their denotations below, the aggregate definitions *foldr* and *foldl* refer to the auxiliary definitions $foldr_0/foldr_{i+1}$ and $foldl_i/foldl_{i+1}$, respectively, where $i \geq 0$.

$$
\begin{aligned}
[\![foldr(P,Q)]\!] &= \bigcup_{i=0}^{\infty} [\![foldr_i(P,Q)]\!] \\
[\![foldr_0(P,Q)]\!] &= \big\{\{a0 \mapsto Y, as \mapsto [], b \mapsto Z\} \in H_{\alpha_f} \mid \{a \mapsto Y, b \mapsto Z\} \in [\![Q]\!]\big\} \\
[\![foldr_{i+1}(P,Q)]\!] &= \big\{\{a0 \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W\} \in H_{\alpha_f} \mid \\
&\qquad \big(\exists Z \in H \ s.t. \\
&\qquad\quad \{a0 \mapsto Y, as \mapsto Xs, b \mapsto Z\} \in [\![foldr_i(P,Q)]\!] \wedge \\
&\qquad\quad \{a \mapsto X, b \mapsto Z, ab \mapsto W\} \in [\![P]\!]\big)\big\} \\
[\![foldl(P,Q)]\!] &= \bigcup_{i=0}^{\infty} [\![foldl_i(P,Q)]\!] \\
[\![foldl_0(P,Q)]\!] &= \big\{\{a0 \mapsto Y, as \mapsto [], b \mapsto Z\} \in H_{\alpha_f} \mid \{a \mapsto Y, b \mapsto Z\} \in [\![Q]\!]\big\} \\
[\![foldl_{i+1}(P,Q)]\!] &= \big\{\{a0 \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W\} \in H_{\alpha_f} \mid \\
&\qquad \big(\exists Z \in H \ s.t. \\
&\qquad\quad \{a \mapsto X, b \mapsto Y, ab \mapsto Z\} \in [\![P]\!] \wedge \\
&\qquad\quad \{a0 \mapsto Z, as \mapsto Xs, b \mapsto W\} \in [\![foldl_i(P,Q)]\!]\big)\big\} \\
\textit{where } H_{\alpha_f} &= \big\{\{a0 \mapsto A_0, as \mapsto As, b \mapsto B\} \mid \langle A_0, As, B\rangle \in H^3\big\}
\end{aligned}
$$

The name maps for the fold operations are fixed, as well as their operand expressions P and Q. Operand expressions must 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, due to the lack of argument indices. With their fixed names, argument *a0* refers to the initial value, argument *as* refers to the list, and *b* refers to the result of the folding. The fixed name maps are as follows: $\alpha_{foldr} = \{a0 \mapsto 1, as \mapsto 2, b \mapsto 3\}$, $\alpha_P = \{a \mapsto 1, b \mapsto 2, ab \mapsto 3\}$, $\alpha_Q = \{a \mapsto 1, b \mapsto 2\}$. There are also two binary variants, omitting the argument *a0*, instead obtaining the initial value through the base case. These are defined in terms of the generic *fold* operators, as in the definition of *foldr2*:

$$foldr2(P,Q) = proj(foldr(P, ande(Q, id)), \{as \mapsto as, b \mapsto b\})$$

3.6 Fixpoint semantics

The model-theoretical meaning $M_{\models}(P_{cnp})$ of a CNP program P_{cnp} is expressed in a similar way to that of Combilog as the least fixed point of the power func-

tion of an immediate consequence operator T^{cnp}. Similar to Combilog, the domain of T^{cnp} is an extension map instead of the usual Herbrand interpretations. Extension maps are structure that map predicates names to their extensions. For a CNP program P_{cnp} with m predicate definitions, the extension map is formalized as: $E^{\alpha} = \bigcup_{i=1}^{m} \{p_i \mapsto e_i^{\alpha}\}$ Similarly, for a CNP program P_{cnp}, the immediate consequence operator $T_{P_{cnp}}^{cnp}$ is defined using extension maps is $T_{P_{cnp}}^{cnp}(E^{\alpha}) = \bigcup_{i}^{m} \{p_i \mapsto [\![\varphi_i]\!]_{E^{\alpha}}\}$, where p_i refers to the ith predicate symbol, φ_i to the ith predicate body, and $[\![\varphi]\!]_{E^{\alpha}}$ denotes the α-extension of the body φ with regard to extension map E^{α} and the denotations of elementary predicates. The definition of a power function of the $T_{P_{cnp}}^{cnp}$ is given as:

$$
\begin{aligned}
T_{P_{cnp}}^{cnp} \uparrow 0 &= \bigcup_{j=1}^{m} \{p_j \mapsto \emptyset\} \\
T_{P_{cnp}}^{cnp} \uparrow (i+1) &= T_{P_{cnp}}^{cnp}(T_{P_{cnp}}^{cnp} \uparrow i) \\
T_{P_{cnp}}^{cnp} \uparrow \omega &= \bigcup_{j=1}^{m} \{p_j \mapsto \bigcup_{i=0}^{\infty} \big((T_{P_{cnp}}^{cnp} \uparrow i)(p_j)\big)\}
\end{aligned}
$$

and the model-theoretical meaning of a CNP program P_{cnp} is calculated as the least fixed point of the power function: $M_{\models}(P_{cnp}) = T_{P_{cnp}}^{cnp} \uparrow \omega$.

Using meaning-preserving reversible transformation steps, Combilog and CNP programs can be converted to each other, and through these transformation steps it can be proven that the least fixed point of any two program transformed would be model-theoretically isomorphic, modulo introduction/removal of names. This proof is omitted here for brevity, but it can be found in authors' other work [15]. Because the meaning of Combilog programs are proven to be equivalent to a corresponding definite clause program, the meaning of a CNP program and a corresponding definite clause program would also be isomorphic by transitivity.

4 Usability of CNP programs

It is self-evident from observing the examples of CNP code in the previous section that CNP code is more readable compared to the original Combilog code. The question remains that how does it compare to notations with variables most common in the Logic Programming paradigm? In order to answer this, a within-subjects usability test has been conducted. The study compared two notations, a *Notation X* which is based on Prolog-like syntax with variables but included none of the syntactic sugar, such as list construction ([|]) or pattern matching; and a *Notation Y*, which is based on CNP. Elimination of syntactic sugar was deemed necessary to focus the study on the argument binding problem. Counterbalancing was performed by ordering the notations differently for two groups consisting of 10 participants each. Participants consisted of programmers working either in the industry or at a university, equally weighted, and distributed among the two groups. Through a pre-questionnaire, participants who had experience using Prolog were disqualified in order to have balanced results. The

study was conducted through a text-based on-line questionnaire. It included six questions, out of which three involved comprehensibility and three modifiability. Each question was based on between one and four short predicate definitions. The code segments were based on working code, relating to well-known textbook examples, but they were obfuscated differently for each notation to reduce the learning effect, and included increasing levels of complexity in terms of number of operations (variable binding or logic operators). Let us observe the first modification question, which includes the following fragments of code, given here before obfuscation of predicate names. In notation X (Prolog-like):

```
flightRoute(A, B).
trainRoute(A, B).
outInRouteOpt(D, E) :- flightRoute(D, _), trainRoute(D, _),
                       flightRoute(_, E), trainRoute(_, E).
```

and in notation Y (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}))
```

The question required the participants to identify and refactor out a reusable component from the conjunction of `flightRoute` and `trainRoute` predicates, thereby creating a new `route` predicate. In plain text the *proj* operator is implicit, where `flightRoute {x->a}` is equivalent to $proj(\mathit{flightRoute}, \{x \mapsto a\})$.

The participants took 276s (seconds, on average) to answer the first three comprehension questions for Notation X, while they took 345s for Notation Y, which corresponds to a 25% longer task time while using Notation Y ($P < 0.05$). For the following three modification questions, the results were in favour of Notation Y. While the participants took 468s to finish modification questions in Notation X, they took 366s for Notation Y, which corresponds to 22% shorter task time while using Notation Y ($P < 0.05$). The correctness of participants' answers were also measured. For comprehension questions there were no noticeable differences. For modification questions, the participants gave 42% more correct answers while they were using Notation Y ($P < 0.01$). This difference is mostly due to a single refactoring question which required alpha-renaming in Notation X but was a simple replacement in Notation Y. In a post-test questionnaire, the participants were asked three questions about their preferences. In answer to the question "Which notation did you find easier to read", 12 participants replied notation X, and 8 replied notation Y. The second question was "Which notation did you find easier to modify?", to which 8 participants replied X, 11 replied Y, and 1 replied no preference. The third question was "Which notation would you choose, if you had to use one for a project?", to which 8 participants replied X, 10 replied Y, and 2 replied no preference.

5 Compositional synthesis

The synthesis approach in CombInduce can be described as a top-down search procedure that attempts to place language operators and elementary predicates in an expression tree, written as a reversed meta-interpreter in Prolog [9, 11]. The synthesizer incorporates well-modedness constraints to make sure the generated programs are procedurally terminating, and operators are utilized in valid combinations [10]. The CNP synthesizer is written the same way, in Prolog, as a single predicate *synInc* as the entry point [15]. This predicate gradually increases the depth of the search, to find shorter programs first, if there are any, while also changing the default depth-first search strategy of Prolog for a breadth-first search. To demonstrate the technique through examples, let us synthesize the naive reverse operation using the CNP synthesizer. This operation requires two levels of recursion, and an elementary predicate to construct a unit list, a list consisting of one element. Let us start by synthesizing this predicate first.

```
?- synInc(P, [a:in, aList:out], [[a:1, aList:[1]]]).
```

The call above asks for a predicate P with two arguments $\{a, aList\}$, and specifies a mode $\{a : in, aList : out\}$ for the program that the synthesizer should guarantee it will procedurally terminate when executed. The call also includes a single input/output example, specifying when the argument a is 1, the argument *aList* should be a unit list [1]. The first predicate suggested is the correct one:

```
P = proj(ande(cons, proj(isNil, [nil->b])), [a->a, ab->aList]).
```

The suggestion constructs a conjunction of *cons* and *isNil*, binding the tail of a list to the empty list, its head to an argument a, and the whole list to the argument *aList*. After the user inspects and confirms the code above, and assigns a predicate name *asList*, it is available as background knowledge to the synthesizer. The next step is to synthesize the recursion stage:

```
?- synInc(P, [as:in, bs:out], [[as:[1,2,3], bs:[3,2,1]]]).
```

The call requests a predicate P with two arguments $\{as, bs\}$, with a single input/output example. The first suggested implementation is the correct one:

```
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 suggested predicate uses two nested *fold* variants, traversing a given list *as* (the *foldr2* operation) where each element is appended to a running list (the *foldr* operation). The *foldr* operation is a variant of the ordinary *append* predicate visited earlier in Section 2 with the second argument being a single element instead of a list. After the user confirms the synthesized predicate as the correct implementation, and manually assigns it a predicate name (`reverse`), it can be tested through the meta-interpreter predicate named *cnp*:

```
?- cnp(reverse, [as:[a,b,c,d], bs:Bs]).
```

succeeds with the answer binding the `Bs` variable:

```
Bs = [d,c,b,a].
```

Even though the program was asked to terminate in one direction, it can be used to un-reverse a list:

```
?- cnp(reverse, [as:As, bs:[d,c,b,a]]).
```

which succeeds with the following answer, binding the `As` variable instead:

```
As = [a,b,c,d].
```

6 Conclusion

In the introduction we drew attention to the usability of the target language, as it is necessary that the user inspects, comprehends and confirms the synthesized programs. If necessary, the user may choose to add more example cases that are not covered to achieve more specific programs. We have demonstrated this through a synthesis application in Section 5 through the synthesis of a naive reverse predicate, where the user had to intervene to initiate and confirm two separate predicates. Besides being readable, if the notation is modifiable, the user may manually alter the program as well. Among the results of the user study presented in Section 4, it was shown that the argument binding using nominal projection compares well to use of variables when measured in isolation. It shall be made clear that we do not claim to have devised a language which is equally user-friendly as Prolog. The user study measured only specific aspects of the languages, narrowly focusing on argument binding comprehensibility and modifiability. Programming for general problem solving involves various other skills. It is important to consider the context of our work, CNP and Combilog are fundamentally different to languages such as Prolog or Mercury, due to compositionality. A fair comparison can only be made with the likes of Predicate-Functor Logic or Combinatory Logic, which are also variable-free, compositional systems of logic. For future work, we intend to improve the synthesis for even deeper levels of recursion, such as fold-3 programs, which would be able to synthesise programs such as *exponent* in Peano arithmetic.

References

1. 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.
2. Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

3. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Information and Computation*, 169(1):23 – 80, 2001.
4. A. Cropper, A. Tamaddoni-Nezhad, and S. H. Muggleton. *Meta-Interpretive Learning of Data Transformation Programs*, pages 46–59. Springer, 2016.
5. H. B. Curry. Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, 52(3):pp. 509–536, 1930.
6. Haskell B Curry and Robert Feys. Combinatory logic, volume i of studies in logic and the foundations of mathematics, 1958.
7. J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.*, 50(6):229–239, June 2015.
8. Thomas R.G. Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.
9. A. Hamfelt and J. F. Nilsson. Inductive metalogic programming. In *Proceedings Fourth International Workshop on Inductive Logic programming*, pages 85–96. Bad Honnef/Bonn GMD-Studien Nr. 237, 1994.
10. A. Hamfelt and J. F. 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.
11. A. Hamfelt and J. F. Nilsson. Inductive synthesis of logic programs by composition of combinatory program schemes. In P. Flener, editor, *Procs. Workshop on Logic Based Program Transformation and Synthesis (LNCS 1559)*, pages 143–158. Springer-Verlag, 1999.
12. E. Kneuss, I. Kuraj, V. Kuncak, and P. 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.
13. S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.
14. M. Numao and M. 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.
15. G. Paçacı. *Representation of Compositional Relational Programs*. PhD thesis, Uppsala University, Information Systems, 2017.
16. G. Paçacı and A. 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.
17. G. Paçacı and A. 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.
18. W. V. Quine. Predicate-functor logic. In E. Fenstad, editor, *Procs. Second Scandinavian Logic Symposium*, pages 309–315. North-Holland, 1971.
19. M. Schönfinkel. über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3-4):305–316, 1924.
20. Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(03):73–89, 1941.
21. J. van Benthem and A. Ter Meulen. *Handbook of logic and language*. Elsevier, 1996.