Generating Collection Transformations from Proofs

MICHAEL BENEDIKT, Oxford University, United Kingdom
PIERRE PRADIC, Oxford University, United Kingdom

Nested relations, built up from atomic types via product and set types, form a rich data model. Over the last decades the *nested relational calculus*, NRC, has emerged as a standard language for defining transformations on nested collections. NRC is a strongly-typed functional language which allows building up transformations using tupling and projections, a singleton-former, and a map operation that lifts transformations on tuples to transformations on sets.

In this work we describe an alternative declarative method of describing transformations in logic. A formula with distinguished inputs and outputs gives an *implicit definition* if one can prove that for each input there is only one output that satisfies it. Our main result shows that one can synthesize transformations from proofs that a formula provides an implicit definition, where the proof is in an intuitionistic calculus that captures a natural style of reasoning about nested collections. Our polynomial time synthesis procedure is based on an analog of Craig's interpolation lemma, starting with a provable containment between terms representing nested collections and generating an NRC expression that interpolates between them.

We further show that NRC expressions that implement an implicit definition can be found when there is a classical proof of functionality, not just when there is an intuitionistic one. That is, whenever a formula implicitly defines a transformation, there is an NRC expression that implements it.

CCS Concepts: • **Theory of computation** → *Proof theory*; **Logic and databases**.

Additional Key Words and Phrases: nested collections, synthesis, proofs

ACM Reference Format:
Michael Benedikt and Pierre Pradic. 2021. Generating Collection Transformations from Proofs. *Proc. ACM Program. Lang.* 5, POPL, Article 14 (January 2021), 28 pages. https://doi.org/10.1145/3434295

1 INTRODUCTION

Nested relations are a natural data model for hierarchical data. Nested relations are objects within a type system built up from basic types via tupling and a set-former. In the 1980's and 90's, a number of algebraic languages were proposed for defining transformations on nested collections. Eventually a standard language emerged, the *nested relational calculus* (NRC). The language is strongly-typed and functional, with transformations built up via tuple manipulation operations as well as operators for lifting transformations over a type T to transformations taking as input a set of objects of type T, such as singletons constructors and a mapping operator. One common formulation of these uses variables and a "comprehension" operator for forming new objects from old ones [Buneman et al. 1995], while an alternative algebraic formalism presents the language as a set of operators that can be freely composed. It was shown that each NRC expression can be evaluated in polynomial time in the size of a finite data input, and that when the input and output is "flat" (i.e. only one level of nesting), NRC expresses exactly the transformations in the standard relational database language

Authors' addresses: Michael Benedikt, Computer science department, Oxford University, United Kingdom; Pierre Pradic, Computer science department, Oxford University, United Kingdom.

2475-1421/2021/1-ART14
https://doi.org/10.1145/3434295

relational algebra. Wong's thesis [Wong 1994] summarizes the argument made by this line of work "NRC can be profitably regarded as the 'right' core for nested relational languages". NRC has been the basis for most work on transforming nested relations. It is the basis for a number of commercial tools [Melnik et al. 2010], including those embedding nested data transformations in programming languages [Meijer et al. 2006], in addition to having influence in the effective implementation of data transformations in functional programming languages [Gibbons 2016; Gibbons et al. 2018].

Although NRC can be applied to other collection types, such as bags and lists, we will focus here on just nested sets. We will show a new connection between NRC and first-order logic. There is a natural logic for describing properties of nested relations, the well-known Δ_0 formulas, built up from equalities using quantifications $\exists x \in \tau$ and $\forall y \in \tau$ where τ is a term. For example, formula $\forall x \in c\ \pi_1(x) \in \pi_2(x)$ might describe a property of a nested relation c that is a set of pairs, where the first component of a pair is of some type T and the second component is a set containing elements of type T. A Δ_0 formula $\Sigma(\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}, \mathrm{o}_{out})$ over variables $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ and variable o_{out} thus defines a relationship between $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ and o_{out}. For such a formula to define a transformation it must be *functional*: it must enforce that o_{out} is determined by the values of $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$. More generally, if we have a formula $\Sigma(\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}, \mathrm{o}_{out}, \vec{a})$, we say that Σ *implicitly defines* o_{out} *as a function of* $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ if:

(*) For each two bindings σ_1 and σ_2 of the variables $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}, \vec{a}, \mathrm{o}_{out}$ to nested relations satisfying Σ, if σ_1 and σ_2 agree on each o^i_{in}, then they agree on o_{out}.

That is, Σ entails that the value of o_{out} is a partial function of the value of $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$.

Note that when we say "for each binding of variables to nested relations" in the definitions above, we include infinite nested relations as well as finite ones. An alternative characterization of Σ being an implicit definition, which will be more relevant to us in the sequel, is that there is a proof that Σ defines a functional relationship. Note that (*) is a first-order entailment: $\Sigma(\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}, \mathrm{o}_{out}, \vec{a}) \wedge \Sigma(\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}, \mathrm{o}'_{out}, \vec{a}') \models \mathrm{o}_{out} = \mathrm{o}'_{out}$ where in the entailment we omit some first-order "sanity axioms" about tuples and sets. We refer to a proof of (*) for a given Σ and subset of the input variables $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$, as a *proof that* Σ *implicitly defines* o_{out} *as a function of* $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$, or simply a *proof of functionality* dropping Σ, o_{out}, and $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ when they are clear from context. By the completeness theorem of first-order logic, whenever Σ defines o_{out} as a function of $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ according to the semantic definition above, this is witnessed by a proof, in any of the standard complete proof calculi for classical first-order logic (e.g. tableaux, resolution). Such a proof will use the sanity axioms referred to above, which capture extensionality of sets, the compatibility of the membership relation with the type hierarchy, and properties of projections and tupling.

Example 1.1. We consider a specification in logic involving two nested collections, F and G. The collection F is of type $\mathrm{Set}(\mathcal{U} \times \mathcal{U})$, where $\mathcal{U}$ refers to the basic set of elements, the "Ur-elements" in the sequel. That is, F is a set of pairs. The collection G is of of type $\mathrm{Set}(\mathcal{U} \times \mathrm{Set}(\mathcal{U}))$, a set whose members are pairs, the first component an element and the second a set.

Our specification Σ will state that for each element g in G there is an element f_1 appearing as the first component of a pair in F, such that g represents f_1, in the sense that its first component is f_1 and its second component accumulates all elements paired with f_1 in F. This can be specified easily by a Δ_0 formula:

$$\begin{aligned} \forall g \in G\ \exists f \in F \quad & \pi_1(g) = \pi_1(f)\ \wedge\ \forall x \in \pi_2(g)\ \langle \pi_1(f), x \rangle \in F \\ & \wedge\ \forall f' \in F\ [\pi_1(f') = \pi_1(f) \rightarrow \pi_2(f') \in \pi_2(g)] \end{aligned}$$

Σ also states that for each element f_1 lying within a pair in F there is a corresponding element g of G that pairs f_1 with all of the elements linked with f in F.

$$\forall f \in F\ \exists g \in G \quad \pi_1(g) = \pi_1(f) \ \wedge\ \forall x \in \pi_2(g)\ \langle \pi_1(f), x\rangle \in F$$
$$\wedge\ \forall f' \in F\ \ [\pi_1(f') = \pi_1(f) \rightarrow\ \ \pi_2(f') \in \pi_2(g)]$$

We can prove from Σ that G is a function of F, and thus Σ implicitly defines a transformation from F to G. We give the argument informally here. Fixing F, G and F, G' satisfying Σ, we will prove that if $g \in G$ then $g \in G'$. The proof begins by using the conjunct in the first item to obtain an $f \in F$. We can then use the second item on G' to obtain a $g' \in G'$. We now need to prove that $g' = g$. Since g and g' are pairs, it suffices to show that their two projections are the same. We can easily see that $\pi_1(g) = \pi_1(f) = \pi_1(g')$, so it suffices to prove $\pi_2(g') = \pi_2(g)$. Here we will make use of extensionality, arguing for containments between $\pi_1(g')$ and $\pi_2(g)$ in both directions. In one direction we consider an $x \in \pi_2(g')$, and we need to show x is in $\pi_2(g)$. By the second conjunct in the second item we have $\langle \pi_1(f), x\rangle \in F$. Now using the first item we can argue that $x \in \pi_2(g)$. In the other direction we consider $x \in \pi_2(g)$, we can apply the first item to claim $\langle \pi_1(f), x\rangle \in F$ and then employ the second item to derive $x \in \pi_2(g')$.

Now let us consider G as the input and F as the output. We cannot say that Σ describes F as a *total* function of G, since Σ enforces constraints on G: that the second component of a pair in G cannot be empty, and that any two pairs in G that agree on the first component must agree on the second. But we can prove from Σ that F is a partial function of G: fixing F, G and F', G satisfying Σ, we can prove that $F = F'$. ◁

Our first main contribution is a polynomial time synthesis procedure that takes as input a proof that Σ implicitly defines o as a function of $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$, generating an NRC expression with input $\mathrm{o}^1_{in} \ldots \mathrm{o}^k_{in}$ that implements the transformation that Σ defines. We require a proof of functionality in a certain intuitionistic calculus. Although the calculus is not complete for classical entailment, we argue that it is quite rich and show that it is equivalent to certain prior intuitionistic calculi.

Example 1.2. Let us return to Example 1.1. From a proof in our calculus that Σ defines G as a function of F, our synthesis algorithm will produce an expression in NRC that generates G from F. This will be an expression that simply "groups on the first component".

From a proof from Σ that F is a function of G, our algorithm will generate an NRC expression that forms F by flattening G. ◁

We also show that this phenomenon applies when there is a classical proof of functionality, not just an intuitionistic one. That is, we show that whenever a formula Σ projectively implicitly defines a transformation $\mathcal{T}$, that transformation can be expressed in a slight variant of NRC. The result can be seen as an analog of the well-known Beth definability theorem for first-order logic [Beth 1953], stating that a property of a first-order structure is defined by a first-order open formula exactly when it is implicitly defined by a first-order sentence. In the process we prove an *interpolation theorem*, showing that whenever we have provable containments between nested relations, there is an NRC expression that sits between them. Overall our results show a close connection between logical specifications of transformations on nested collections and the functional transformation language NRC, a result which is not anticipated by the prior theory.

Organization. We overview related work in Section 2 and provide preliminaries in Section 3. Section 4 details our proof calculus and the algorithm that synthesizes definitions from proofs. We include an example (Figure 4) of how one would use it to prove functionality of an expression, and an illustration of how our synthesis algorithm would generate an NRC expression from the proof (Example 4.8). Section 5 concerns another logic-based specification that can be transformed into

NRC expressions, based on the notion of interpretations. Section 6 shows that even for classical proofs there is a corresponding NRC expression. This conversion goes through the interpretation representation introduced in Section 6. We show a general result that implicit definitions in multi-sorted logic can be converted to interpretations, and then use the results of Section 6 to argue that these interpretations can be converted to NRC expressions.

We close with conclusions in Section 7. In the body of the paper we focus on explaining the results and some proof ideas, with most proof details deferred to the supplementary materials.

2 RELATED WORK

In the context of transformations of ordinary "flat" relations, Segoufin and Vianu [Segoufin and Vianu 2005] showed that transformations definable in relational algebra are the same as those that satisfy a variant of implicit definability ("determinacy"). The result of [Segoufin and Vianu 2005] makes use of a refinement of Craig's interpolation theorem due to Otto [Otto 2000]. The use of interpolation theorems in moving from implicit to explicit is well-established, dating back to Craig's proof of the Beth definability theorem [Craig 1957]. Segoufin and Vianu's result is motivated by the ability to evaluate transformations defined over one set of "base predicates" using another set of "view predicates", where the views are defined implicitly by a background theory relating them to the base predicate. The idea that one can use interpolation algorithms to synthesize transformations from implicit specifications first appears in the work of Toman and Weddell [Toman and Weddell 2011] and has been developed in a number of directions subsequently [Benedikt et al. 2016]. In the absence of nesting of sets, the relationship between formulas and terms of an algebra is much more straightforward; relational algebra defines exactly those transformations whose output is a comprehension by a first-order formula over the elements that are in the projection of some relation. In the presence of nesting the relationship of algebra and logic is more complex, and so in this work we will need to develop some different techniques (e.g. a new kind of interpolation result) to analyze the relationship between logical and algebraic definability.

The development of the nested relational model, culminating in the convergence on the language NRC, has a long history. The thesis of Wong [Wong 1994] and the related paper of Buneman et al. [Buneman et al. 1995] gave an elegant presentation of NRC, and summarize the equivalences known between a number of variations on the syntax. Connections with logic are implicit in results stating that NRC queries can be "simulated" by flat queries: see [Paredaens and Van Gucht 1992; Van den Bussche 2001]. Further discussion on these simulations can be found in Section 5.

More powerful languages than NRC were also considered, including an extension with an operator for forming the powerset of a set. This extension can be captured using the natural logic with membership [Abiteboul and Beeri 1995]. The increased expressiveness implies correspondingly higher complexity (e.g. non-elementary in combined complexity), and perhaps for this reason the subsequent development has focused on NRC. Much of the development of NRC in the last decades has focused primarily on integration with functional languages [Gibbons 2016; Gibbons et al. 2018; Meijer et al. 2006], rather than synthesis or expressiveness.

Quite independently of work on logics for nested relations in computer science, researchers in other areas have investigated the relationships between various restricted algebras for manipulating sets. Gandy [Gandy 1974] defines a class of *Basic functions*, and compares them to functions definable by Δ_0 formulas. Later languages build on Gandy's work, particularly for a finer-grained analysis of the constructible sets [Jensen 1972]. An important distinction from the setting of NRC is that these works do not restrict to sets built up from finitely many levels of nesting above the Ur-elements. For instance, Gandy showed that there are Basic functions checking whether an input is an ordinal, or is the ordinal ω; in fact, he showed that there are Basic functions that are not primitive recursive. In the setting of [Gandy 1974], the Δ_0 functions are strictly more expressive than the Basic functions.

Model theorists have looked at generalizing the Beth definability theorem that relates implicit and explicit definability to the case where the "implicitly definable structure" has new elements, not just new relations. Hodges and his collaborators [Hodges 1993; Hodges et al. 1990] explore this in some restricted cases. Our approach in Section 6 to showing a relationship between implicitly definable transformations and interpretations is inspired by the unpublished draft [Andréka et al. 2008], motivated from the perspective of algebraic logic, which provides model-theoretic tools for connecting semantic and syntactic notions of definability in multi-sorted logic.

Our effective result yields an algorithm translating intuitionistic proofs of functionality into NRC definitions. In contrast, extraction procedures related to the Curry-Howard correspondence typically take as input constructive proofs, possibly with cuts, of statements of the type $\forall x \exists y\, \varphi(x, y)$ witnessing that $\varphi(x, y)$ defines a total relation and turn those proofs into programs for functions f such that $\forall x\, \varphi(x, f(x))$ hold. Our procedure works on cut-free proofs that a formula defines a *partial function* using techniques more closely related to interpolation. This leaves open the question of extracting NRC terms from constructive totality proofs. Sazonov [Sazonov 1985] addressed this question for an untyped analogue of NRC. He uses weak set theories based on intuitionistic Kripke-Platek set theory. These theories are richer than the ones we use for functionality proofs.

3 PRELIMINARIES

Despite their long history of study in several communities, we know of no succinct presentation of the basics of nested collection transformation languages. So we will give a quick introduction here that assumes no background. Indeed, for the issues that we will be concerned with in this work, the aspects of these transformation languages that have been the focus of most past work (e.g. integration with functional languages [Cooper 2009; Meijer et al. 2006] and complexity of evaluation [Koch 2006]) will not be critical.

Nested relations. We deal with schemas that describe objects of various *types* given by the following grammar.

$$T,\ U ::= \mathcal{U} \mid T \times U \mid \mathsf{Unit} \mid \mathsf{Set}(T)$$

For simplicity throughout the remainder we will assume only two basic types: the one-element type Unit and $\mathcal{U}$, whose inhabitant are not specified further; according to the application we may think of $\mathcal{U}$ as being infinite or empty. We call this set the *Ur-elements*. From the Ur-elements and a unit type we can build up the set of types via product and the power set operation. We use standard conventions for abbreviating types, with the n-ary product abbreviating an iteration of binary products. A *nested relational schema* consists of declarations of variable names associated to objects of given types.

Example 3.1. An example nested relational schema declares two objects $R : \mathsf{Set}(\mathcal{U} \times \mathcal{U})$ and $S : \mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U}))$. That is, R is a set of pairs of Ur-elements: a standard "flat" binary relation. S is a collection of pairs whose first elements are Ur-elements and whose second elements are sets of Ur-elements. ◁

The types have a natural interpretation, which we refer to as the *universe over* $\mathcal{U}$. The unit type has a unique member and the members of $\mathsf{Set}(T)$ are the sets of members of T. An *instance* of such a schema is defined in the obvious way, or a $\mathcal{U}$-instance if we want to emphasize the set of Ur-elements on which it is based. Notice that nested relational schemas allow one to describe programming language data structures that are built up inductively via the tupling and set constructors, rather than just sets of tuples. Thus the literature often refers also to the types above as "object types" and to the "complex object data model" [Abiteboul and Beeri 1995; Wong 1994]. In this work we will sometimes refer to the interpretation of a variable in an instance of a nested relational schema as

$$\frac{}{\Gamma,\ x : T,\ \Gamma' \vdash x : T}$$

$$\frac{}{\Gamma \vdash () : \mathsf{Unit}} \qquad \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash \langle e_1, e_2 \rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash e : T_1 \times T_2 \qquad i \in \{1,2\}}{\Gamma \vdash \pi_i(e) : T_i}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \{e\} : \mathsf{Set}(T)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Set}(T_1) \qquad \Gamma,\ x : T_1 \vdash e_2 : \mathsf{Set}(T_2)}{\Gamma \vdash \bigcup \{e_2 \mid x \in e_1\} : \mathsf{Set}(T_2)}$$

$$\frac{}{\Gamma \vdash \emptyset_T : \mathsf{Set}(T)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Set}(T) \qquad \Gamma \vdash e_2 : \mathsf{Set}(T)}{\Gamma \vdash e_1 \cup e_2 : \mathsf{Set}(T)} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Set}(T) \qquad \Gamma \vdash e_2 : \mathsf{Set}(T)}{\Gamma \vdash e_1 \setminus e_2 : \mathsf{Set}(T)}$$

Fig. 1. NRC syntax and typing rules

an *object*. The *subobjects* of an object are defined in the obvious way. For example, if o is an object of type $\mathsf{Set}(T)$, then it is of the form $\{t_1, \ldots\}$, where each t_i is a subobject of o of type T.

For the schema in Example 3.1 above, assuming that $\mathcal{U} = \mathbb{N}$, one possible instance has $R = \{\langle 4, 6\rangle, \langle 7, 3\rangle\}$ and $S = \{\langle 4, \{6, 9\}\rangle\}$.

Transformation languages for nested relations. A *nested relational transformation* (over input schema $\mathcal{SCH}_{in}$ and output schema $\mathcal{SCH}_{out}$) is a function that takes as input an instance of $\mathcal{SCH}_{in}$, and returns an instance of $\mathcal{SCH}_{out}$. For example, suppose our input schema consists of a declaration $R : \mathsf{Set}(\mathcal{U} \times \mathcal{U})$ and our output schema consists also of a declaration $S : \mathsf{Set}(\mathcal{U} \times (\mathsf{Set}(\mathcal{U}))$. Then one possible transformation would return the nested relation formed by grouping on the first position: informally returning a set of pairs $\langle a, s\rangle$ where a is any Ur-element appearing in the first component of a tuple in the input R, and s nt is the set of b such that $\langle a, b\rangle$ is in R.

Transformation equivalence. We say that two transformations are *equivalent* if they agree on all instances (finite and infinite) of a given input schema over any set of Ur-elements. It will turn out that for the transformations we are interested in, "over any set of Ur-elements" can be freely replaced by "over any infinite set of Ur-elements" or "over some fixed infinite set of Ur-elements". When we say that a transformation $\mathcal{T}$ is *expressible* in some class of transformations C, we mean that there is a transformation $\mathcal{T}'$ in C that is equivalent to $\mathcal{T}$ in the sense above.

Nested Relational Calculus. We review the main language for declaratively transforming nested relations, Nested Relational Calculus (NRC). Each expression is associated with an *output type*, which are in the type system described above. We let Bool denote the type $\mathsf{Set}(\mathsf{Unit})$. Then Bool has exactly two elements, and will be used to simulate Booleans.

The grammar and typing rules of NRC expressions are presented in Figure 1.

The definition of the free and bound variables of an expression is standard. For example, the union operator $\bigcup\{E \mid x \in R\}$ binds variable x.

The semantics of these expressions should be fairly evident. If E has type T, and has input variables $x_1 \ldots x_n$ of types $T_1 \ldots T_n$, respectively, then the semantics associates with E a function that given a binding associating each free variable a value of the appropriate type, returns an object of type T. For example, the expression $()$ always returns the empty tuple, while $\emptyset$ returns the empty set of type T. The expression $\{e\}$ evaluates to $\{o\}$, where e evaluates to o.

In the sequel, we thus assume that every NRC expression is implicitly associated with an *input schema*, which declares a list of free variables and their input types, $X_1 : T_1 \ldots X_n : T_n$, along with an output type S. We may write $E : T_1, \ldots, T_n \to S$ and refer to S as the *output type* of E. We often abuse notation by identifying an NRC expression with the associated transformation. For example,

if E is an NRC expression and o_{in} is an object of the input type of E, we will write $E(o_{in})$ for the output of (the function defined by) E on o_{in}.

As explained in [Wong 1994], the following transformations are definable with their expected semantics.

- For every type T there is an NRC expression $=_T$ of type Bool representing equality of elements of type T. In particular, there is an expression $=_{\mathcal{U}}$ representing equality between Ur-elements.
- For every type T there is an NRC expression $\in_T$ of type Bool representing membership between an element of type T in an element of type $\mathrm{Set}(T)$.

Further, if E is a NRC expression with free variable x of type T and F is an expression of type T, then the NRC expression

$$\bigcup\{\{E\} \mid x \in \{F\}\}$$

represents the query obtained by running E with x set to the output of F. Combining this with the first observations above, we can see that for expressions E_1 and E_2 of type T, we have an expression representing $E_1 =_T E_2$ of type Bool. Using this, we will often treat $=_T$ and $\in_T$ as additional constructors of the language.

Boolean operations $\wedge, \vee, \neg$ can also be represented as NRC expressions with output type Bool. For example $\neg\, x$ is just $\{()\} \setminus x$. Applying the observation about composition as we did above, we see that given E of type Bool we can obtain an expression $\neg\, E$ of type Bool, and thus as we did with $=_T$ and $\in_T$ we will treat the Boolean operations as primitives.

Arbitrary arity tupling and projection operations $\langle E_1, \ldots E_n\rangle$, $\pi_j(E)$ for $j > 2$ can be seen as abbreviations for a composition of binary operations. Further

- If B is an expression of type Bool and E_1, E_2 expressions of type $\mathrm{Set}(T)$, then there is an expression $\mathrm{case}(B, E_1, E_2)$ of type $\mathrm{Set}(T)$ that implements "if B then E_1 else E_2".
- If E_1 and E_2 are expressions of type $\mathrm{Set}(T)$, then there is an expression $E_1 \cap E_2$ of type $\mathrm{Set}(T)$.

The derivations of these are not difficult. For example, the conditional required by the first item is given by:

$$\bigcup\{E_1 \mid\ x \in B\} \cup \bigcup\{E_2 \mid x \in (\neg\, B)\}$$

Example 3.2. Consider an input schema including a binary relation $F : \mathrm{Set}(\mathcal{U} \times \mathcal{U})$. The transformation $\mathcal{T}_{\mathsf{Proj}}$ with input F returning the projection of F on the first component can be expressed in NRC as $\bigcup\{\{\pi_1(f)\} \mid f \in F\}$. The transformation $\mathcal{T}_{\mathsf{Filter}}$ with input F and also v of type $\mathcal{U}$ that filters F down to those pairs which agree with v on the first component can be expresses in NRC as $\bigcup\{\mathrm{case}([\pi_1(f) =_{\mathcal{U}} v], \{f\}, \emptyset) \mid f \in F\}$. Consider now the transformation $\mathcal{T}_{\mathsf{Group}}$ that groups F on the first component, returning an object of type $\mathrm{Set}(\mathcal{U} \times \mathrm{Set}(\mathcal{U}))$; this is the first transformation mentioned in Example 1.2. The transformation can be expressed in NRC as $\bigcup\left\{\{\langle v, \bigcup\{\{\pi_2(f)\} \mid f \in \mathcal{T}_{\mathsf{Filter}}\}\rangle\} \mid v \in \mathcal{T}_{\mathsf{Proj}}\right\}$. Finally, consider the second transformation $\mathcal{T}_{\mathsf{Flatten}}$ mentioned in Example 1.2, that flattens an input G of type $\mathrm{Set}(\mathcal{U} \times \mathrm{Set}(\mathcal{U}))$. This can be expressed in NRC as

$$\bigcup\left\{\left\{\bigcup\{\{\langle \pi_1(g), x\rangle\} \mid x \in \pi_2(g)\} \mid g \in G\right\}\right.$$

◁

The language NRC cannot define certain natural transformations whose output type is $\mathcal{U}$, such as, for instance, $\mathrm{case}(B, E_1, E_2)$ for E_1 and E_2 of sort $\mathcal{U}$. To get a canonical language for such transformations, we let NRC[Get] denote the extension of NRC with the family of operations $\mathrm{Get}_T : \mathrm{Set}(T) \to T$ that extracts the unique element from a singleton. Get was considered in [Wong 1994], with connection to parallel evaluation explored in [Suciu 1995]. The semantics are: if E returns a singleton set $\{x\}$, then $\mathrm{Get}_T(E)$ returns x; otherwise it returns some default object of

the appropriate type. The semantics of $\mathrm{Get}_T(x)$ on non-singleton x is not particularly important; to fix ideas, we can define for each type T a default element d_T that will be the output of $\mathrm{Get}_T(x)$ when x is not a singleton assuming that we have a constant c_0 in $\mathcal{U}$: take $d_{\mathcal{U}} = c_0$, $d_{\mathrm{Set}(T)} = \emptyset$, $d_{\mathsf{Unit}} = ()$ and $d_{T_1 \times T_2} = (d_{T_1}, d_{T_2})$. In [Suciu 1995], it is shown that Get is not expressible in NRC at sort $\mathcal{U}$. However, Get_T for general T is definable from $\mathrm{Get}_{\mathcal{U}}$ and the other NRC constructs.

Δ_0 **formulas.** We need a logic appropriate for talking about nested relations. A natural and well-known subset of first-order logic formulas with a set membership relation are the Δ_0 formulas. They are built up from equality of Ur-elements via the Boolean operators $\vee, \neg$ as well as relativized existential and universal quantification. All terms involving tupling and projections are allowed. Formally, we deal with multi-sorted first-order logic, with sorts corresponding to each of our types. We use the following syntax for Δ_0 formulas and terms. Terms are built using tupling and projections. All formulas and terms are assumed to be well-typed in the obvious way, with the expected sort of t and u being $\mathcal{U}$ in expressions $t =_{\mathcal{U}} u$ and $t \neq_{\mathcal{U}} u$, while in $t \in_T u$ the sort of t is T and the sort of u is $\mathrm{Set}(T)$.

$$\begin{array}{lcl} t, u & ::= & x \mid () \mid \langle t, u \rangle \mid \pi_1(t) \mid \pi_2(t) \\ \varphi, \psi & ::= & t =_{\mathcal{U}} t' \mid t \neq_{\mathcal{U}} t' \mid \top \mid \bot \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \forall x \in_T t\ \varphi(x) \mid \exists x \in_T t\ \varphi(x) \end{array}$$

Note that there is no primitive negation or equalities for sorts other than $\mathcal{U}$. This does not limit expressiveness of formulas with respect to classical semantics. Negation $\neg\varphi$ may be defined by induction on φ by dualizing every connective; we write $\varphi \Rightarrow \psi$ for $\neg\varphi \vee \psi$ in the sequel. Equality, inclusion and membership predicates may be defined as notations by induction on the involved types.

$$\begin{array}{rcl@{\qquad}rcl} t \in_T u & := & \exists z' \in u\ t =_T z' & t \subseteq_T u & := & \forall z \in_T t\ z \in_T u \\ t =_{\mathrm{Set}(T)} u & := & t \subseteq_T u \ \wedge\ u \subseteq_T t & t =_{\mathsf{Unit}} u & := & \top \text{ (since all elements of this type are equal)} \end{array}$$
$$t =_{T_1 \times T_2} u \ := \ \pi_1(t) =_{T_1} \pi_1(u) \ \wedge\ \pi_2(t) =_{T_2} \pi_2(u)$$

Here we have not defined $\in$ at higher types as an atomic predicate, but rather as a derived predicate. We can think of the kind of entailments we want to prove in terms of these derived predicates, without use of a set-extensionality axiom:

$$(\forall z \in_T x\ z \in_T y) \ \wedge\ (\forall z \in_T y\ z \in_T x) \ \Rightarrow\ x =_{\mathrm{Set}(T)} y$$

Alternatively, we can think of them as new primitives with extensionality as an axiom relating them to the other primitives we have given above.

The notion of a formula φ *entailing* another formula ψ, writing $\varphi \models \psi$, is the standard one in first-order logic, meaning that every model of φ is a model of ψ.

NRC **and** Δ_0 **formulas.** Since we have a Boolean type in NRC, one may ask about the expressiveness of NRC for defining transformations of shape $T_1, \ldots, T_n \to \mathsf{Bool}$. It turns out that they are equivalent to Δ_0 formulas. This gives one justification for focusing on Δ_0 formulas.

Proposition 3.3. *There is a polynomial time algorithm taking a Δ_0 formula $\varphi(\vec{x})$ as input and producing an* NRC *expression* $\mathsf{Verify}_\varphi(\vec{x})$ *of type* Bool *such that* $\mathsf{Verify}_\varphi(\vec{x})$ *returns true if and only if $\varphi(\vec{x})$ holds.*

This useful result is proved by an easy induction over φ.

4 SYNTHESIZING TRANSFORMATIONS FROM INTUITIONISTIC PROOFS

We will now present our first main result, concerning synthesis of nested relational transformations from proofs.

$$\textsc{Contraction}\ \frac{\Theta;\ \Gamma, \varphi, \varphi \vdash t \in_T u}{\Theta;\ \Gamma, \varphi \vdash t \in_T u} \qquad \in_{\mathcal{U}}\text{-R}\ \frac{}{\Theta,\ t \in_{\mathcal{U}} u;\ \Gamma \vdash t \in_{\mathcal{U}} u}$$

$$=_{\mathsf{Set}}\text{-R}\ \frac{\Theta;\ \Gamma \vdash t \subseteq_T u \qquad \Theta;\ \Gamma \vdash u \subseteq_T t}{\Theta;\ \Gamma \vdash t =_{\mathsf{Set}(T)} u} \qquad =_{\times}\text{-R}\ \frac{\Theta;\ \Gamma \vdash \pi_1(t) =_{T_1} \pi_1(u) \qquad \Theta;\ \Gamma \vdash \pi_2(t) =_{T_2} \pi_2(u)}{\Theta;\ \Gamma \vdash t =_{T_1 \times T_2} u}$$

$$=_{\mathsf{Unit}}\text{-R}\ \frac{}{\Theta;\ \Gamma \vdash t =_{\mathsf{Unit}} u} \qquad =_{\mathcal{U}}\text{-R}\ \frac{\Theta,\ t \in_{\mathcal{U}} z;\ \Gamma \vdash u \in_{\mathcal{U}} z \qquad z \notin \mathsf{FV}(\Theta, \Gamma, t, u)}{\Theta;\ \Gamma \vdash t =_{\mathcal{U}} u}$$

$$\subseteq\text{-R}\ \frac{\Theta,\ z \in_T t;\ \Gamma \vdash z \in_T u \qquad z \notin \mathsf{FV}(\Theta;\ \Gamma, t, u)}{\Theta;\ \Gamma \vdash t \subseteq_T u} \qquad \in_{\mathsf{Set}}\text{-R}\ \frac{\Theta,\ t \in_{\mathsf{Set}(T)} v;\ \Gamma \vdash t =_{\mathsf{Set}(T)} u}{\Theta,\ t \in_{\mathsf{Set}(T)} v;\ \Gamma \vdash u \in_{\mathsf{Set}(T)} v}$$

$$\bot\text{-L}\ \frac{}{\Theta;\ \Gamma,\ \bot \vdash t \in_T u} \qquad \wedge\text{-L}\ \frac{\Theta;\ \Gamma,\ \varphi,\ \psi \vdash t \in_T u}{\Theta;\ \Gamma,\ \varphi \wedge \psi \vdash t \in_T u} \qquad \vee\text{-L}\ \frac{\Theta;\ \Gamma,\ \varphi \vdash t \in_T u \qquad \Theta;\ \Gamma,\ \psi \vdash t \in_T u}{\Theta;\ \Gamma,\ \varphi \vee \psi \vdash t \in_T u}$$

$$\forall\text{-L}\ \frac{\Theta,\ t \in_T z;\ \Gamma,\ \varphi[t/y] \vdash v \in_{T'} w}{\Theta,\ t \in_T z;\ \Gamma,\ \forall y \in_T z\ \varphi \vdash v \in_{T'} w} \qquad \exists\text{-L}\ \frac{\Theta,\ x \in_T y;\ \Gamma,\ \varphi \vdash t \in_{T'} v \qquad x \notin \mathsf{FV}(\Theta, \Gamma, y, t, v)}{\Theta;\ \Gamma,\ \exists x \in_T y\ \varphi \vdash t \in_{T'} v}$$

$$=\text{-}\textsc{subst}\ \frac{\Theta[y/x];\ \Gamma[y/x] \vdash (v \in_T w)[y/x]}{\Theta;\ \Gamma,\ x =_{\mathcal{U}} y \vdash v \in_T w} \qquad \neq\text{-L}\ \frac{}{\Theta;\ \Gamma,\ t \neq_{\mathcal{U}} t \vdash u \in_T v}$$

$$\times_\beta\ \frac{\Theta[t_i/y];\ \Gamma[t_i/y] \vdash (t \in_T u)[t_i/y] \qquad i \in \{1, 2\}}{\Theta[\pi_i(\langle t_1, t_2\rangle)/y];\ \Gamma[\pi_i(\langle t_1, t_2\rangle)/y] \vdash (t \in_T u)[\pi_i(\langle t_1, t_2\rangle)/y]}$$

$$\times_\eta\ \frac{\Theta[\langle x_1, x_2\rangle/x];\ \Gamma[\langle x_1, x_2\rangle/x] \vdash (t \in_T u)[\langle x_1, x_2\rangle/x] \qquad x_1, x_2 \notin \mathsf{FV}(\Theta;\ \Gamma, t, u)}{\Theta;\ \Gamma \vdash t \in_T u}$$

Fig. 2. Our intuitionistic sequent calculus for proofs of implicit definability

We consider an input schema $\mathcal{SCH}_{in}$ with one input object o_{in} and an output schema with one output object o_{out}. Using product objects, we can easily model any nested relational transformation in this way. We deal with a Δ_0 formula $\varphi(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ with distinguished variables $\mathrm{o}_{in}, \mathrm{o}_{out}$. Recall from the introduction that such a formula *implicitly defines* o_{out} *as a function of* o_{in} if for each nested relation o_{in} there is at most one o_{out} such that $\varphi(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ holds for some $\vec{a}$. A formula $\varphi(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ *projectively implicitly defines* a transformation $\mathcal{T}$ from o_{in} to o_{out} if for each o_{in}, $\varphi(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ holds for some $\vec{a}$ if and only if $\mathcal{T}(\mathrm{o}_{in}) = \mathrm{o}_{out}$. We drop "projectively" if $\vec{a}$ is empty.

Example 4.1. Consider the transformation $\mathcal{T}_{\mathrm{Group}}$ from Example 3.2. It has a simple implicit Δ_0 definition as given in Example 1.1, which we can restate as follows. First, define the auxiliary formula $\chi(x, p, R)$ stating that $\pi_1(p)$ is x and $\pi_2(p)$ is the set of y such that $\langle x, y\rangle$ is in R (the "fiber of R above x"):

$$\chi(x, p, R) \ :=\ \pi_1(p) = x \ \wedge\ (\forall t' \in R\ [\pi_1(t') = x \ \Rightarrow\ \pi_2(t') \in \pi_2(p)]) \ \wedge\ \forall z \in \pi_2(p)\ \langle x, z\rangle \in R$$

Then T_{Group} is implicitly defined by $\forall t \in R\ \exists p \in q\ \chi(\pi_1(t), p, R)) \wedge \forall p \in q\ \chi(\pi_1(p), p, R)$. ◁

$$\textsc{wk}\ \frac{\Theta;\ \Gamma \vdash \psi}{\Theta;\ \Gamma,\ \varphi \vdash \psi} \qquad \textsc{ax}\ \frac{}{\Theta;\ \psi \vdash \psi} \qquad \in\textsc{-l}\ \frac{\Theta,\ t \in_T u;\ \Gamma \vdash \psi}{\Theta;\ \Gamma,\ t \in_T u \vdash \psi} \qquad \subseteq\textsc{-l}\ \frac{\Theta,\ t \in_T v;\ \Gamma \vdash \psi}{\Theta,\ t \in_T u;\ \Gamma,\ u \subseteq_T v \vdash \psi}$$

$$\Rightarrow\textsc{-l}\ \frac{\Theta;\ \Gamma,\ \theta \vdash \psi}{\Theta;\ \Gamma,\ \varphi \Rightarrow \theta,\ \varphi \vdash \psi} \qquad =\textsc{-r}\ \frac{}{\Theta;\ \Gamma \vdash t =_T t}$$

Fig. 3. Some typical admissible rules.

Restricted proof system. Our synthesis result requires a proof of functionality within a restricted proof system. We present a special-purpose sequent calculus in Figure 2 deriving judgments $\Theta;\ \Gamma \vdash \varphi$ where Γ is a multi-set of Δ_0 formulas, Θ a multi-set of membership formulas $t \in u$, and φ is a Δ_0 formula with one of the following shapes: $t \in_T u$, $t =_T u$ or $t \subseteq_T u$. A multi-set of formulas will also be called a *context*, and above we write $C,\ C'$ for the concatenation of contexts C and C'. Informally, a judgment $\Theta;\ \Gamma \vdash \varphi$ is meant to be read as "If all the containments in Θ and formulas in Γ hold, then φ does". In the figure, we use FV to denote the free variables of a context, and we use $\varphi[t/x]$ to denote the result of substituting t for x in φ.

The main essential restriction on the proof system is that it is intuitionistic. There is no way to deduce $\Theta;\ \Gamma \vdash \varphi$ from $\Theta;\ \Gamma,\ \neg\varphi \vdash \bot$ in general. Informally, this means that we forbid reasoning by contradiction. In particular, this means that some sequents are classically valid but not derivable in our calculus. For instance, consider $w \in r;\ \forall x \in l\ l \in r,\ \forall y \in w\ l \in r \vdash l \in r$. This is seen to be classically valid by considering separately the following three cases: l non-empty, w non-empty and $l = w = \emptyset$. However, it is also easy to check that this cannot be derived intuitionistically. The other restrictions, such as the specific shape of formulas on the right-hand side for many rules, do not limit the power of the system when it comes to functionality proofs, but allow us to prove our main extraction result more easily.

It is straightforward to capture the informal reasoning used to argue for functionality in Example 1.1 within our proof system. We also note that many natural proof rules are *admissible* in our system; they are conservative in terms of the set of proofs that they enable. We collect the most useful cases in Figure 3. Showing that they are admissible is done by rather elementary inductions, and it can be noted that eliminating those additional proof rules can be done in polynomial time in the size of proof trees and the types of the involved formulas. This list is not meant to be exhaustive, as it can be shown that the derivable sequents in our system are exactly those derivable in more standard sequent calculus for multi-sorted intuitionistic logic that appear in the prior literature (see e.g. [Jacobs 2001, Section 4.1]). We offer a detailed discussion of the correspondence between our proof system and several previously known intuitionistic calculi in the supplementary materials.

A technicality is that in our presentation of the proof system there is a slight asymmetry between how the set predicates $=_T$, $\subseteq_T$ and $\in_T$ are treated on the left and on the right. The proof rules decomposing formulas on the right, such as $\subseteq$-R, are specialized to deal with the semantics of these predicates. They are justified either based on extensionality – if one thinks of these predicates as primitive – or by definition, if one thinks of these predicates as derived. On the other hand, on the left side we require that all of our formulas in Γ are described in the basic grammar of Δ_0 formulas, which does not have these predicates as atomic. We do this only for convenience, to avoid having additional proof rules capturing extensionality in decomposing formulas on the left.

Provably implicit definitions. By an intuitionistic proof that $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ implicitly defines o_{out} as a function of o_{in} we mean a formal derivation of a sequent $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a}),\ \Sigma(\mathrm{o}_{in}, \mathrm{o}'_{out}, \vec{a}') \vdash \mathrm{o}_{out} =_T \mathrm{o}'_{out}$ in our proof system.

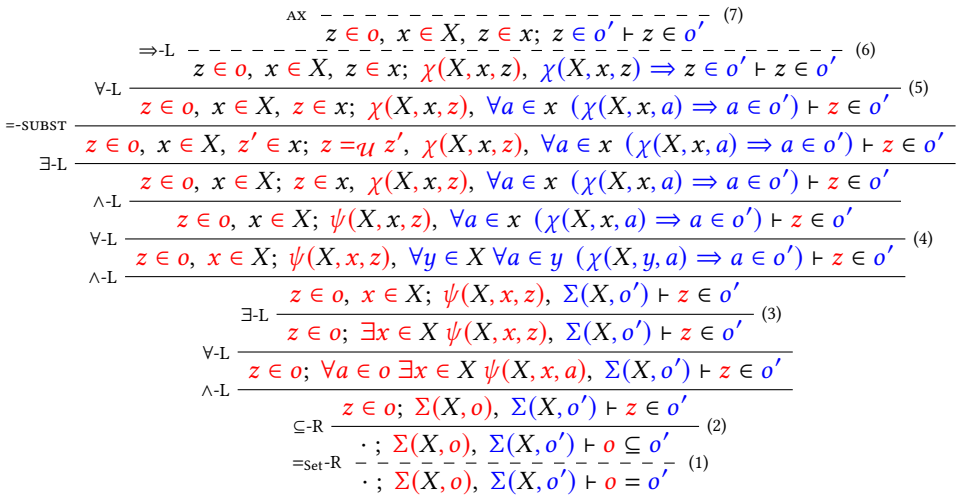

Fig. 4. Formal proof tree of functionality for Example 4.3. Admissible rules are denoted with dashed lines and some instances of the admissible weakening rule (WK) are omitted for legibility. Formulas and variables specific to the left and right-hand side are respectively colored in red and blue.

We can now state our main result on effectively generating NRC expressions from proofs:

Theorem 4.2. *There is a* PTIME *procedure which takes as input an intuitionistic proof that* $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ *defines* o_{out} *as a function of* o_{in}*, and returns an* NRC *expression* E *such that whenever* $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ *holds, then* $E(\mathrm{o}_{in}) = \mathrm{o}_{out}$.

Let us provide a detailed example to illustrate Theorem 4.2.

Example 4.3. Given a set of sets of Ur-elements $X \in \mathrm{Set}(\mathrm{Set}(\mathcal{U}))$, say that an Ur-element a *distinguishes* a set $x \in X$ if x is the unique element of X containing a. Consider the transformation taking as input such an X and returning the set of Ur-elements that distinguish some element of X. This is implicitly definable by a Δ_0 formula $\Sigma(X, o)$ stating that every a in o distinguishes some element of X and conversely. Writing this in our restricted syntax for Δ_0 formulas, in which membership of higher-order objects must be expressed using bounded quantification and equality, we obtain an implicit definition

$$\Sigma(X,o) \;:=\; (\forall a \in o\ \exists x \in X\ \psi(X,x,a)) \;\wedge\; (\forall x \in X\ \forall a \in x\ [\chi(X,x,a) \;\Rightarrow\; a \in_{\mathcal{U}} o]) \quad \text{where}$$
$$\chi(X,x,a) \;:=\; \forall y \in X\ \big(a \in_{\mathcal{U}} y \;\Rightarrow\; x =_{\mathrm{Set}(\mathcal{U})} y\big) \quad \text{and} \quad \psi(X,x,a) \;:=\; a \in_{\mathcal{U}} x \;\wedge\; \chi(X,x,a)$$

Note that when $a \in_{\mathcal{U}} x$, $x =_{\mathrm{Set}(\mathcal{U})} y$, and $a \in_{\mathcal{U}} o$ occur on the left side of a sequent, they should be thought of as abbreviations for more complex formulas built up through bounded quantification. Similarly $\Rightarrow$ is a derived connective, built up from the Boolean operations allowed in Δ_0 formulas in the obvious way. ◁

Figure 4 contains a formal derivation of functionality for $\Sigma(X, o)$. We may render this proof informally as follows (putting references to proof steps in Figure 4 in parentheses).

Proof of functionality of Example 4.3. Assume $\Sigma(X, o)$ and $\Sigma(X, o')$. To show $o = o'$, we need to show that $o \subseteq o'$ and $o' \subseteq o$. Since the roles of o and o' are symmetric, without loss of generality, it suffices to give the proof that $o \subseteq o'$ (1). So fix $z \in o$ (2). Since $\Sigma(X, o)$ holds, according

to its first conjunct, we have in particular that there exists some $x \in X$ such that $\psi(X, x, z)$ holds (3). Because $\Sigma(X, o')$ holds and $x \in X$, the second conjunct tells us that for every $a \in x$, we have $\chi(X, x, a) \Rightarrow a \in o'$ (4). Recall that $\psi(X, x, z)$ is the conjunction of $z \in x$ and $\chi(X, x, z)$, so that we may deduce that $\chi(X, x, z) \Rightarrow z \in o'$ (6) and thus $z \in o'$ (7). □

As per Theorem 4.2, the transformation defined in Example 4.3 is NRC-definable as

$$\bigcup \left\{ \mathsf{case}(\mathsf{Verify}_\theta(X, a), \{a\}, \emptyset) \mid a \in \bigcup X \right\} \qquad \text{with} \quad \theta(X, a) = \exists x \in X\ \psi(X, x, a)$$

where Verify is the filtering function given by Proposition 3.3.

We emphasize that our results apply to proofs of functionality over *any subsignature* of the input. In particular they apply to synthesize inverses of transformations, a problem of considerable interest in several communities [Hu and D'Antoni 2017; Srivastava et al. 2011]:

Example 4.4. Return to the setting of Example 1.1, and suppose that we are interested in the transformation over an input object G of type $\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U}))$ which simply "flattens" G. We write this explicitly in NRC, as we did in Example 3.2:

$$E = \bigcup \left\{ \bigcup \{ \{ \langle \pi_1(g), t \rangle \} \mid t \in \pi_2(g) \} \mid g \in G \right\}$$

From E we can automatically generate a Δ_0 formula such as Σ from Example 1.1, stating that F is the output of G under E. Indeed, this is true for any NRC transformation: one just encodes the semantics of NRC in logic.

This transformation is invertible, as mentioned in Example 1.1, and we can prove its invertibility in our calculus. Our synthesis algorithm will generate from this proof an expression in NRC that represents the inverse, namely an expression that groups F to form G. ◁

Example 4.5. Another application are for the synthesis result of Theorem 4.2 is to rewrite transformations using cached results, a variation on the idea of "rewriting with views" in relational databases [Afrati and Chirkova 2019; Halevy 2001; Lenzerini 2002; Nash et al. 2010; Toman and Weddell 2011].

Consider a sequence where assigns to variable J of type $\mathsf{Set}(\mathcal{U} \times \mathcal{U})$ the intersection of A and B, and later assigns to variable S of type $\mathsf{Set}(\mathcal{U})$ the set of elements that have a self-loop in both A and B.

$$J := A \cap B; \ldots; S := \bigcup \left\{ \bigcup \{ \mathsf{case}(\pi_1(a) = \pi_2(a) = \pi_1(b) = \pi_2(b), \{\pi_1(a)\}, \emptyset) \mid a \in A \} \mid b \in B \right\}; \ldots$$

One can easily see that S is a function of J. And from a proof of functionality, our method produces a rewriting of the assignment producing S, using an NRC expression that makes use of J. An example of such a rewriting is

$$S := \bigcup \{ \mathsf{case}(\pi_1(j) = \pi_2(j), \{\pi_1(j)\}, \emptyset) \mid j \in J \}$$

Such a rewriting of S using the cached value of J may be much more efficient than recomputing S from scratch. ◁

We now turn to explaining the ingredients that underlie the procedure of Theorem 4.2.

Interpolation for Δ_0 formulas. Often a key ingredient in moving from implicit to explicit definition is an *interpolation theorem*, stating that for each entailment between formulas φ_L and φ_R there is an intermediate formula (an *interpolant* for the entailment), which is entailed by φ_L and entails φ_R while using only symbols common to φ_L and φ_R. We can show using a standard inductive approach to interpolation (e.g. [Fitting 1996]) that our calculus admits efficient interpolation.

Proposition 4.6. *Let $\Theta_L, \Theta_R, \Gamma_L$ and Γ_R be contexts and ψ a formula and call $C = \mathsf{FV}(\Theta_L, \Gamma_L) \cap \mathsf{FV}(\theta_R, \Gamma_R)$ the set of common free variables. For every derivation $\Theta_L;\ \Theta_R;\ \Gamma_L,\ \Gamma_R \vdash \psi$ there exists a Δ_0 formula θ with $\mathsf{FV}(\theta) \subseteq C$ such that the following holds*

$$\Theta_L;\ \Gamma_L \models \theta \qquad \text{and} \qquad \Theta_R;\ \Gamma_R,\ \theta \models \psi$$

Further the interpolant θ can be found in polynomial time from the derivation.

The interpolation result above should be thought of as giving us the result we want for transformations of *Boolean* type. From it we can derive that a formula whose truth value is implicitly defined by a set of input variables must be given as a Δ_0 formula over those inputs. By Proposition 3.3, these formulas can be converted to NRC.

The higher-type interpolation lemma. Our main result is deduced from a more general interpolation result, which says that whenever a binary relationship between variables, such as the containment relationship $t \subseteq_T u$, is provable from a theory that is partitioned into left and right formulas, and the variables t and u appear exclusively in distinct sides of the partition, then there is an *interpolating expression* in NRC[Get], taking as input the variables common to the left and right partitions. For an equality relationship between variables, the synthesized expression will take as input the common variables on the left and right and select an object that is equal to the variables participating in the equality. For membership relationships $t \in u$, our algorithm derives a bounding expression E taking inputs in the common signature such that $t \in E$; this could be strengthened to $t \in E \subseteq u$. The result bears some similarity with other extraction procedures that produce a program from a proof, such as those based on the Curry-Howard correspondence. However, it is formally much closer to the kind of interpolation theorem from logic mentioned earlier in connection to Proposition 4.6. In the past, interpolation results have been applied to extract program invariants [Hoder et al. 2010; McMillan 2003]; here we are proving and applying interpolation results to produce a different kind of program artifact.

Lemma 4.7. *[Higher-type Interpolation Lemma] Let $\Theta = \Theta_L, \Theta_R$ be a $\in$-context and $\Gamma = \Gamma_L, \Gamma_R$ a context. Suppose that t and u are terms of suitable types such that $\mathsf{FV}(t) \subseteq \mathsf{FV}(\Theta_L, \Gamma_L)$ and $\mathsf{FV}(u) \subseteq \mathsf{FV}(\Theta_R, \Gamma_R)$ and call $C = \mathsf{FV}(\Theta_L, \Gamma_L) \cap \mathsf{FV}(\Theta_R, \Gamma_R)$ the set of common free variables. Then we have:*

- *If $\Theta;\ \Gamma \vdash t =_T u$ is derivable, there is an NRC[Get] expression E of type T such that*

$$\Theta;\ \Gamma \models t = E = u \quad \text{and} \quad \mathsf{FV}(E) \subseteq C$$

- *If $\Theta;\ \Gamma \vdash t \subseteq_T u$ is derivable, there is an NRC[Get] expression E of type $\mathsf{Set}(T)$ such that*

$$\Theta;\ \Gamma \models t \subseteq E \subseteq u \quad \text{and} \quad \mathsf{FV}(E) \subseteq C$$

- *If $\Theta;\ \Gamma \vdash t \in_T u$ is derivable, then there is an NRC[Get] expression E of type $\mathsf{Set}(T)$ such that*

$$\Theta;\ \Gamma \models t \in E \quad \text{and} \quad \mathsf{FV}(E) \subseteq C$$

Further the desired expressions can be constructed in time polynomial in the proof.

Proof of Theorem 4.2. A proof that $\Sigma(\mathsf{o}_{in}, \mathsf{o}_{out}, \vec{a})$ defines o_{out} as a function of o_{in} is exactly a proof that $\Sigma(\mathsf{o}_{in}, \mathsf{o}_{out}, \vec{a}),\ \Sigma(\mathsf{o}_{in}, \mathsf{o}'_{out}, \vec{a}') \vdash \mathsf{o}_{out} =_T \mathsf{o}'_{out}$ where o'_{out} and $\vec{a}'$ are new variables. Applying Lemma 4.7 with Θ empty, $\Gamma_L = \Sigma(\mathsf{o}_{in}, \mathsf{o}_{out}, \vec{a})$, and $\Gamma_R = \Sigma(\mathsf{o}_{in}, \mathsf{o}'_{out}, \vec{a}')$ yields an NRC[Get] expression $E(\mathsf{o}_{in})$ such that $\Sigma(\mathsf{o}_{in}, \mathsf{o}_{out}, \vec{a}),\ \Sigma(\mathsf{o}_{in}, \mathsf{o}'_{out}, \vec{a}') \models \mathsf{o}_{out} = E(\mathsf{o}_{in}) = \mathsf{o}'_{out}$. Hence we have $\Sigma(\mathsf{o}_{in}, \mathsf{o}_{out}, \vec{a}) \models \mathsf{o}_{out} = E(\mathsf{o}_{in})$ and the proof of Theorem 4.2 is complete. □

Lemma 4.7 is proven by induction on the derivation, which requires examining every proof rule in Figure 2. The more interesting cases are the left-hand side rules for first-order connectives ($\wedge$-L, $\vee$-L, $\forall$-L and $\exists$-L) and the rules for the right-hand side formulas $\in_{\mathsf{Set}}$-R and $=_{\mathcal{U}}$-R. Regarding the

left-hand side rules, since the right-hand side formula of both the premise and conclusion is of the shape $t \in_T u$, the inductive invariant requires us to output an NRC expression bounding the term t. To prove the inductive step, we use the binary union operator $E_1 \cup E_2$ of NRC for the rule $\vee$-L and the big union operator $\bigcup\{E \mid x \in y\}$ for the rule $\exists$-L. On the other hand, the inductive steps for the rules $\wedge$-L and $\forall$-L do not require modifying the expression obtained as part of the induction hypothesis. To treat the inductive steps corresponding to the rules $\subseteq$-R and $=_{\mathcal{U}}$-R, we use a combination of the usual "Boolean" interpolation (Proposition 4.6) and the conversion of Δ_0 formulas to expressions of Boolean type in NRC (Proposition 3.3).

Example 4.8. Let us illustrate the algorithm provided by Lemma 4.7 on the proof tree in Figure 4 by providing the corresponding intermediate NRC expressions that are synthesized, starting from top to bottom: from step (7) to (5), the NRC expression is the singleton $\{z'\}$. After the conclusion of the subsequent $\exists$-L rule, the expression becomes

$$\bigcup\{\{z'\} \mid z' \in x\}$$

which is semantically equivalent to x. After the next $\exists$-L rule at step (3), we obtain

$$\bigcup\left\{\bigcup\{\{z'\} \mid z' \in x\} \mid x \in X\right\}$$

which is equivalent to the union $\bigcup X$. The final expression is then obtained right after step (2), by first computing an interpolant $\theta(X, z)$ such that $z \in o \wedge \varphi(X, o) \models \theta(X, z)$ and $\theta(X, z) \wedge \varphi(X, o') \models z \in o'$. Computing according to the procedure underlying Proposition 4.6 yields $\theta(X, a) = \exists x \in X\ \psi(X, x, a)$ and the final NRC expression

$$\bigcup\left\{\mathsf{case}(\mathsf{Verify}_\theta(X, a), \{a\}, \emptyset) \mid a \in \bigcup\left\{\bigcup\{\{z'\} \mid z' \in x\} \mid x \in X\right\}\right\}$$

◁

We now detail two cases of the inductive argument required to prove Lemma 4.7, the other cases being relegated to the supplementary materials. We also omit the routine complexity analysis of the underlying algorithm.

Rule $\forall$-L: Assume that the last proof rule used introduces a universal quantifier on the left.

$$\forall\text{-L}\ \frac{\Theta,\ w \in_T y;\ \Gamma,\ \varphi[w/x] \vdash t \in_{T'}}{\Theta,\ w \in_T y;\ \Gamma,\ \forall x \in_T y\ \varphi \vdash t \in_{T'} v}$$

To simplify matters, assume that w is a variable. We apply the induction hypothesis to obtain a NRC expression, say E' with $\mathsf{FV}(E') \subseteq \{w\} \cup C$, by splitting the $\Theta,\ w \in_T y;\ \Gamma,\ \varphi[w/x]$ in the obvious way (e.g., if $\forall x \in_T y\ \varphi$ was on the left context in the conclusion, we make $\varphi[w/x]$ part of the left context in the premise). If $w \notin \mathsf{FV}(E')$, then it also satisfies the invariant in the conclusion. Otherwise, it must be the case that $y \in C$. Hence, we may show that the invariant is satisfied by

$$E = \bigcup\{E' \mid w \in y\}$$

Rule $\subseteq$-R: If the last proof rule used introduces an inclusion on the right

$$\subseteq\text{-R}\ \frac{\Theta,\ z \in_T t;\ \Gamma \vdash z \in_T u \qquad z \notin \mathsf{FV}(\Theta,\ \Gamma, t, u)}{\Theta;\ \Gamma \vdash t \subseteq_T u}$$

then the inductive hypothesis gives us an expression E' such that $\Theta,\ z \in_T t;\ \Gamma \models z \in_T E'$ and $\mathsf{FV}(E') \subseteq C$. Apply interpolation to the premise so as to obtain a Δ_0 formula θ with $\mathsf{FV}(\theta) \subseteq \{z\} \cup C$ such that $\Theta_L;\ \Gamma_L,\ z \in_T t \models \theta$ and $\Theta_R;\ \Gamma_R,\ \theta \models z \in_T u$. In this case, we take $E = \{z \in E' \mid \theta\}$, which is NRC[Get]-definable as

$$\bigcup\{\mathsf{case}(\mathsf{Verify}_\theta, \{z\}, \emptyset) \mid z \in E'\}$$

Now, let us assume that Γ holds and show that $t \subseteq E$ and $E \subseteq u$.

- Suppose that $z \in t$. By the induction hypothesis, we know that $z \in E'$. But we also know that Γ_L is satisfied, so that θ holds. By definition, we thus have $z \in E$.
- Now suppose that $z \in E$, that is, that $z \in E'$ and θ holds. The latter directly implies that $z \in u$ since Γ_R holds.

5 INTERPRETATIONS AND NESTED RELATIONS

We will be interested in extending our synthesis result to classical proofs. But first we give another characterization of NRC, an equivalence with transformations defined by *interpretations*.

We first review the notion of an interpretation, which has become a common way of defining transformations using logical expressions [Bojanczyk et al. 2018; Colcombet and Löding 2007]. Let $\mathcal{SCH}_{in}$ and $\mathcal{SCH}_{out}$ be multi-sorted vocabularies. A first-order interpretation with input signature $\mathcal{SCH}_{in}$ and output signature $\mathcal{SCH}_{out}$ consists of:

- for each output sort S', a sequence of input sorts $\tau(\mathrm{S}') = \vec{\mathrm{S}}$,
- a formula $\varphi^{\mathrm{S}'}_{\equiv}(\vec{x}_1, \vec{x}_2)$ for each output sort S' in $\mathcal{SCH}_{out}$ (where both tuples of variables $\vec{x}_1$ and $\vec{x}_2$ have types $\tau(\mathrm{S}')$),
- a formula $\varphi^{\mathrm{S}'}_{\mathrm{Domain}}(\vec{x}_1)$ for each output sort S' in $\mathcal{SCH}_{out}$ (the variables $\vec{x}_1$ have types $\tau(\mathrm{S}')$),
- a formula $\varphi_R(\vec{x}_1, \ldots \vec{x}_n)$ for every relation R of arity n in $\mathcal{SCH}_{out}$ (where the variables $\vec{x}_i$ have types $\tau(\mathrm{S}'_i)$, provided the i-th argument of R has sort S'_i),
- for every function symbol $f(x_1, \ldots, x_k)$ of $\mathcal{SCH}_{out}$ with output sort S' and input x_i of sort S_i, a sequence of terms $\overline{f}_1(\vec{x}_1, \ldots, \vec{x}_k), \ldots, \overline{f}_m(\vec{x}_1, \ldots, \vec{x}_k)$ with sorts $\tau(\mathrm{S}_{out})$ and $\vec{x}_i$ of sorts $\tau(\mathrm{S}_i)$.

subject to the following constraints:

- $\varphi^{\mathrm{S}}_{\equiv}(\vec{x}, \vec{y})$ should define a partial equivalence relation, i.e. be symmetric and transitive,
- $\varphi^{\mathrm{S}}_{\mathrm{Domain}}(\vec{x})$ should be equivalent to $\varphi^{\mathrm{S}}_{\equiv}(\vec{x}, \vec{x})$,
- $\varphi_R(\vec{x}_1, \ldots, \vec{x}_n)$ and $\varphi^{\mathrm{S}_i}_{\equiv}(\vec{x}_i, \vec{y}_i)$ for $1 \leq i \leq n$, where S_i is the output sort associated with position i of the relation R, should jointly imply $\varphi_R(\vec{y}_1, \ldots, \vec{y}_n)$.
- the formulas $\varphi^{\mathrm{S}}_{\equiv}$ should be congruent with the interpretation of terms: for every output function symbol $f(x_1, \ldots, x_k)$ represented by terms $\overline{f}_1(\vec{x}_1, \ldots, \vec{x}_k), \ldots, \overline{f}_m(\vec{x}_1, \ldots, \vec{x}_k)$, writing $\vec{x}$ for the concatenation of $\vec{x}_1, \ldots, \vec{x}_k$, and $\vec{y}$ for the concatenation of $\vec{y}_1, \ldots, \vec{y}_k$, we enforce

$$\forall\ \vec{x}\ \vec{y} \quad \left(\bigwedge_{i=1}^{k} \varphi^{\mathrm{S}_i}_{\equiv}(\vec{x}_i, \vec{y}_i) \right) \implies \varphi^{\mathrm{S}'}_{\equiv}\left(\overline{f}_1(\vec{x}), \ldots, \overline{f}_m(\vec{x}), \overline{f}_1(\vec{y}), \ldots, \overline{f}_m(\vec{y}) \right)$$

  where S' is the sort of the output of f and the S_i correspond to the arities.

In $\varphi^{\mathrm{S}}_{\equiv}$ and $\varphi^{\mathrm{S}}_{\mathrm{Domain}}$, each $\vec{x}_1, \vec{x}_2$ is a tuple containing variables of sorts agreeing with the prescribed sequence of input sorts for S'. Given a structure M for the input sorts and a sort S we call a binding of these variables to input elements of the appropriate input sorts an M, S *input match*. If in output relation R position i is of sort S_i, then in $\varphi_R(\vec{t}_1, \ldots \vec{t}_n)$ we require $\vec{t}_i$ to be a tuple of variables of sorts agreeing with the prescribed sequence of input sorts for S_i. Each of the above formulas is over the vocabulary of $\mathcal{SCH}_{in}$. An interpretation $\mathcal{I}$ defines a function from structures over vocabulary $\mathcal{SCH}_{in}$ to structures over vocabulary $\mathcal{SCH}_{out}$ as follows:

- The domain of sort S' is the set of equivalence classes of the partial equivalence relation defined by $\varphi^{\mathrm{S}'}_{\equiv}$ over the M, S' input matches.
- A relation R in the output schema is interpreted by the set of those tuples $\vec{a}$ such that $\varphi_R(\vec{t}_1, \ldots \vec{t}_n)$ holds for some $\vec{t}_1 \ldots \vec{t}_n$ with each $\vec{t}_i$ a representative of a_i.

An interpretation $\mathcal{I}$ also defines a map $\varphi \mapsto \varphi^*$ from formulas over $\mathcal{SCH}_{out}$ to formulas over $\mathcal{SCH}_{in}$ in the obvious way. This map commutes with all logical connectives and thus preserves logical consequence.

In the sequel, we are concerned with interpretations preserving certain theories consisting of sentences in first-order logic. Recall that a *theory* in first-order logic is just a set of sentences. Given a theory Σ over $\mathcal{SCH}_{in}$ and a theory Σ' over $\mathcal{SCH}_{out}$, we say that $\mathcal{I}$ is an interpretation of Σ' within Σ if $\mathcal{I}$ is an interpretation such that for every theorem φ of Σ', φ^* is a theorem of Σ. Since $\varphi \mapsto \varphi^*$ preserves logical consequence, if Σ' is generated by a set of axioms A, it suffices to check that Σ proves φ^* for $\varphi \in A$.

Finally, we are also interested in interpretations restricting to the identity on part of the input. Suppose that $\mathcal{SCH}_{out}$ and $\mathcal{SCH}_{in}$ share a sort S. An interpretation $\mathcal{I}$ of $\mathcal{SCH}_{out}$ within $\mathcal{SCH}_{in}$ is said to preserve S if the output sort associated to S is S itself and the induced map of structures is the identity over S. Up to equivalence, that means we fix $\varphi^T_{\mathsf{Domain}}(x)$ to be, up to equivalence, $\top$, $\varphi^{\mathsf{S}}_{\equiv}(x, y)$ to be the equality $x = y$ and map constants of type S to themselves.

Interpretations defining nested relational transformations. We now consider how to define nested relational transformations via interpretations. The main idea will be to restrict all the constituent formulas to be Δ_0 and to relativize the notion of interpretation to a background theory that corresponds to our sanity axioms about tupling and sets.

We define the notion of *component types* of a type T inductively as follows.

- T is a component type of $\mathsf{Set}(T')$ if $T = \mathsf{Set}(T')$ or if it is a component type of T'.
- T is a component type of $T_1 \times T_2$ if $T = T_1 \times T_2$ or if it is a component type of either T_1 or T_2.
- The only component types of $\mathcal{U}$ and Unit are themselves.

Note in particular that if we have a complex object of sort T, the possible sorts over its subobjects are exactly the component types of T.

For every type T, we build a multi-sorted vocabulary $\mathcal{SCH}_T$ as follows.

- The sorts are all component types of T, Unit and $\mathsf{Bool} = \mathsf{Set}(\mathsf{Unit})$.
- The function symbols are the projections, tupling, the unique element of type Unit, the constants ff, tt of sort Bool representing $\emptyset, \{()\}$ and a special constant o of sort T.
- The relation symbols are the equalities at every sort and the membership predicates $\in_T$.

Let T_{obj} be a type which will represent the type of a complex object obj. We build a theory $\Sigma(T_{\mathrm{obj}})$ on top of $\mathcal{SCH}_{T_{\mathrm{obj}}}$ from the following axioms:

- Equality should satisfy the congruence axioms for every formula φ

$$\forall xy\ (x = y\ \wedge\ \varphi\ \Rightarrow\ \varphi[y/x])$$

  Note that it is sufficient to require this for atomic formulas to infer it for all formulas.
- We require that projection and tupling obey the usual laws for every type of $\mathcal{SCH}_{T_{\mathrm{obj}}}$.

$$\forall x^{T_1}\ y^{T_2}\ \pi_1(\langle x, y\rangle) = x \qquad \forall x^{T_1}\ y^{T_2}\ \pi_2(\langle x, y\rangle) = y \qquad \forall x^{T_1 \times T_2}\ \langle \pi_1(x), \pi_2(x)\rangle = x$$

- We require that Unit be a singleton and every $\mathsf{Set}(T)$ in $\mathcal{SCH}_{T_{\mathrm{obj}}}$

$$\forall x^{\mathsf{Unit}}\ () = x$$

- Lastly our theory imposes set extensionality

$$\forall x^{\mathsf{Set}(T)}\ y^{\mathsf{Set}(T)}\ \Big([\forall z^T\ (z \in_T x \Leftrightarrow z \in_T y)] \Rightarrow x =_T y\Big)$$

Note that in interpretations we associate the input to a structure that includes a distinguished constant. For example, an input of type $\mathsf{Set}(\mathcal{U})$ will be coded by a structure with an element relation, an Ur-element sort, and a constant whose sort is the type $\mathsf{Set}(\mathcal{U})$. In other contexts, like NRC expressions and implicit definitions of transformations, we considered inputs to be *free variables*.

This is only a change in terminology, but it reflects the fact that in evaluating the interpretation on any input i_0 we will keep the interpretation of the associated constant fixed, while we need to look at multiple bindings of the variables in each formula in order to form the output structure.

We will show that NRC[Get] expressions defining transformations from a nested relation of type T_1 to a nested relation of type T_2 correspond to a subset of interpretations of $\Sigma(T_2)$ within $\Sigma(T_1)$ that preserve $\mathcal{U}$. The only additional restriction we impose is that all formulas $\varphi^T_{\mathsf{Domain}}$ and $\varphi^T_{\equiv}$ in the definition of such an interpretation must be Δ_0. This forbids, for instance, universal quantification over the whole set of Ur-elements. We thus call a first-order interpretation of $\Sigma(T_2)$ within $\Sigma(T_1)$ consisting of Δ_0 formulas a Δ_0 *interpretation of* $\Sigma(T_2)$ *within* $\Sigma(T_1)$.

We now describe what it means for such an interpretation to define a transformation from an instance of one nested relational schema to another; that is, to map one object to another. We will denote the distinguished constant lying in the input sort by o_{in} and the distinguished constant in the output sort by o_{out}. Given any object o of type T, define M_o as the least structure such that

- every subobjects of o is part of M_o
- when $T_1 \times T_2$ is a component type of T and a_1, a_2 are objects of sort T_1, T_2 of M_o, then $\langle a_1, a_2\rangle$ is an object of M_o
- a copy of $\emptyset$ is part of M_o for every sort $\mathsf{Set}(T)$ in $\mathcal{SCH}_T$
- $()$ and $\{()\}$ are in M_o at sorts Unit and Bool.

The map $o \mapsto M_o$ shows how to translate an object to a logical structure that is appropriate as the input of an interpretation. Note that M_o satisfies $\Sigma(T)$ and that every sort has at least one element in M_o and that there is one sort, Bool, which contains two elements; these technicality are important to ensure that interpretation be expressive enough.

We now discuss how the output of an interpretation is mapped back to an object. The output of an interpretation is a multi-sorted structure with a distinguished constant o_{out} encoding the output nested relational schema, but it is not technically a nested relational instance as required by our semantics for nested relational transformations. For example, an element of $M_{\mathsf{Set}(\mathcal{U})}$ is not a set of Ur-elements, but simply a value connected to Ur-elements by a membership relation. We can convert the output to a semantically appropriate entity via a modification of the well-known Mostowski collapse [Mostowski 1949]. We define $\mathsf{Collapse}(e, M)$ on elements e of the domain of a structure M for the multi-sorted encoding of a schema, by structural induction on the type of e:

- If e has sort $T_1 \times T_2$ then we set $\mathsf{Collapse}(e, M) = \langle \mathsf{Collapse}(\pi_1(e), M), \mathsf{Collapse}(\pi_2(e), M)\rangle$
- If e has sort $\mathsf{Set}(T)$, then we set $\mathsf{Collapse}(e, M) = \{\mathsf{Collapse}(t, M) \mid t \in e\}$
- Otherwise, if e has sort Unit or $\mathcal{U}$, we set $\mathsf{Collapse}(e, M) = e$

We now formally describe how Δ_0 interpretations define functions between objects in the nested relational data model.

Definition 5.1. We say that a nested relational transformation $\mathcal{T}$ from T_1 to T_2 is defined by a Δ_0 interpretation $\mathcal{I}$ if, for every object o_{in} of type T_1, the structure M associated with o_{in} is mapped to M' where $\mathcal{T}(\mathsf{o}_{in})$ is equal to $\mathsf{Collapse}(\mathsf{o}_{out}, M')$.

We will often identify a Δ_0 interpretation with the corresponding transformation, speaking of its input and output as a nested relation (rather than the corresponding structure). For such an interpretation $\mathcal{I}$ and an input object o_{in} we write $\mathcal{I}(\mathsf{o}_{in})$ for the output of the transformation defined by $\mathcal{I}$ on o_{in}.

Example 5.2. Consider an input schema consisting of a single binary relation $R : \mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U}))$, so an input object is a set of pairs, with each pair consisting of an Ur-element and a set of Ur-elements. The corresponding theory is $\Sigma(\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U})))$, which has sorts $\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U}))$, $\mathcal{U} \times \mathsf{Set}(\mathcal{U})$, $\mathsf{Set}(\mathcal{U})$ and $\mathcal{U}$ and relation symbols $\in_{\mathcal{U}}$ and $\in_{\mathcal{U}\times\mathsf{Set}(\mathcal{U})}$ and one equality symbol for

each above sort. If we consider the following instance of the nested relational schema

$$R_0 = \{\langle a, \{a, b\}\rangle, \langle a, \{a, c\}\rangle, \langle b, \{a, c\}\rangle\}$$

Then the corresponding encoded structure M consists of:

- $M^{\mathsf{Set}(\mathcal{U}\times\mathsf{Set}(\mathcal{U}))}$ containing only the constant R_0
- $M^{\mathcal{U}\times\mathsf{Set}(\mathcal{U})}$ consisting of the elements of R_0,
- $M^{\mathcal{U}}$ consisting of $\{a, b, c\}$
- $M^{\mathsf{Set}(\mathcal{U})}$ consisting of the sets $\{a, b\}$, $\{a, c\}$,
- $M^{\mathsf{Unit}} = \{()\}$ and $M^{\mathsf{Bool}} = \{\emptyset, \{()\}\}$
- the element relations interpreted in the natural way

Consider the transformation that groups on the first component, returning an output object of type $\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathsf{Set}(\mathcal{U})))$. This is a variation of the grouping transformation from Example 1.2 and Example 3.2. On the example input R_0 the transformation would return

$$\{\langle a, \{\{a, b\}, \{a, c\}\}\rangle, \langle b, \{\{a, c\}\}\rangle\}$$

The output would be represented by a structure having sorts $\mathsf{Set}(\mathcal{U}\times\mathsf{Set}(\mathsf{Set}(\mathcal{U}))), \mathcal{U}\times\mathsf{Set}(\mathsf{Set}(\mathcal{U})), \mathcal{U}, \mathsf{Set}(\mathsf{Set}(\mathcal{U}))$ and $\mathsf{Set}(\mathcal{U})$ in addition to Unit and Bool. It is easy to capture this transformation with a Δ_0 interpretation. For example, the interpretation could code the output sort $\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathsf{Set}(\mathcal{U})))$ as $\mathsf{Set}(\mathcal{U} \times \mathsf{Set}(\mathcal{U}))$, representing each group by the corresponding Ur-element. ◁

We will often make use of the following observation about interpretations:

Proposition 5.3. *Δ_0 interpretations can be composed, and their composition corresponds to the underlying composition of transformations.*

The composition of nested relational interpretations amounts to the usual composition of FO-interpretations (see e.g. [Benedikt and Koch 2009]) and an easy check that the additional requirements we impose on nested relational interpretations are preserved.

We can now state the equivalence of NRC and interpretations formally:

Theorem 5.4. *Every transformation in* NRC[Get] *can be translated effectively to a Δ_0 interpretation. Conversely, for every Δ_0 interpretation, one can effectively form an equivalent* NRC[Get] *expression. The translation from* NRC[Get] *to interpretations can be done in* EXPTIME *while the converse translation can be performed in* PTIME.

This characterization holds when equivalence is over finite nested relational inputs and also when arbitrary nested relations are allowed as inputs to the transformations.

From this theorem one can easily derive many of the "conservativity results"; e.g. [Paredaens and Van Gucht 1992], which states that every nested relational algebra query from flat type $(\mathsf{Set}(\mathcal{U}^n)$ to flat types can be expressed in relational algebra: we simply convert to an interpretation and then note that in going backward from an interpretation to an NRC expression we will not introduce additional levels of nesting on top of those present in the input and output.

Note that a number of very similar results occur in the literature. The underlying idea in one direction is that one can "shred" a transformation of collections to work on a flat representation. This has been investigated in several communities for NRC and related languages [Benedikt and Koch 2009; Cheney et al. 2014], in databases going at least as far back as [Abiteboul and Bidoit 1986]. The connection extends to richer collection types such as multi-sets, which have been the focus in using the shredding technique in systems [Cheney et al. 2014; Grust et al. 2010; Ulrich 2019]. Algorithms for shredding can also be useful as a technique for lifting optimizations, such as incremental query processing, from relational languages to nested languages [Koch et al. 2016]. And even in the

collection of richer collection types, many of the conservativity properties of NRC are maintained [Wong 1996]. But with these additional type-formers, one needs to move beyond first-order logic in the simulating language. Thus although they are still extremely relevant to implementation, reasoning with the resulting representations becomes problematic. The thesis [Ulrich 2019] provides a detailed look at shredding techniques, and also additional historical background.

Results of [Koch 2006] show that a PTIME translation of NRC expressions to interpretations would imply a collapse of the complexity class $TA[2^{O(n)}, n]$ to PSPACE, even at Boolean type. The early paper [Van den Bussche 2001] proves a translation of NRC similar to the one in the first half of Theorem 5.4 for flat-to-nested queries, and the nested-to-nested case can be easily obtained from this. However [Van den Bussche 2001] does not formalize the output of the interpretation as an interpretation, and we will need this connection to obtain our other characterizations. In the context of the XML query language XQuery, [Benedikt and Koch 2009] proves a transformation to first-order interpretations over trees. As noted in [Koch 2006], there is a very close relationship between XQuery and NRC, and the translation to interpretations in [Benedikt and Koch 2009] can be easily lifted to NRC.

There is also similarity to results from the 1960's of Gandy [Gandy 1974]. Gandy defines a class of set functions that are similar to NRC, and shows that they are "substitutable". This is the core of the argument for translating NRC to interpretations.

6 SYNTHESIZING INTERPRETATIONS FROM CLASSICAL PROOFS

In Section 4 we showed that from an intuitionistic proof that $\Sigma(\mathrm{o}_{in}, \ldots, \mathrm{o}_{out})$ defines o_{out} as a function of o_{in}, we could synthesize an NRC expression that produces o_{out} from o_{in}. One might believe such a "witnessing theorem" to be specific to intuitionistic calculi. But we will now demonstrate that this result extends to classical proofs, and that it is actually a general phenomenon connecting implicit definitions to interpretations. We will show that whenever we have a Δ_0 specification where there is a classical proof that the specification is functional, we can generate an interpretation that realizes the function. We can then rely on Theorem 5.4 from the previous section to infer that an NRC[Get] expression realizes the function as well. That is, we will prove:

Theorem 6.1. *For any Δ_0 formula $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ which implicitly defines o_{out} as a function of o_{in}, there is a Δ_0 interpretation $\mathcal{I}$ such that whenever $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ holds, then $\mathcal{I}(\mathrm{o}_{in}) = \mathrm{o}_{out}$.*

In particular, if in addition for each o_{in} there is some o_{out} and $\vec{a}$ such that $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ holds, then the interpretation and the formula define the same transformation.

Recall from Section 4 that projective implicit definitions allow extra parameters $\vec{a}$ while implicit definitions allow only the input and output variables o_{in} and o_{out}. From Theorem 6.1 we easily get the following characterization:

Corollary 6.2. *The following are equivalent for a transformation $\mathcal{T}$:*

- *$\mathcal{T}$ is projectively implicitly definable by a Δ_0 formula*
- *$\mathcal{T}$ is implicitly definable by a Δ_0 formula*
- *$\mathcal{T}$ is definable via a Δ_0 interpretation*
- *$\mathcal{T}$ is* NRC[Get] *definable*

Finite instances versus all instances. In Theorem 6.1 and Corollary 6.2 we emphasize that our results concern the class $\mathsf{Fun}_{\mathsf{All}}$ of transformations $\mathcal{T}$ such that there is a Δ_0 formula Σ which defines a functional relationship between o_{in} and o_{out} on all instances, finite and infinite, and where the function agrees with $\mathcal{T}$. We can consider $\mathsf{Fun}_{\mathsf{All}}$ as a class of transformations on all instances or of finite instances, but the class is defined by reference to all instances for o_{in}. Expressed semantically

$$\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a}) \wedge \Sigma(\mathrm{o}_{in}, \mathrm{o}'_{out}, \vec{a}') \models \mathrm{o}'_{out} = \mathrm{o}_{out}$$

An equivalent characterization of $\mathsf{Fun}_{\mathsf{All}}$ is *proof-theoretic*: these are the transformations such that there is a classical proof of functionality in a complete first-order proof system using some basic axioms about Ur-elements, products and projection functions, and the extensionality axiom for the membership relation. For example, it is easy to extend the intuitionistic proof system given in Section 4 to be complete for classical entailment.

Whether one thinks of $\mathsf{Fun}_{\mathsf{All}}$ semantically or proof-theoretically, our results say that $\mathsf{Fun}_{\mathsf{All}}$ is identical with the set of transformations given by NRC expressions. But the proof-theoretic perspective is crucial for the synthesis procedure.

It is natural to ask about the analogous class $\mathsf{Fun}_{\mathsf{Fin}}$ of transformations $\mathcal{T}$ over *finite inputs* for which there is a Δ_0 $\Sigma_{\mathcal{T}}$ which is functional, when only finite inputs are considered, and where the corresponding function agrees with $\mathcal{T}$. It is well-known that $\mathsf{Fun}_{\mathsf{Fin}}$ is not identical to NRC and is not so well-behaved. The transformation returning the powerset of a given input relation o_{in} is in $\mathsf{Fun}_{\mathsf{Fin}}$: the powerset of a finite input o_{in} is the unique collection o_{out} of subsets of o_{in} that contains the empty set and such that for each element e of o_{in}, if a set s is in o_{out} then $s - \{e\}$ and $s \cup \{e\}$ are in o_{out}. From this we can see that $\mathsf{Fun}_{\mathsf{Fin}}$ contains transformations of high complexity. Indeed, even when considering transformations from flat relations to flat relations, $\mathsf{Fun}_{\mathsf{Fin}}$ contains transformations whose membership in polynomial time would imply that UP $\cap$ coUP, the class of problems such that both the problem and its complement can be solved by an unambiguous non-deterministic polynomial time machine, is identical to PTIME [Kolaitis 1990]. Most importantly for our goals, membership in $\mathsf{Fun}_{\mathsf{Fin}}$ is *not* witnessed by proofs in any effective proof system, since this set is not computably enumerable.

Total versus partial functions. When we have a proof that $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ defines o_{out} as a function of o_{in}, the corresponding function may still be partial. Our procedure will synthesize an expression E defining a total function that agrees with the partial function defined by Σ. If $\vec{a}$ is empty, we can also synthesize a Boolean NRC expression $\mathsf{Verify}_{\mathsf{InDomain}}$ that verifies whether a given o_{in} is in the domain of the function: that is whether there is o_{out} such that $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out})$ holds. $\mathsf{Verify}_{\mathsf{InDomain}}$ can be taken as:

$$\bigcup \{\mathsf{Verify}_{\Sigma}(\mathrm{o}_{in}, e) \mid e \in \{E(\mathrm{o}_{in})\}\}$$

where Verify_{Σ} is from Proposition 3.3.

Recall the second transformation from Example 1.2, where the domain of the function is the set of G such that the second component of each pair is never empty and the value of the second component is determined by the value of the first component. This property can clearly be described by a Δ_0 formula, and thus by Proposition 3.3 it can be verified in NRC.

When $\vec{a}$ is not empty we cannot generate a domain check, since the auxiliary parameters might enforce some second-order property of o_{in}: for example $\Sigma(\mathrm{o}_{in}, a, o)$ might state that a is a bijection from $\pi_1(\mathrm{o}_{in})$ to $\pi_2(\mathrm{o}_{in})$ and $o = \mathrm{o}_{in}$. This clearly defines a functional relationship between i_0, i_1 and o, but the domain consists of i_0, i_1 that have the same cardinality, which cannot be expressed in first-order logic.

Organization of the proof of the theorem. Our proof of Theorem 6.1 will proceed first by some reductions (Subsection 6.1), showing that it suffices to prove a general result about implicit definability and definability by interpretations in multi-sorted first-order logic, rather than dealing with higher-order logic and Δ_0 formulas. In Subsection 6.2 we sketch the argument for this multi-sorted logic theorem.

6.1 Reduction to a Characterization Theorem in Multi-Sorted Logic

The first step in the proof of Theorem 6.1 is to reduce to a more general statement relating implicit definitions in multi-sorted logic to interpretations. The first part of this reduction is to argue that we can suppress auxiliary parameters $\vec{a}$ in implicit definitions:

Lemma 6.3. *For any Δ_0 formula $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a})$ that implicitly defines o_{out} as a function of o_{in}, there is another Δ_0 formula $\Sigma'(\mathrm{o}_{in}, \mathrm{o}_{out})$ which implicitly defines o_{out} as a function of o_{in}, such that $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out}, \vec{a}) \Rightarrow \Sigma'(\mathrm{o}_{in}, \mathrm{o}_{out})$.*

The lemma is proven using two applications of *classical* Δ_0 interpolation.

Proposition 6.4. *For any Δ_0 formulas φ and ψ such that $\varphi \models \psi$, there exists another Δ_0 formula θ such that $\varphi \models \theta$ and $\theta \models \psi$.*

This proposition generalizes Proposition 4.6 since we allow classical validity for $\varphi \models \psi$. That being said, we may prove Proposition 6.4 using similar tools, i.e., a complete cut-free sequent calculus for Δ_0 formulas and a standard proof as in [Fitting 1996]. With Lemma 6.3 in hand, from this point on we assume that we do not have auxiliary parameters $\vec{a}$ in our implicit definitions.

Reduction to Monadic schemas. A *monadic type* is a type built only using the atomic type $\mathcal{U}$ and the type constructor Set. To simplify notation we define $\mathcal{U}_0 := \mathcal{U}$, $\mathcal{U}_1 := \mathsf{Set}(\mathcal{U}_0), \ldots \mathcal{U}_{n+1} := \mathsf{Set}(\mathcal{U}_n)$. A monadic type is thus a $\mathcal{U}_n$ for some $n \in \mathbb{N}$. A nested relational schema is monadic if it contains only monadic types, and a Δ_0 formula is monadic if all of its variables have monadic types.

Restricting to monadic formulas simplifies the type system significantly and thus, certain arguments by induction. It turns out that by the usual "Kuratowski encoding" of pairs by sets, we can reduce all of our questions about implicit versus explicit definability to the case of monadic schemas. The following proposition implies that we can derive all of our main results for arbitrary schemas from their restriction to monadic formulas. We will thus restrict to monadic formulas for the remainder of the argument.

Proposition 6.5. *For any nested relational schema $\mathcal{SCH}$, there is a monadic nested relational schema $\mathcal{SCH}'$, an injection* Convert *from instances of $\mathcal{SCH}$ to instances of $\mathcal{SCH}'$ that is definable in* NRC*, and an* NRC[Get] *expression* $\mathsf{Convert}^{-1}$ *such that* $\mathsf{Convert}^{-1} \circ \mathsf{Convert}$ *is the identity transformation from $\mathcal{SCH} \to \mathcal{SCH}$.*

Furthermore, there is a Δ_0 formula $\mathsf{Im}_{\mathsf{Convert}}$ *from $\mathcal{SCH}'$ to* Bool *such that* $\mathsf{Im}_{\mathsf{Convert}}(i')$ *holds if and only if $i' = \mathsf{Convert}(i)$ for some instance i of $\mathcal{SCH}$.*

These translations can also be given in terms of Δ_0 interpretations rather than NRC *expressions.*

Given Proposition 6.5 it suffices to consider only monadic nested relational schemas. Given a Δ_0 implicit definition $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out})$ we can form a new definition that computes the composition of the following transformations: $\mathsf{Convert}^{-1}_{\mathcal{SCH}_{in}}$, a projection onto the first component, the transformation defined by Σ, and $\mathsf{Convert}_{\mathcal{SCH}_{out}}$. Our new definition captures this composition by a formula $\Sigma'(\mathrm{o}'_{in}, \mathrm{o}'_{out})$ that defines o'_{out} as a function of o'_{in}, where the formula is over a monadic schema. Assuming that we have proven the theorem in the monadic case, we would get an NRC expression E' from $\mathcal{SCH}'_{in}$ to $\mathcal{SCH}'_{out}$ agreeing with this formula on its domain. Now we can compose $\mathsf{Convert}_{\mathcal{SCH}_{in}}$, E', $\mathsf{Convert}^{-1}_{\mathcal{SCH}_{out}}$, and the projection to get an NRC expression agreeing with the partial function defined by $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out})$ on its domain, as required.

Reduction to a result in multi-sorted logic. Now we are ready to give our last reduction, relating Theorem 6.1 to a general result concerning multi-sorted logic.

Let $\mathcal{SIG}$ be any multi-sorted signature, Sorts_1 be its sorts and Sorts_0 be a subset of Sorts_1. We say that a relation R is *over* Sorts_0 if all of its arguments are in Sorts_0. Let Σ be a set of sentences in

$\mathcal{SIG}$. Given a model M for $\mathcal{SIG}$, let $\mathrm{Sorts}_0(M)$ be the union of the domains of relations over Sorts_0, and let $\mathrm{Sorts}_1(M)$ be defined similarly.

We say that Sorts_1 is *implicitly interpretable* over Sorts_0 relative to Σ if:

For any models M_1 and M_2 of Σ, if there is a mapping m from $\mathrm{Sorts}_0(M_1)$ to $\mathrm{Sorts}_0(M_2)$ that preserves all relations over Sorts_0, then m extends to a unique mapping from $\mathrm{Sorts}_1(M_1)$ to $\mathrm{Sorts}_1(M_2)$ which preserves all relations over Sorts_1.

Informally, implicit interpretability states that the sorts in Sorts_1 are semantically determined by the sorts in Sorts_0. The property implies in particular that if M_1 and M_2 agree on the interpretation of sorts in Sorts_0, then the identity mapping on sorts in Sorts_0 extends to a mapping that preserves sorts in Sorts_1.

We relate this semantic property to a syntactic one. We say that Sorts_1 is *explicitly interpretable* over Sorts_0 relative to Σ if for all S in Sorts_1 there is a formula $\psi_S(\vec{x}, y)$ where $\vec{x}$ are variables with sorts in Sorts_0, y a variable of sort Sorts_1, such that:

- In any model M of Σ, ψ_S defines a partial function F_S mapping Sorts_0 tuples on to S.
- For every relation R of arity n over Sorts_1, there is a formula $\psi_R(\vec{x}_1, \ldots \vec{x}_n)$ using only relations over Sorts_0 and only quantification over Sorts_0 such that in any model M of Σ, the pre-image of R under the mappings F_S for the different arguments of R is defined by $\psi_R(\vec{x}_1, \ldots \vec{x}_n)$.

Explicit interpretability states that there is an interpretation in the sense of the previous section that produces the structure in Sorts_1 from the structure in Sorts_0, and in addition there is a definable relationship between an element e of a sort in Sorts_1 and the tuple that codes e in the interpretation. Note that ψ_S, the mapping between the elements y in S and the tuples in Sorts_0 that interpret them, can use arbitrary relations. The key property is that when we pull a relation R over Sorts_1 back using the mappings ψ_S, then we obtain something definable using Sorts_0.

With these definitions in hand, we are ready to state a result in multi-sorted logic which allows us to generate interpretations from classical proofs of functionality:

Theorem 6.6. *For any* $\Sigma, \mathrm{Sorts}_0, \mathrm{Sorts}_1$ *such that* Σ *entails that a sort of* Sorts_0 *has at least two elements,* Sorts_1 *is explicitly interpretable over* Sorts_0 *if and only if it is implicitly interpretable over* Sorts_0.

This can be thought of as an analog of Beth's theorem [Beth 1953; Craig 1957] for multi-sorted logic. The proof is sketched in the next subsection. For now we explain how it implies Theorem 6.1. In this explanation we assume a monadic schema for both input and output. Thus every element e in an instance has sort $\mathcal{U}_n$ for some $n \in \mathbb{N}$.

Consider a Δ_0 formula $\Sigma(\mathrm{o}_{in}, \mathrm{o}_{out})$ over a monadic schema that implicitly defines o_{out} as a function of o_{in}. Σ can be considered as a multi-sorted first-order formula with sorts for every subtype occurrence of the input as well as distinct sorts for every subtype occurrence of the output other than $\mathcal{U}$. Because we are dealing with monadic input and output schema, every sort other than $\mathcal{U}$ will be of the form $\mathrm{Set}(T)$, and these sorts have only the element relations $\in_T$ connecting them. We refer to these as *input sorts* and *output sorts*. We modify Σ by asserting that all elements of the input sorts lie underneath o_{in}, and all elements of the output sorts lie underneath o_{out}, where an element e is said to lies underneath an element e' if there is a chain $e = e_1 \in \ldots \in e_n = e'$. Since Σ was Δ_0, this does not change the semantics. We also conjoin to Σ the sanity axioms for the schema, including the extensionality axiom at the sorts corresponding to each object type. Let Σ^* be the resulting formula. In this transformation, as was the case with interpretations, we change our perspective on inputs and outputs, considering them as constants rather than as free variables. We do this only to match our result in multi-sorted logic, which deals with a set of *sentences* in multi-sorted first-order logic, rather than formulas with free variables.

Given models M and M' of Σ^*, we define relations $\equiv_i$ connecting elements of M of depth i with elements of M' of depth i. For $i = 0$, $\equiv_i$ is the identity: that is, it connects elements of $\mathcal{U}$ if and only if they are identical. For $i = j + 1$, $\equiv_i$ (x, x') holds exactly when for every $y \in x$ there is $y' \in x'$ such that $y \equiv_j y'$, and vice versa.

The fact that Σ implicitly defines o_{out} as a function of o_{in} tells us that:

Suppose $M \models \Sigma^*$, $M' \models \Sigma^*$ and M and M' are identical on the input sorts. Then the mapping m taking a $y \in M$ of depth i to a $y' \in M'$ such that $y' \equiv_i y$ is an isomorphism of the output sorts that is the identity on $\mathcal{U}$. Further, any isomorphism of $\mathrm{Sorts}_1(M)$ on to $\mathrm{Sorts}_1(M')$ that is the identity on $\mathcal{U}$ must be equal to m: one can show this by induction on the depth i using the fact that Σ^* includes the extensionality axiom.

From this, we see that the output sorts are implicitly interpretable over the input sorts relative to Σ^*. Using Theorem 6.6, we conclude that the output sorts are explicitly interpretable in the input sorts relative to Σ^*. Applying the conclusion to the formula $x = x$, where x is a variable of a sort corresponding to object type T of the output, we obtain a first-order formula $\varphi^T_{\mathsf{Domain}}(\vec{x})$ over the input sorts. Applying the conclusion to the formula $x = y$ for x, y variables corresponding to the object type T we get a formula $\varphi_{\equiv_T}(\vec{x}, \vec{x}')$ over the input sorts. Finally applying the conclusion to the element relation ϵ_T at every level of the output, we get a first-order formula $\varphi_{\epsilon_T}(\vec{x}, \vec{x}')$ over the input sorts. Because Σ^* asserts that each element of the input sorts lies beneath a constant for o_{in}, we can convert all quantifiers to bind only beneath o_{in}, giving us Δ_0 formulas. It is easy to verify that these formulas give us the desired interpretation. This completes the proof of Theorem 6.1, assuming Theorem 6.6.

6.2 Proof of the Multi-Sorted Logic Result

In the previous subsection we reduced our goal result about generating interpretations from proofs to a result in multi-sorted first-order logic, Theorem 6.6. We will sketch the proof of Theorem 6.6. The direction from explicit interpretability to implicit interpretability is straightforward, so we will be interested only in the direction from implicit to explicit. Although the theorem appears to be new, each of the components is a variant of arguments that already appear in the model theory literature.

In the body of the paper we make use of only quite basic results from model theory:

- the *compactness theorem* for first-order logic, which states that for any theory Γ, if every finite subcollection of Γ is satisfiable, then Γ is satisfiable;
- the *downward Lowenheim-Skolem theorem*, which states that if Γ is countable and has a model, then it has a countable model;
- the *omitting types theorem* for first-order logic. A first-order theory Σ is said to be *complete* if for every other first-order sentence φ in the vocabulary of Σ, either φ or $\neg\varphi$ is entailed by Σ. Given a set of constants B, a *type* over B is an infinite collection $\tau(\vec{x})$ of formulas using variables $\vec{x}$ and constants B. A type is *complete* with respect to a theory Σ if every first-order formula with variables in $\vec{x}$ and constants from B is either entailed or contradicted by $\tau(\vec{x})$ and Σ. A type τ is said to be *realized* in a model M if there is a $\vec{x}_0$ in M satisfying all formulas in τ. τ is *non-principal* (with respect to a first-order theory Σ) if there is no formula $\gamma_0(\vec{x})$ such that $\Sigma \wedge \gamma_0(\vec{x})$ entails all of $\tau(\vec{x})$. The version of the omitting types theorem that we will use states that:

  if we have a countable set Γ of complete types that are all non-principal relative to a complete theory Σ, there is some model M of Σ in which none of the types in Γ are realized.

Each of these results follows from a standard model construction technique [Hodges 1993].

We can easily show that to prove the multi-sorted result, it suffices to consider Σ that is a complete theory.

Proposition 6.7. *Theorem 6.6 follows from its restriction to Σ a complete theory.*

Recall that our assumption is that Σ yields a function from o_{in} to o_{out}. Our next step will be to show that the output of this function is always "sub-definable": each element in the output is definable from the input if we allow ourselves to guess some parameters. For example, consider the grouping transformation mentioned in Example 1.2 and Example 3.2. Each output is obtained from grouping input relation F over some Ur-element a. So each member of the output is definable from the input constant F and a "guessed" input element a. We will show that this is true in general.

Given a model M of Σ and $\vec{x}_0 \in \mathrm{Sorts}_1$ within M, the *type of $\vec{x}_0$ with parameters from* Sorts_0 is the set of all formulas satisfied by $\vec{x}_0$, using any sorts and relations but only constants from Sorts_0.

A type p is *isolated over* Sorts_0 if there is a formula $\varphi(\vec{x}, \vec{a})$ with parameters $\vec{a}$ from Sorts_0 such that $M \models \varphi(\vec{x}, \vec{a}) \rightarrow \gamma(\vec{x})$ for each $\gamma \in p$. The following is a step towards showing that elements in the output are well-behaved:

Lemma 6.8. *Suppose* Sorts_1 *is implicitly interpretable over* S_0 *with respect to* Σ*. Then in any model M of Σ the type of any $\vec{b}$ over* Sorts_1 *with parameters from* Sorts_0 *is isolated over* Sorts_0.

Proof. Fix a counterexample $\vec{b}$, and let Γ be the set of formulas in Sorts_1 with constants from Sorts_0 satisfied by $\vec{b}$ in M. We claim that there is a model M' with $\mathrm{Sorts}_0(M')$ identical to $\mathrm{Sorts}_0(M)$ where there is no tuple satisfying Γ. This follows from the failure of isolation and the omitting types theorem.

Now we have a contradiction of implicit interpretability, since the identity mapping on Sorts_0 cannot extend to an isomorphism of relations over Sorts_1 from M to M'. □

The next step is to argue that every element of Sorts_1 is definable by a formula using parameters from Sorts_0.

Lemma 6.9. *Assume implicit interpretability of* Sorts_1 *over* Sorts_0 *relative to Σ. In any model M of Σ, for every element e of a sort* S_1 *in* Sorts_1*, there is a first-order formula $\psi_e(\vec{y}, x)$ with variables $\vec{y}$ having sort in* Sorts_0 *and x a variable of sort* S_1*, along with a tuple $\vec{a}$ in* $\mathrm{Sorts}_0(M)$ *such that $\psi_e(\vec{a}, x)$ is satisfied only by e in M.*

Proof. Since a counterexample involves only formulas in a countable language, thanks to the Lowenheim-Skolem theorem mentioned above, it is enough to consider the case where M is countable. By Lemma 6.8, the type of every e is isolated by a formula $\varphi(\vec{x}, \vec{a})$ with parameters from Sorts_0 and relations from Sorts_1. We claim that φ defines e: that is, e is the only satisfier. If not, then there is $e' \neq e$ that satisfies φ. Consider the relation $\vec{e} \equiv \vec{e}'$ holding if $\vec{e}$ and $\vec{e}'$ satisfy all the same formulas using relations and variables from Sorts_1 and parameters from Sorts_0. Isolation implies that $e \equiv e'$. Further, isolation of types shows that $\equiv$ has the "back-and-forth property" given $\vec{d} \equiv \vec{d}'$, and $\vec{e}$ we can obtain $\vec{e}'$ with $\vec{d}\vec{e} \equiv \vec{d}'\vec{e}'$. To see this, fix $\vec{d} \equiv \vec{d}'$ and consider $\vec{e}$. We have $\gamma(\vec{x}, \vec{y}, \vec{a})$ isolating the type of $\vec{d}, \vec{e}$, and further $\vec{d}$ satisfies $\exists \vec{y}\ \gamma(\vec{x}, \vec{y}, \vec{a})$ and thus so does $\vec{d}'$ with witness $\vec{e}'$. But then using $\vec{d} \equiv \vec{d}'$ again we see that $\vec{d}, \vec{e} \equiv \vec{d}', \vec{e}'$. Using countability of M and this property we can inductively create a mapping on M fixing Sorts_0 pointwise, preserving all relations in Sorts_1, and taking $\vec{b}$ to $\vec{b}'$. But this contradicts implicit interpretability. □

Lemma 6.10. *The formula in Lemma 6.9 can be taken to depend only on the sort* S.

Proof. Consider the type over the single variable x of sort S consisting of the formulas $\neg\delta_\varphi(x)$, taking $\delta_\varphi(x)$ to be defined as

$$\exists\vec{b}\ [\ \varphi(\vec{b},x) \wedge \forall x'\ (\varphi(\vec{b},x') \Rightarrow x' = x)]$$

where the tuple $\vec{b}$ ranges over Sorts_0. By Lemma 6.9, this type cannot be satisfied in a model of Σ. Since it is unsatisfiable, by compactness, there are finitely many formulas $\varphi_1(\vec{b},x),\ldots,\varphi_n(\vec{b},x)$ such that $\forall x\ \bigvee_{i=1}^{n}\delta_{\varphi_i}(x)$ is satisfied. Therefore, each $\varphi_i(\vec{b},x)$ defines a partial function from tuples of S_0 to S and every element of S is covered by one of the φ_i. Recall that we assumed that Σ enforces that Sorts_0 has a sort with at least two elements. Thus we can combine the $\varphi_i(\vec{b},x)$ into a single formula $\psi(\vec{b},\vec{c},x)$ defining a surjective partial function from S_0 to S where $\vec{c}$ is an additional parameter in Sorts_0 selecting some $i \leq n$. □

We now need to go from the "sub-definability" or "element-wise definability" result above to an interpretation. Consider the formulas ψ_S produced by Lemma 6.10. For a relation R of arity n over Sorts_1, where the i^{th} argument has sort S_i, consider the formula

$$\psi_R(\vec{x}_1\ldots\vec{x}_n)\ =\ \exists y_1\ldots y_n\ R(y_1\ldots y_n) \wedge \bigwedge_i \psi_{\text{S}_i}(\vec{x}_i,y_i)$$

where $\vec{x}_i$ is a tuple of variables of sorts in Sorts_0. The formulas ψ_S for each sort S and the formulas ψ_R for each relation R are as required by the definition of explicitly interpretable, except that they may use quantified variables and relations of Sorts_1, while we only want to use variables and relations from Sorts_0. We take care of this in the following lemma, which says that formulas over Sorts_1 do not allow us to define any more subsets of Sorts_0 than we can with formulas over Sorts_0.

Lemma 6.11. *Under the assumption of implicit interpretability, for every formula $\varphi(\vec{x})$ over Sorts_1 with $\vec{x}$ variables of sort in Sorts_0 there is a formula $\varphi^\circ(\vec{x})$ over Sorts_0 – that is, containing only variables, constants, and relations from Sorts_0 – such that for every model M of Σ,*

$$M \models \forall\vec{x}\ \varphi(\vec{x}) \leftrightarrow \varphi^\circ(\vec{x})$$

Proof. Assume not, with φ as a counterexample. By the compactness and Lowenheim-Skolem theorems, we know that there is a countable model M of Σ containing $\vec{c}$, $\vec{c}'$ that agree on all formulas in Sorts_0 but that disagree on φ. As in Lemma 6.9, we can obtain a mapping on M preserving Sorts_0 but sending $\vec{c}$ to $\vec{c}'$. This contradicts implicit interpretability, since the mapping cannot be extended. □

Above we obtained the formulas ψ_R for each relation symbol R needed for an explicit interpretation. We can obtain formulas defining the necessary equivalence relations $\psi_\equiv$ and ψ_Domain easily from these. Thus, putting Lemmas 6.9, 6.10, and 6.11 together yields a proof of Theorem 6.6.

6.3 Putting It All Together

We summarize our results on extracting NRC[Get] expressions from classical proofs of functionality. We have shown in Subsection 6.1 how to convert the problem to one with no extra variables other than input and output and with only monadic schemas – and thus no use of products or tupling. We also showed how to convert the resulting formula into a theory in multi-sorted first-order logic. That is, we no longer need to talk about Δ_0 formulas.

In Subsection 6.2 we showed that from a theory in multi-sorted first-order logic we can obtain an interpretation. This first-order interpretation in a multi-sorted logic can then be converted back to a Δ_0 interpretation, since the background theory forces each of the input sorts in the multi-sorted structure to correspond to a level of nesting below one of the constants corresponding to an input

object. Finally, the results of Section 5 allow us to convert this interpretation to an NRC[Get] expression. With the exception of the result in multi-sorted logic, all of the constructions are effective. Further, these effective conversions are all in polynomial time except for the transformation from an interpretation to an NRC[Get] expression, which is exponential time in the worst case. Outside of the multi-sorted result, which makes use of infinitary methods, the conversions are each sound when equivalence over finite input structures is considered as well as the default case when arbitrary inputs are considered. As explained in Subsection 6.1, when equivalence over finite inputs is considered, we cannot hope to get a synthesis result of this kind.

7 CONCLUSION

We have provided a method taking a proof that a logical formula defines a functional transformation and generating an expression in a functional transformation language that implements it. In the process we provide a more general synthesis procedure (Lemma 4.7) that can generate expressions interpolating between variables whenever there is a provable containment. This connection between provably functional formulas and the functional transformation language NRC studied in data management and programming languages is, to our knowledge, new and non-trivial.

We are currently working on an implementation of our effective synthesis result in the COQ proof assistant [Coq 2020]. This involves formalizing the proof calculus, the semantics of Δ_0 formulas, the syntax and semantics of NRC, in COQ, as well as the synthesis algorithm. In addition to giving us a verified proof, we will gain the ability to create proofs of functionality within a COQ session, allowing us to build up tactics and definitions on top of the basic rules of the proof calculus.

An open issue is to make the classical interpolation result effective. There is an obvious extension of our proof system that gains completeness for classical logic: we allow multiple disjuncts in the consequence, and revise the rules in the obvious way. For instance, the rule $\in_{\mathrm{Set}(T)}$-R would become

$$\in_{\mathrm{Set}}\text{-R}\ \frac{\Theta,\ t \in_{\mathrm{Set}(T)} v;\ \Gamma \vdash t =_{\mathrm{Set}(T)} u,\ t_1 \in_{T_1} u_1, \ldots,\ t_k \in_{T_k} u_k}{\Theta,\ t \in_{\mathrm{Set}(T)} v;\ \Gamma \vdash u \in_{\mathrm{Set}(T)} v,\ t_1 \in_{T_1} u_1, \ldots,\ t_k \in_{T_k} u_k}$$

Theorem 6.1 shows that when we have a proof in such a system we can create an NRC definition, and we conjecture that it is possible to do this efficiently. In fact, we can also show that the higher-type interpolation lemma, Lemma 4.7, holds for classical entailment. Although our proof of Lemma 4.7 is via induction on proofs, the extension for classical entailment can be done using model-theoretic techniques, in particular a dichotomy theorem for automorphisms stemming from work of Makkai [Makkai 1964]. We are investigating an extension of our proof system that will allow us to lift our current inductive argument for Lemma 4.7 to the classical setting. We conjecture that it will lead us to an efficient procedure for extracting NRC terms from classical functionality proofs, thereby simultaneously generalizing Theorem 4.2 and Theorem 6.1.

In addition to the application areas exhibited in Examples 4.4 and 4.5, we think that procedures for generating implementations in functional languages from implicit definitions should have other applications in programming languages and verification. For example they could be relevant for generating programs transforming structured data in the context of more specialized input structures, such as strings and trees [Bojanczyk et al. 2018].

We focused here on a stripped-down setting where at the base level we have no additional structure, but many of our results (e.g. Theorem 6.1) generalize in the presence of additional axiomatizable structure on the base set. Another important direction is to generalize the algorithmic development (e.g. Theorem 4.2) to incorporate specialized decision procedures available on this additional structure.

ACKNOWLEDGMENTS

We are very grateful to Szymon Toruńczyk, who outlined a route to show that implicitly definable transformations over nested relations can be defined via interpretations, in the process conjecturing a more general result concerning definability in multi-sorted logic. Szymon also helped in simplifying the mapping of NRC expressions to interpretations, a basic component in one of our characterizations. We also thank Ehud Hrushovski, who sketched a proof of the Beth-style result for multi-sorted logic that serves as another component. His proof proceeds along very similar lines to the one we present in this paper, but makes use of a prior Beth-style result in classical model theory [Makkai 1964]. This work was funded by EPSRC grant EP/M005852/1.

REFERENCES

Serge Abiteboul and Catriel Beeri. 1995. The Power of Languages for the Manipulation of Complex Values. *VLDB J.* 4, 4 (1995), 727–794.

Serge Abiteboul and Nicole Bidoit. 1986. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *J. Comput. Syst. Sci.* 33, 3 (1986), 361–393.

Foto Afrati and Rada Chirkova. 2019. *Answering Queries Using Views.* Morgan & Claypool Publishers.

H. Andréka, J. X. Madarász, and I. Németi. 2008. Definability of New Universes in Many-sorted logic. (2008). manuscript available at old.renyi.hu/pub/algebraic-logic/kurzus10/amn-defi.pdf.

Michael Benedikt, Balden Ten Cate, Julien Leblay, and Efthymia Tsamoura. 2016. *Generating plans from proofs: the interpolation-based approach to query reformulation.* Morgan Claypool.

Michael Benedikt and Christoph Koch. 2009. From XQuery to Relational Logics. *ACM TODS* 34, 4 (2009), 25:1–25:48.

E. W. Beth. 1953. On Padoa's Method in the Theory of Definitions. *Indagationes Mathematicae* 15 (1953), 330 – 339.

Mikolaj Bojanczyk, Laure Daviaud, and Shankara Narayanan Krishna. 2018. Regular and First-Order List Functions. In *LICS*.

Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48.

James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*.

Thomas Colcombet and Christof Löding. 2007. Transforming structures by set interpretations. *Logical Methods in Computer Science* 3, 2 (2007).

Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *DBPL*.

Coq. 2020. The Coq Proof Assistant. (2020). coq.inria.fr.

William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Logic* 22, 3 (1957), 269–285.

Melvin Fitting. 1996. *First-order Logic and Automated Theorem Proving.* Springer.

R. O. Gandy. 1974. Set-theoretic functions for elementary syntax. In *Proceedings of Symposia in Pure Mathematics, 13, Part II*, Thomas Jech (Ed.). American Mathematical Society, 103–126.

Jeremy Gibbons. 2016. Comprehending Ringads - For Phil Wadler, on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*.

Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. 2018. Relational algebra by way of adjunctions. *PACMPL* 2, ICFP (2018).

Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. *PVLDB* 3, 1–2 (2010), 162––172.

Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB Journal* 10, 4 (2001), 270–294.

Kryštof Hoder, Laura Kovács, and Andrei Voronkov. 2010. *Interpolation and Symbol Elimination in Vampire.*

Wilfrid Hodges. 1993. *Model Theory.* Cambridge University Press.

Wilfrid Hodges, I.M. Hodkinson, and Dugald Macpherson. 1990. Omega-categoricity, relative categoricity and coordinatisation. *Annals of Pure and Applied Logic* 46, 2 (1990), 169 – 199.

Qinheping Hu and Loris D'Antoni. 2017. Automatic Program Inversion Using Symbolic Transducers. In *PLDI*.

Bart Jacobs. 2001. *Categorical Logic and Type Theory.* Elsevier.

R. B. Jensen. 1972. The fine structure of the constructible hierarchy, with a section by Jack Silver. *Annals of Mathematical Logic* 4 (1972), 229–308.

Christoph Koch. 2006. On the Complexity of Non-recursive XQuery and Functional Query Languages on Complex Values. *ACM TODS* **31**, 4 (2006), 1215–1256.

Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *PODS*.

Phokion G. Kolaitis. 1990. Implicit Definability on Finite Structures and Unambiguous Computations. In *LICS*.

Maurizio Lenzerini. 2002. Data Integration: A Theoretical Perspective. In *PODS*.

M. Makkai. 1964. On a generalization of a theorem of E. W. Beth. *Acta Mathematica Academiae Scientiarum Hungaricae* 15 (1964), 227–235.

K.L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *CAV*.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1-2 (2010), 330–339.

Andrzej Mostowski. 1949. An undecidable arithmetical statement. *Fundamenta Mathematicae* 36, 1 (1949), 143–164.

Alan Nash, Luc Segoufin, and Victor Vianu. 2010. Views and queries: Determinacy and rewriting. *ACM TODS* 35, 3 (2010).

Martin Otto. 2000. An interpolation theorem. *Bulletin of Symbolic Logic* 6, 4 (2000), 447–462.

Jan Paredaens and Dirk Van Gucht. 1992. Converting Nested Algebra Expressions into Flat Algebra Expressions. *ACM TODS* 17, 1 (1992), 65–93.

Vladimir Yu. Sazonov. 1985. Collection principle and existential quantifier. *Vychislitel'nye sistemy* 107 (1985), 30–39.

Luc Segoufin and Victor Vianu. 2005. Views and queries: determinacy and rewriting. In *PODS*.

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-Based Inductive Synthesis for Program Inversion. In *PLDI*.

Dan Suciu. 1995. *Parallel Programming Languages for Collections*. Ph.D. Dissertation. Univ. Pennsylvania.

David Toman and Grant Weddell. 2011. *Fundamentals of Physical Design and Query Compilation*. Morgan Claypool.

Alexander Ulrich. 2019. *Query Flattening and the Nested Data Parallelism Paradigm*. Ph.D. Dissertation. University of Tübingen, Germany. https://publikationen.uni-tuebingen.de/xmlui/handle/10900/87698/

Jan Van den Bussche. 2001. Simulation of the Nested Relational Algebra by the Flat Relational Algebra, with an Application to the Complexity of Evaluating Powerset Algebra Expressions. *Theoretical Computer Science* **254**, 1–2 (2001), 363–377.

Limsoon Wong. 1994. *Querying Nested Collections*. Ph.D. Dissertation. Univ. Pennsylvania.

Limsoon Wong. 1996. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. *J. Comput. Syst. Sci.* 52, 3 (1996), 495–505.