SMT Solving for Functional Programming over Infinite Structures*

Bartek Klin Michał Szynwelski
University of Warsaw
{klin,szynwelski}@mimuw.edu.pl

We develop a simple functional programming language aimed at manipulating infinite, but first-order definable structures, such as the countably infinite clique graph or the set of all intervals with rational endpoints. Internally, such sets are represented by logical formulas that define them, and an external satisfiability modulo theories (SMT) solver is regularly run by the interpreter to check their basic properties.

The language is implemented as a Haskell module.

1 Introduction

A common theme in computer science is effective manipulation of infinite but finitely presented data structures. It is one of the main features of functional programming, where computable functions, themselves infinite set-theoretic objects, are *bona fide* data values. In lazy programming languages such as Haskell one can also conveniently manipulate structures such as infinite lists or trees.

To achieve computability one usually restricts the interface used to manipulate infinite structures to a few basic and well-behaved operations. For example, the only way to access a function type data value is to apply it to an argument. Similarly, infinite lists provide a limited interface that allows only continuous operations on them to be implemented.

In mathematics a rich source of infinite but finitely presented objects are relational structures that are first-order definable over fixed, well understood structures. Examples include the set of ordered triples of natural numbers:

$$\big\{(a,b,c) \mid a,b,c \in \mathbb{N}\big\}, \tag{1}$$

or the infinite clique graph, with natural numbers as vertices and unordered pairs of distinct numbers as edges:

$$\big(\mathbb{N}, \big\{\{a,b\} \mid a,b \in \mathbb{N},\ a \neq b\big\}\big). \tag{2}$$

These structures are first-order defined over the set $\mathbb{N}$ of natural numbers with equality. On the other hand, the set of all closed intervals with rational endpoints:

$$\big\{\{c \mid c \in \mathbb{Q},\ a \leq c \leq b\} \mid a,b \in \mathbb{Q}\big\}, \tag{3}$$

or the same set partially ordered by inclusion, are defined over the set $\mathbb{Q}$ of rational numbers with the ordering relation $\leq$. The elements of the underlying structure, such as $\mathbb{N}$ or $\mathbb{Q}$ above, will be called *atoms*.

We wish to manipulate first-order definable structures effectively in the context of a functional programming language via a limited interface that can only access atoms by relations in their signature.

* Supported by the Polish National Science Centre (NCN) grant 2012/07/B/ST6/01497.

Atkey & Krishnaswami (Eds.): MSFP 2016
EPTCS ??, 2015, pp. 57–75, doi:10.4204/EPTCS.??.3

Therefore, for example, if a set X is definable over $\mathbb{N}$ with equality, then we do not have the ambition to check whether X contains all even numbers as that property is not expressible using equality alone. On the other hand, we may wish to check whether X is empty or contained in another definable set Y.

Computability of these and other similar conditions relies on first-order properties of the underlying structures of atoms. For example, to ensure that the set (3) contains some nonempty interval, one needs to know that there exist some rational numbers a,c,b such that $a \leq c \leq b$. The structure of atoms should be simple enough for all such conditions to be effectively checkable. For this purpose, we shall assume that underlying structures of atoms are uniquely (up to isomorphism) determined as countable models of their first-order theories and that these first-order theories are decidable. In this paper we concentrate on two particular structures:

- natural numbers $\mathbb{N}$ with equality, understood as the unique countable model of the first-order theory of equality,
- rational numbers $\mathbb{Q}$ with order $\leq$, understood as the unique countable, total, dense order without endpoints.

Our goal is a set of programming idioms that would hide from the programmer as much as it is possible the fact that she or he is dealing with infinite sets presented by first-order formulas rather than with finite sets presented by enumerating their elements. For example, consider a program to compute the transitive closure of a binary relation. When only finite relations on a set X are concerned, one can model them in Haskell as values of the type `Set(a,a)`, assuming that X is a set of values of a type `a`. One can then code a function `compose` that computes the relational composition of two relations and a function `transitiveClosure` to compute the transitive closure of a relation as follows:

```
compose : (Ord a, Ord b, Ord c) => Set(a,b) -> Set(b,c) -> Set(a,c)
compose r s = sum (map (\(a,b) ->
                          map (\(_,c) -> (a,c))
                              (filter ((==b) . fst) s))
                       r)

transitiveClosure : Ord a => Set(a,a) -> Set(a,a)
transitiveClosure r =
  let r' = union r (compose r r)
  in if r==r' then r else (transitiveClosure r')
```

using functions from the standard Haskell module `Data.Set`:

```
sum = unions . elems :: Set (Set a) -> Set a
map :: Ord b => (a -> b) -> Set a -> Set b
filter :: (a -> Bool) -> Set a -> Set a
union :: Ord a => Set a -> Set a -> Set a
```

One of our goals is to provide a version of the `Set` type constructor that would allow the programmer to construct both finite and infinite first-order definable sets and then treat them uniformly, so that the above piece of code could be reused to compute the transitive closure of an infinite relation, internally represented by first-order formulas.

We continue the line of work started in [2], where a core programming language Nλ was introduced, aimed at direct manipulation of orbit-finite nominal sets [15]. These sets are typically infinite, but they can be finitely presented and they are in a strong sense equivalent to first-order definable sets over natural numbers with equality.[1] In [2], nominal sets were constructed using so-called *hulls*, i.e., closures of sets under actions of automorphisms of atoms. Internally they were represented as collections of orbits. For reasons explained in Section 5, we give up the orbit-based presentation of infinite sets and we use a representation based on first-order formulas instead. Technically we keep the syntax of Nλ from [2] with few changes and semantic intuitions remain similar as well: set-typed expressions evaluate to orbit-finite sets or equivalently to first-order definable sets over atoms. However, we further propose a concrete semantics and an implementation that is significantly different from the one in [2]. In particular, sets are represented by first-order formulas rather than on an orbit-by-orbit basis.

Since data values are represented using logical formulas over atoms, in order to evaluate expressions one often needs to evaluate and compare such formulas to check e.g. whether a set is empty or whether two sets are equal. This task fits in the well-researched area of *satisfiability modulo theories* (SMT), and there are off-the-shelf software tools tuned to that purpose. In our implementation we use the freely available Z3 checker [4] developed by Microsoft Research, which offers satisfiability checking for first-order formulas over the theory of equality and over the theory of dense total orders without endpoints. Our implementation of Nλ intensively interacts with Z3 to analyse formulas that arise in representations of infinite data structures. We believe that this application of logical satisfiability checking in functional programming is novel; a similar application in the context of imperative programming has been developed in [10] where mechanisms for manipulating first-order definable sets are added to the language C++.

This paper is closely related to its predecessor [2] and to our sister project [10], but the general idea of symbolic manipulation of infinite sets is far older; indeed, the entire field of constraint programming [16] is based on it. An example of a simple programming language that integrates with an SMT solver is μZ. The language SETL [17] operates on set expressions, but it restricts attention to finite sets. Nominal sets, which are closely related to first-order definable sets, are manipulated in the functional programming language Fresh O'Caml [18], but the main focus there is on atom binding operations, which we do not deal with here.

The structure of this paper is as follows. In Section 2, we introduce first-order definable sets; the presentation is based on [9, 10, 14]. We also relate them to nominal sets [15]. In Section 3, we describe the syntax and intuitive meaning of Nλ programs; this part of the paper is closely related to [2], although the language is changed a little to reflect different semantic choices. In Section 4, a new logic-based semantics of Nλ is provided. Section 5 presents a more detailed comparison to [2], and sketches an extension of the core language of Sections 3–4 with operations to compute hulls and orbits. In Section 6 some implementation issues are explained, and Section 7 illustrates the use of Nλ on two simple examples.

A prototype implementation of Nλ as a Haskell module is available for download from [13].

Acknowledgments. We are grateful to Eryk Kopczyński and Szymon Toruńczyk, who came up with the idea of using formulas to represent orbit-finite sets with atoms, and whose work on the LOIS library for C++ [10] has been a source of constant inspiration. We also thank anonymous reviewers whose insightful comments helped us improve the paper.

2 Sets with atoms

Fix a countably infinite relational structure $\mathscr{A}$ over some finite signature Σ. We call the elements of $\mathscr{A}$ *atoms*. It would be enough to assume that $\mathscr{A}$ has a decidable first-order theory and it is an *ultrahomoge-*

[1] Other underlying structures of atoms were also considered in [2], with assumptions similar to ours.

nous structure, also known as a *Fraïssé limit* [7]. In particular, this implies that $\mathscr{A}$:

- is ω-*categorical*, i.e., it is the only (up to isomorphism) countable model of its first-order theory and
- has *quantifier elimination*, i.e., every first-order formula over $\mathscr{A}$ is equivalent to a quantifier-free formula.

In this paper and for the purposes of implementation we focus on two particular structures with all these properties:

- $\mathscr{A} = (\mathbb{N}, =)$, i.e., natural numbers with equality (we call these *equality atoms*)
- $\mathscr{A} = (\mathbb{Q}, \leq)$, i.e., rational numbers with ordering (we call these *ordered atoms*).

For a fixed structure $\mathscr{A}$, a *set expression* is

- a variable x from some fixed infinite set of atom variables, or
- a finite sequence, written $\{\xi_1, \ldots, \xi_n\}$ (or $\{\}$ for the empty sequence), of expressions of the form

  $$\xi = e : \phi \text{ for } x_1, \ldots, x_k \tag{4}$$

  where e is a set expression, ϕ is a first-order formula over Σ, and $x_1, \ldots, x_k$ are atom variables.

If $k = 0$ then we write simply $e : \phi$ instead of (4). We also omit ϕ if it is the always true formula $\top$.

The set of free variables in a set expression is defined inductively by:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(e : \phi \text{ for } x_1, \ldots, x_k) &= FV(e) \cup FV(\phi) \setminus \{x_1, \ldots, x_k\} \\ FV(\{\xi_1, \ldots, \xi_n\}) &= FV(\xi_1) \cup \cdots \cup FV(\xi_n) \end{aligned}$$

where $FV(\phi)$ is the standard set of free (atom) variables in a first-order formula. A *valuation* for a set expression e is a function $v : FV(e) \to \mathscr{A}$. A set expression e together with a valuation v denotes a set (or an atom) $[\![e]\!]_v$ in the expected way:

$$\begin{aligned} [\![x]\!]_v &= v(x) \\ [\![e : \phi \text{ for } x_1, \ldots, x_k]\!]_v &= \left\{ [\![e]\!]_{v[x_i \mapsto a_i]} \mid a_1, \ldots, a_n \in \mathscr{A} \text{ s.t. } \mathscr{A}, v[x_i \mapsto a_i] \models \phi \right\} \\ [\![\{\xi_1, \ldots, \xi_n\}]\!]_v &= [\![\xi_1]\!]_v \cup \cdots \cup [\![\xi_n]\!]_v \end{aligned}$$

where $\mathscr{A}, v \models \phi$ means that the formula ϕ holds in $\mathscr{A}$ with the valuation v of the free variables in ϕ. We say that a set of the form $[\![e]\!]_v$ is *definable* over $\mathscr{A}$.

Standard set-theoretic tricks can be used to encode ordered pairs (e.g. as Kuratowski pairs $(x,y) = \{\{x\}, \{x,y\}\}$), tuples (as nested pairs), and integers (e.g. as von Neumann numerals $n = \{0, 1, \ldots, n-1\}$).

For example, over equality atoms, the expression

$$\{(x,y) : \neg(x = y) \text{ for } x, y\},$$

with the empty valuation denotes the set of ordered pairs of distinct atoms. The same definition works for ordered atoms, where we see $x = y$ as shorthand for $x \leq y \wedge y \leq x$. Over ordered atoms, the expression

$$\{x : x \leq u \text{ for } x,\ y : w \leq y \text{ for } y\}$$

with a valuation $u \mapsto 2,\ w \mapsto 5$, denotes the set of all atoms outside of the open interval $(2;5)$. The same set is denoted by the expression

$$\{x : x \leq u \vee w \leq x \text{ for } x\}$$

with the same valuation.

We shall restrict attention to *well-typed* expressions with a set of types defined by:

$$\tau, \rho ::= \mathsf{A} \mid \mathsf{N} \mid (\tau, \rho) \mid \mathbb{S}\tau \tag{5}$$

where $\mathbb{S}$ is a unary type constructor, with $\mathbb{S}\tau$ meant to be the type of sets whose elements are of type τ. Set expressions are provided with types by the following relation (actually, a partial function):

$$\frac{}{x : \mathsf{A}} \qquad \frac{}{n : \mathsf{N}} \qquad \frac{e_1 : \tau \quad e_2 : \rho}{(e_1, e_2) : (\tau, \rho)} \qquad \frac{e_1 : \tau \quad \cdots \quad e_n : \tau}{\{e_1 : \phi_1 \cdots, \ldots, e_n : \phi_n \cdots\} : \mathbb{S}\tau}$$

where x ranges over atom variables and n over integers. Essentially it is required that all elements of a well-typed set have the same type. Pairs and integers are treated separetely here, since neither Kuratowski pairs nor von Neumann numerals are well typed in this sense.

The above constructions appear in the literature under various guises. Indeed, sets definable over atoms $\mathscr{A}$ are essentially *first-order interpretable* structures over $\mathscr{A}$ in the sense of model theory [7]. They also correspond to nominal sets [15]; we sketch this connection briefly as it relates this paper to previous work [2] on extending functional programming to sets with atoms.

Consider a set X with a group action $\_ \cdot \_ : \mathrm{Aut}(\mathscr{A}) \times X \to X$ of the automorphism group of the structure $\mathscr{A}$. A set $S \subseteq \mathscr{A}$ *supports* an element $x \in X$ if $\pi \cdot x = x$ for every $\pi \in \mathrm{Aut}(\mathscr{A})$ such that $\pi(a) = a$ for all $a \in S$. If every element of X has some finite support, then X is called *$\mathscr{A}$-nominal.* For $\mathscr{A}$ equality atoms this specializes to the notion considered in [15].

A function $f : X \to Y$ between nominal sets is *equivariant* if $f(\pi \cdot x) = \pi \cdot f(x)$ for every $x \in X$ and $\pi \in \mathrm{Aut}(\mathscr{A})$.

An *orbit* of an element $x \in X$ is the set $\{\pi \cdot x \mid \pi \in \mathrm{Aut}(\mathscr{A})\} \subseteq X$. Orbits form a partition of the $\mathscr{A}$-nominal set X; we call X *orbit-finite* if it has finitely many orbits.

For every set expression e without free variables, the set $[\![e]\!]_\emptyset$ is equipped with a canonical group action of $\mathrm{Aut}(\mathscr{A})$: for $e' : \phi$ for $x_1, \ldots, x_k$ a part of e, and for $a_1, \ldots, a_n \in \mathscr{A}$ such that $\mathscr{A}, [x_i \mapsto a_i] \models \phi$, define

$$\pi \cdot [\![e']\!]_{[x_i \mapsto a_i]} = [\![e']\!]_{[x_i \mapsto \pi(a_i)]};$$

$\mathscr{A}, [x_i \mapsto \pi(a_i)] \models \phi$ follows from π being an automorphism of $\mathscr{A}$, since $FV(\phi) \subseteq \{x_1, \ldots, x_n\}$. It is easy to see that $[\![e']\!]_{[x_i \mapsto a_i]}$ is supported by $\{a_1, \ldots, a_n\}$, so $[\![e]\!]_\emptyset$ is a $\mathscr{A}$-nominal set. Moreover, the set is orbit-finite; this follows from the fact that for every ω-categorical structure $\mathscr{A}$, the set $\mathscr{A}^n$ with the pointwise action of $\mathrm{Aut}(\mathscr{A})$ is orbit-finite, by the celebrated Ryll-Nardzewski theorem from model theory [7].

This means that every set definable by an expression without free variables is $\mathscr{A}$-nominal and orbit-finite. The converse also holds: every $\mathscr{A}$-nominal, orbit-finite set is equivariantly bijective to a set of the form $[\![e]\!]_\emptyset$ for some set expression e. Moreover, if pairs are included in the language of expressions, one can choose e to be well-typed. Details of this correspondence are developed in the first chapter of [14].

In [2], a functional programming language Nλ was designed to compute and manipulate orbit-finite nominal sets. There, infinite structures were internally represented on an orbit-by-orbit basis using a representation theorem from [3] saying that every single-orbit set is in equivariant bijection with a set of tuples of atoms quotiented by an equivalence relation of a certain shape. In this paper we continue the programme of [2] and develop a language with a new semantics and implementation, where orbit-finite sets are internally represented by set expressions over atoms.

3 A basic functional language

To provide a functional language to construct and operate on definable sets over atoms, begin with a lambda calculus with a type A for atoms and a type B for boolean values, extended with a unary type constructor $\mathbb{S}$ that cannot be applied to values of function types. Thus types are defined by the following grammar:

$$\begin{aligned} \tau &::= \mathsf{A} \mid \mathsf{B} \mid \mathbb{S}\tau \\ \alpha,\beta &::= \tau \mid \alpha \to \beta \end{aligned}$$

The intuition is that values of type $\mathbb{S}\tau$ are (definable) sets of values of type τ. This excludes function types, as one expects set elements to be equipped with a computable equality operation.

Terms of the core language are defined by the grammar:

$$M ::= C \mid x \mid \lambda x.M \mid MM$$

with the usual typing relation of lambda calculus, where C comes from the following set of typed constants:

$\mathtt{empty} : \mathbb{S}\tau$	(the empty set)
$\mathtt{atoms} : \mathbb{S}\mathsf{A}$	(the set of all atoms)
$\mathtt{insert} : \tau \to \mathbb{S}\tau \to \mathbb{S}\tau$	(adds an element to a set)
$\mathtt{map} : (\tau_1 \to \tau_2) \to \mathbb{S}\tau_1 \to \mathbb{S}\tau_2$	(applies a function to every element)
$\mathtt{sum} : \mathbb{S}\mathbb{S}\tau \to \mathbb{S}\tau$	(union of a family of sets)
$\mathtt{true},\mathtt{false} : \mathsf{B}$	(boolean values)
$\mathtt{not} : \mathsf{B} \to \mathsf{B}$	(logical negation)
$\mathtt{and},\mathtt{or} : \mathsf{B} \to \mathsf{B} \to \mathsf{B}$	(conjunction and disjunction)
$\mathtt{isEmpty} : \mathbb{S}\tau \to \mathsf{B}$	(emptiness test)
$\mathtt{if} : \mathsf{B} \to \alpha \to \alpha \to \alpha$	(conditional)

We refrain from providing formal semantics for all these operations until the next section, but their meaning should be intuitively clear as specified on the right above. Additionally, we include some constants that depend on the signature of the underlying structure $\mathscr{A}$ of atoms. For equality atoms we take simply:

$\mathtt{eq}_{\mathsf{A}} : \mathsf{A} \to \mathsf{A} \to \mathsf{B}$	(equality relation on atoms)

and for ordered atoms, additionally:

$\mathtt{leq} : \mathsf{A} \to \mathsf{A} \to \mathsf{B}$	(ordering relation on atoms).

For other structures $\mathscr{A}$ this part of the language may change.

This core language can be extended with product types, integers, (mutually) recursive definitions, algebraic types and other features using standard techniques; we omit the details for brevity, noting only that the type metavariable τ should include all equality types. One can then define additional functions such as:

$$\begin{aligned}
\texttt{singleton} &: \tau \to \mathbb{S}\tau & \texttt{singleton x} &= \texttt{insert x empty}\\
\texttt{filter} &: (\tau \to \mathsf{B}) \to \mathbb{S}\tau \to \mathbb{S}\tau & \texttt{filter f s} &= \texttt{sum (map}\\
& & &\quad (\lambda\texttt{x.if (f x) (singleton x) empty) s)}\\
\texttt{exists} &: (\tau \to \mathsf{B}) \to \mathbb{S}\tau \to \mathsf{B} & \texttt{exists f s} &= \texttt{not (isEmpty (filter f s))}\\
\texttt{forall} &: (\tau \to \mathsf{B}) \to \mathbb{S}\tau \to \mathsf{B} & \texttt{forall f s} &= \texttt{isEmpty (filter (}\lambda x\texttt{.not (f x)) s)}\\
\texttt{contains} &: \mathbb{S}\tau \to \tau \to \mathsf{B} & \texttt{contains s x} &= \texttt{exists (eq x) s}\\
\texttt{isSubsetOf} &: \mathbb{S}\tau \to \mathbb{S}\tau \to \mathsf{B} & \texttt{isSubsetOf s t} &= \texttt{forall (contains t) s}\\
\texttt{eq} &: \mathbb{S}\tau \to \mathbb{S}\tau \to \mathsf{B} & \texttt{eq s t} &= \texttt{and (isSubsetOf s t)}\\
& & &\quad \texttt{(isSubsetOf t s)}\\
\texttt{union} &: \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau & \texttt{union s t} &= \texttt{sum (insert s (singleton t))}\\
\texttt{intersection} &: \mathbb{S}\tau \to \mathbb{S}\tau \to \mathbb{S}\tau & \texttt{intersection s t} &= \texttt{filter (contains t) s}
\end{aligned}$$

and so on. In particular, for an equality type τ, equality can be defined for the type $\mathbb{S}\tau$.

One can also construct sets definable by well-typed set expressions. For example,

$$\texttt{atomPairs} = \texttt{sum (map (}\lambda\texttt{x.map (}\lambda\texttt{y.(x,y)) atoms) atoms)} : \mathbb{S}(\mathsf{A},\mathsf{A})$$

evaluates to the set of all pairs of atoms, and

$$\texttt{filter (}\lambda\texttt{(x,y).not(eq}_{\mathsf{A}}\texttt{ x y)) atomPairs} : \mathbb{S}(\mathsf{A},\mathsf{A})$$

to the set of all distinct pairs of atoms.

In general, every set over atoms that is definable by a well-typed set expression is a value of some program. More formally, for every set expression $e : \tau$ with free variables $x_1,\ldots,x_k$ there is a term $\texttt{set}_e : \mathsf{A}^k \to \tau$ in the programming language, that evaluates to a function from $\mathscr{A}^k$ that, when applied to arguments $a_1,\ldots,a_k$, returns $[\![e]\!]_{[x_i\mapsto a_i]}$. This follows by induction on the structure of expressions. The only interesting case is

$$e = \{e' : \phi \text{ for } x_{k+1},\ldots,x_m\} : \mathbb{S}\tau$$

for some $e' : \tau$ such that $FV(e'), FV(\phi) \subseteq \{x_1,\ldots,x_m\}$. It is easy to generalize the term `atomPairs` above to a function

$$\texttt{atomTuples}_{k,m} : \mathsf{A}^k \to \mathbb{S}\mathsf{A}^m$$

that extends a given k-tuple of atoms to the set of all m-tuples that arise by putting arbitrary atoms on the remaining $m-k$ components. Then put

$$\texttt{set}_e\ \texttt{t} = \texttt{map set}_{e'}\ \texttt{(filter form}_\phi\ \texttt{(atomTuples}_{k,m}\ \texttt{t))}$$

where $\texttt{set}_{e'}$ exists by the inductive assumption, and $\texttt{form}_\phi : \mathsf{A}^m \to \mathsf{B}$ is a term that encodes the first-order formula ϕ. Such a term exists since $\mathscr{A}$ has quantifier elimination, and so without loss of generality we may assume that ϕ is quantifier-free.

4 Logic-based semantics

From the description of the language in Section 3 it may not be clear how to implement operations postulated in it. For example, how to implement the function `map` so that a function can be applied to every element of the infinite set of atoms in finite time? In this section we provide a (small-step) reduction semantics of the core functional language, that implements the set-theoretic intuitions provided in Section 3, yet is clearly computable.

The semantics is based on the following general ideas:

- Values of set types $\mathbb{S}\tau$ are represented not by enumerating their elements (that would be impossible, as usually they are infinite sets), but by set expressions as in Section 2.
- Values of type B are not just boolean values; they are rather first-order formulas over a special kind of variables called *atom variables* that denote atoms.
- Terms are evaluated in contexts that specify what relations hold between atom variables in them.
- Sometimes a condition ϕ in a conditional expression `if` ϕ M N is neither tautologically true nor false. In such cases it is not clear whether the conditional should evaluate to M or N and the choice is delayed for as long as possible. When delaying is not further possible, e.g. when M and N are atom variables, a *variant* is created that has value M or N, formally depending on the value of ϕ.

Formally, keeping the set of types as in Section 3, we extend the grammar of terms to:

$$M ::= C \;\Big|\; x \;\Big|\; \lambda x.M \;\Big|\; MM \;\Big|\; a \;\Big|\; \phi \;\Big|\; \{M : \phi \text{ for } \sigma, \ldots, M : \phi \text{ for } \sigma\} \;\Big|\; M : \phi | \cdots | M : \phi$$

where:

- C ranges over the same set of typed constants as in Section 3,
- a ranges over a fixed infinite set of *atom variables*, disjoint from the set of program variables such as x,
- ϕ ranges over the set of first-order formulas (with quantifiers allowed) over the signature of $\mathscr{A}$ and over atom variables,
- σ ranges over finite sets of atom variables. We omit "for σ" if σ is empty.

Note that the new terms are unavailable to the programmer and they shall appear only as final or intermediate values in the reduction semantics.

Atom variables in the sets σ in set expressions are binding occurrences, just as the program variable x is a binding occurrence in $\lambda x.M$. Terms are considered up to α-equivalence, defined as expected. For example,

$$\{a : \neg(a = c) \text{ for } a\} \qquad \text{and} \qquad \{b : \neg(b = c) \text{ for } b\}$$

are α-equivalent.

Expressions of the form

$$M_1 : \phi_1 | \cdots | M_n : \phi_n$$

are called *variants*. They look syntactically similar to set expressions of the form $\{M_1 : \phi_1, \ldots, M_n : \phi_n\}$, but their meaning is very different. A variant as above does not denote a set of values, but a *single* value whose identity cannot be determined at the moment and will be fixed depending on which one of the formulas ϕ_1 to ϕ_n holds.

In addition to standard typing rules for the lambda calculus, the newly added terms are typed according to:

$$a : \mathsf{A} \qquad \phi : \mathsf{B} \qquad \frac{M_1 : \tau \quad \cdots \quad M_n : \tau}{\{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} : \mathbb{S}\tau} \qquad \frac{M_1 : \tau \quad \cdots \quad M_n : \tau}{(M_1 : \phi_1 | \cdots | M_n : \phi_n) : \tau} \tag{6}$$

relative to any typing context of free program variables in $M_1, \ldots, M_n$.

We define a small-step operational semantics where terms are evaluated in the context of a formula over atom variables. The basic semantic statements are of the form

$$\psi \vdash M \to N$$

where ψ is a formula and M, N are program terms. Reduction rules are given in Fig. 1.

Rules (7) provide the standard infrastructure of the lambda calculus. The notion of capture-avoiding substitution $M[N/x]$ works as usual taking into account the fact that atom variables in σ bind in $\{M : \phi \text{ for } \sigma\}$. We do not commit to any particular reduction strategy allowing reductions both in functions and in their arguments.

Rules (8)–(14) are mostly self-explanatory and they agree with the intuitive meaning of program constants as listed in Section 3. We only note that in rule (11), inner expressions $M_i : \phi_i$ for σ_i may need to be α-converted so that the side condition of the rule holds. Note also that the rule for `atoms` in (8) is the only place where a new atom variable is created and that rule (14) may cause quantified first-order formulas to appear.

The conditional constant `if` is evaluated in a special way and it deserves a separate section of the semantics. A premise $\mathscr{A} \models \psi \Rightarrow \phi$ means that the formula $\psi \Rightarrow \phi$ holds in $\mathscr{A}$ under every valuation of its free variables. If some valuation falsifies the formula, we write $\mathscr{A} \not\models \psi \Rightarrow \phi$. Rules (15) apply where the value of the logical condition ϕ is determined by the ambient formula ψ. In such situations the condition ϕ behaves like a standard boolean value and the conditional expression is resolved as expected.

If the value of ϕ remains undetermined under the assumption of ψ, then both values to be chosen from must be combined in the result of the conditional expression. The course of action depends on the type of those values with the general idea to postpone the choice by pushing it down the structure of terms. If the two values are functions, in (16) a new "lazy" function is created where the choice is postponed until the function argument is provided. If they are formulas or set expressions, rules (16) and (17) combine them in an expected way. The most interesting case is a choice between atom variables: in rule (18), a variant is created. It may be seen as an "ambiguous atom" equal to a or b depending on the value of ϕ. Formally, a separate rule (19) for a choice between variants is required but it works as expected similarly to rule (17).

Notice that rule (18) is the only place where variants are created, and those variants are always built of atom variables. One may wonder why the typing rule for variants in (6) allowed arbitrary types τ instead of simply A. This is in anticipation of other basic types added to the language such as integers or strings, excluded from the core language for brevity. For each such basic type, a rule corresponding to (18) would need to be added.

Variants tend to be short-lived intermediate values and they are dissolved as soon as they emerge as elements of set expressions. Rule (21) shows how this is done. Rules (20) specify how reductions are done in the context of set expressions and variants; these rules show how ambient formulas ψ are constructed.

Rule (22) specifies the behaviour of the equality function used for equality atoms. This rule also applies to single atom variables which are here understood as degenerated variants $a : \top$. For ordered atoms the function `leq` is specified analogously.

β-reduction:

$$\frac{\psi \vdash M \to M'}{\psi \vdash MN \to M'N} \qquad \frac{\psi \vdash N \to N'}{\psi \vdash MN \to MN'} \qquad \psi \vdash (\lambda x.M)\, N \to M[N/x] \tag{7}$$

Basic constants:

$$\psi \vdash \texttt{empty} \to \{\;\} \qquad \psi \vdash \texttt{atoms} \to \{a : \top \text{ for } a\} \tag{8}$$

$$\psi \vdash \texttt{insert}\, M\ \{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} \to \{M : \top, M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} \tag{9}$$

$$\psi \vdash \texttt{map}\, M\ \{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} \to \{MM_1 : \phi_1 \text{ for } \sigma_1, \ldots, MM_n : \phi_n \text{ for } \sigma_n\} \tag{10}$$

$$\begin{array}{l} \psi \vdash \texttt{sum}\ \{\ldots, \{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} : \phi \text{ for } \sigma, \ldots\} \\ \qquad \to \{\ldots, M_1 : \phi_1 \wedge \phi \text{ for } \sigma_1 \cup \sigma, \ldots,\ M_n : \phi_n \wedge \phi \text{ for } \sigma_n \cup \sigma, \ldots\} \end{array} \qquad \text{if } \sigma \cap \bigcup_{i=1}^{n} \sigma_i = \emptyset \tag{11}$$

$$\psi \vdash \texttt{true} \to \top \qquad \psi \vdash \texttt{false} \to \bot \qquad \psi \vdash \texttt{not}\ \phi \to \neg\phi \tag{12}$$

$$\psi \vdash \texttt{or}\ \phi_1\ \phi_2 \to \phi_1 \vee \phi_2 \qquad \psi \vdash \texttt{and}\ \phi_1\ \phi_2 \to \phi_1 \wedge \phi_2 \tag{13}$$

$$\psi \vdash \texttt{isEmpty}\ \{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\} \to \bigwedge_{1 \le i \le n} \underbrace{\forall a_1 \forall a_2 \cdots \forall a_k}_{\sigma_i = \{a_1, \ldots, a_k\}} .\neg\phi_i \tag{14}$$

Conditional expressions:

$$\frac{\mathscr{A} \models \psi \Rightarrow \phi}{\psi \vdash \texttt{if}\ \phi\ M\ N \to M} \qquad \frac{\mathscr{A} \models \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ M\ N \to N} \tag{15}$$

$$\frac{\mathscr{A} \not\models \psi \Rightarrow \phi \quad \mathscr{A} \not\models \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ \lambda x.M\ \lambda x.N\ \to \lambda x.(\texttt{if}\ \phi\ M\ N)} \qquad \frac{\mathscr{A} \not\models \psi \Rightarrow \phi \quad \mathscr{A} \not\models \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ \phi_1\ \phi_2 \to (\phi_1 \wedge \phi) \vee (\phi_2 \wedge \neg\phi)} \tag{16}$$

$$\frac{\mathscr{A} \not\models \psi \Rightarrow \phi \quad \mathscr{A} \not\models \psi \Rightarrow \neg\phi}{\begin{array}{l} \psi \vdash \texttt{if}\ \phi\ \{M_1 : \phi_1 \text{ for } \sigma_1, \ldots, M_n : \phi_n \text{ for } \sigma_n\}\ \{N_1 : \theta_1 \text{ for } \pi_1, \ldots, N_k : \theta_k \text{ for } \pi_k\} \\ \quad \to \{M_1 : \phi_1 \wedge \phi \text{ for } \sigma_1, \ldots, M_n : \phi_n \wedge \phi \text{ for } \sigma_n, N_1 : \theta_1 \wedge \neg\phi \text{ for } \pi_1, \ldots, N_k : \theta_k \wedge \neg\phi \text{ for } \pi_k\} \end{array}} \tag{17}$$

$$\frac{\mathscr{A} \not\models \psi \Rightarrow \phi \quad \mathscr{A} \not\models \psi \Rightarrow \neg\phi}{\psi \vdash \texttt{if}\ \phi\ a\ b \to a : \phi | b : \neg\phi} \tag{18}$$

$$\frac{\mathscr{A} \not\models \psi \Rightarrow \phi \quad \mathscr{A} \not\models \psi \Rightarrow \neg\phi}{\begin{array}{l} \psi \vdash \texttt{if}\ \phi\ (M_1 : \phi_1 | \cdots | M_n : \phi_n)\ (N_1 : \theta_1 | \cdots | N_k : \theta_k) \\ \quad \to M_1 : \phi_1 \wedge \phi | \cdots | M_n : \phi_n \wedge \phi | N_1 : \theta_1 \wedge \neg\phi | \ldots | N_k : \theta_k \wedge \neg\phi \end{array}} \tag{19}$$

Set and variant reduction:

$$\frac{\psi \wedge \phi \vdash M \to N}{\psi \vdash \{\ldots, M : \phi \text{ for } \sigma, \ldots\} \to \{\ldots, N : \phi \text{ for } \sigma, \ldots\}} \qquad \frac{\psi \wedge \phi \vdash M \to N}{\psi \vdash \cdots | M : \phi | \cdots \to \cdots | N : \phi | \cdots} \tag{20}$$

$$\psi \vdash \{\ldots, (M_1 : \phi_1 | \cdots | M_n : \phi_n) : \phi \text{ for } \sigma, \ldots\} \to \{\ldots, M_1 : \phi_1 \wedge \phi \text{ for } \sigma, \ldots, M_n : \phi_n \wedge \phi \text{ for } \sigma, \ldots\} \tag{21}$$

Equality:

$$\psi \vdash \mathsf{eq}_{\mathsf{A}}\ (a_1 : \phi_1 | \ldots | a_n : \phi_n)\ (b_1 : \theta_1 | \ldots | b_m : \theta_m) \to \bigvee_{\substack{1 \le i \le n \\ 1 \le j \le m}} (a_i = b_j \wedge \phi_i \wedge \theta_j) \tag{22}$$

Figure 1: Reduction semantics

This reduction semantics has a few expected properties proved by standard arguments:

- *subject reduction* holds, i.e., the reduction relation preserves types,
- the *Church-Rosser property* holds up to first-order formula equivalence, i.e., if $\phi \vdash M \to N$ and $\phi \vdash M \to N'$ then there exist terms Q and Q' such that $\phi \vdash N \to^* Q$ and $\phi \vdash N' \to^* Q'$, where $\to^*$ is the reflexive and transitive closure of $\to$, Q and Q' are equal up to replacing some first-order formulas with equivalent ones. This follows by a parallel reductions argument as described in [19].
- *(weak) normalisation* holds, i.e., each term can be reduced to an irreducible value. This is proved by a standard type of argument [5] assigning degrees to types of the language.

Obviously, normalization fails as soon as the core language is extended with recursion as non-terminating programs can then be written. Otherwise, the semantics can be routinely extended with product types and terms, integers, mutually recursive definitions, algebraic types, etc. This is illustrated by our implementation described in Section 6. Indeed, we do not implement the language from scratch; instead, we write a Haskell module to support features described here, allowing the programmer to use them in conjunction with the power of a full-fledged functional programming language.

5 Hulls, supports and orbits

In [2], which is a direct predecessor to this paper, a different internal representation of infinite sets was used. To construct such sets a programming construction `hull` was provided, which, given a finite list C of atoms and a set of values X of some type (possibly built of atoms), returned the closure of X under all automorphisms of atoms that fix every element of C. For example, the expression

```
hull [] {2}
```

evaluates to the set of all atoms, because every atom can be obtained from the atom 2 by an application of an automorphism of $\mathscr{A}$ that fixes (which is a non-condition) every element of the empty list. Similarly,

```
hull [3] {2}          hull [2] {(2,5)}
```

evaluate respectively to the set of atoms different from 3, and to the set of pairs of atoms where the first element is 2 and the second is different from 2. If ordered atoms are considered, the expression

```
hull [] {(2,3)}
```

evaluates to the set of pairs where the second component is strictly greater than the first one. One could then manipulate sets constructed in this way using functions such as `map` and `sum`, so that, e.g., functions `compose` and `transitiveClosure` could be written more or less as in Section 1. Internally, infinite sets were not represented by first-order formulas. Rather, the hull construction was used as a basic semantic construct in computed values of set types; see [2] for details.

The mechanism for representing infinite sets using hulls has a number of disadvantages. Most importantly, the size of the representation of an orbit-finite set is proportional to the number of its orbits. For example, the set of all triples of atoms is constructed by

```
hull [] {(1,1,1),(1,1,2),(1,2,1),(2,1,1),(1,2,3)}
```

and in general the set of ordered n-tuples needs an internal representation of size exponential in n. This is rather inefficient and as a result in the prototype Haskell implementation of Nλ from [2] only very rudimentary programs could be evaluated in reasonable time. Note that in our semantics the set of atom triples is represented internally by the more concise

$$\{(a_1,a_2,a_3) : \top \text{ for } a_1,a_2,a_3\}.$$

Another problem is that `hull`-based definitions of sets require the use of constants that denote particular atoms, even if mathematical definitions of the same sets do not need to. For example, even though no concrete natural numbers are mentioned in a mathematical definition of triples of numbers, as many as three numbers are used in the `hull`-based definition above. This is not a major problem when equality atoms are concerned, but with more sophisticated structures of atoms it would cause difficulties. For example, although the universal partial order [8] is a legal and well-behaved structure of atoms, no easy and natural representation of it is known and it is not clear how to denote its particular elements in a convenient way.

For these reasons, in this paper we replace the hull-based representation with the logic-based semantics from Section 4. One may even contemplate removing the `hull` construction from the language available to the programmer, and indeed this is what we did for the core language in Sections 3–4. This is justified by the observation from Section 3, missed in [2], that every definable set can be denoted by a program without `hull`. On the other hand, it is not clear how to define the hull function itself:

$$\mathtt{hull} : [\mathsf{A}] \to \mathbb{S}\tau \to \mathbb{S}\tau$$

in the core language (extended with list types $[\alpha]$ in a standard way). As this function is sometimes useful to the programmer, we add it to the language along with a few other basic functions:

$\mathtt{groupAction} : (\mathsf{A} \to \mathsf{A}) \to \tau \to \tau$	(renames free atoms in an argument)
$\mathtt{supports} : [\mathsf{A}] \to \tau \to \mathsf{B}$	(checks if a list of atoms supports the argument)
$\mathtt{support} : \tau \to [\mathsf{A}]$	(returns some finite support of the argument; efficient)
$\mathtt{leastSupport} : \tau \to [\mathsf{A}]$	(returns the least support of the argument; less efficient)
$\mathtt{setOrbit} : \mathbb{S}\tau \to \tau \to \mathbb{S}\tau$	(returns the orbit of an element in a set)
$\mathtt{setOrbits} : \mathbb{S}\tau \to \mathbb{S}\mathbb{S}\tau$	(returns the (finite) set of orbits of a given set)

In [2], most of these functions or their minor variations were derived from `hull`. For example, one may write:

$\mathtt{isSingleton} : \mathbb{S}\tau \to \mathsf{B}$	$\mathtt{isSingleton}\ \mathtt{s} = \mathtt{exists}\ (\lambda\mathtt{x}.\mathtt{forall}\ (\mathtt{eq}\ \mathtt{x})\ \mathtt{s})\ \mathtt{s}$
$\mathtt{supports} : [\mathsf{A}] \to \tau \to \mathsf{B}$	$\mathtt{supports}\ \mathtt{c}\ \mathtt{x} = \mathtt{isSingleton}\ (\mathtt{hull}\ \mathtt{c}\ (\mathtt{singleton}\ \mathtt{x}))$

However, such definitions are rather inefficient. Here, we include `groupAction` and `support` as basic operations and define `hull` and other functions from them, which results in a more efficient implementation.

6 Implementation

We implement Nλ as a Haskell module (available from [13]), which allows the programmer to use all benefits of a full-fledged functional programming language. The module introduces new types and functions operating on infinite structures and first-order formulas.

We shall now explain a few aspects of the implementation worth mentioning.

SMT solving

In Fig. 1, rules (15)–(19) involve premises of the form $\mathscr{A} \models \psi \Rightarrow \phi$, stating that a formula holds in the structure $\mathscr{A}$. Since we only consider ω-categorical structures of atoms, one may equivalently ask whether $\psi \Rightarrow \phi$ follows from the axioms of the first-order theory of $\mathscr{A}$. This is an instance of the general satisfiability modulo theories (SMT) problem and there are software tools available that perform that task efficiently for a variety of first-order theories.

To determine whether a formula holds in $\mathscr{A}$ the interpreter of Nλ calls the external Z3 solver [4] via a system call. The implementation can be easily modified to connect to any other solver compatible with the SMT-LIB standard [1] instead. Currently two SMT-LIB logics are used: LIA (linear integer arithmetic, for equality atoms) and LRA (linear real arithmetic, for ordered atoms). Formula solving is a pure function without side-effects, therefore it is invoked within the Haskell `unsafePerformIO` function to avoid putting the `IO` monad in types of all conditional statements in Nλ.

Experiments performed in our companion project LOIS [10] showed that SMT solvers in general and Z3 in particular do not deal well with quantified formulas that do not involve arithmetic. To improve performance before calling Z3, the interpreter eliminates all quantifiers from the formula to be checked. The quantifier elimination algorithm used for ordered atoms is based on the method of infinitesimals for linear real arithmetic proposed by Loos and Weispfenning [11] and adapted by Nipkow to dense linear order [12] (for equality atoms it is enough to use a simplified version of this algorithm). Roughly, this method involves replacing an existentially quantified formula by a disjunction of formulas where the bound variable is substituted by test points which include values arbitrarily close to either lower or upper bounds of the eliminated variable.

Conditionals

From rules (15)–(19) in Fig. 1 it is clear that the conditional expression in Nλ is substantially different from the standard Haskell `if`...`then`...`else`... construction, in that it must deal with conditions that cannot be resolved to `true` or `false`. Since `if` is a Haskell keyword, a different name must be used for Nλ conditionals; we choose

```
ite :: Conditional a => Formula -> a -> a -> a
```

This function is implemented for all instances of the new `Conditional` typeclass, which includes several basic types, the atom and formula types, list and function types. The function `ite` first tries to determine the logical value of the condition formula with a SMT solver call; failing that, it calls a function `cond` of the same type as `ite` that is defined in a type-specific manner.

For example, the implementation of `cond` for the formula type is:

```
instance Conditional Formula where
  cond f1 f2 f3 = (f1 /\ f2) \/ (not f1 /\ f3)
```

For the function type it works in a lazy way:

```
instance Conditional b => Conditional (a -> b) where
  cond c f1 f2 = \x -> cond c (f1 x) (f2 x)
```

These definitions correspond to rules (16) in Fig. 1.

The result for the type of (definable) sets includes elements from both input sets but with appropriate formulas, according to rule (17) in Fig. 1. In other collection types (lists, tuples, etc.), missing from the core language of Nλ, condition handling is passed to elements. The function for lists with the same lengths is coded as follows:

```
  cond c l1 l2 = zipWith (cond c) l1 l2
```

One problem appears for an ambiguous condition on lists of different lengths. To simplify the implementation we decided to report an error in this case. However, operations on lists can be performed alternatively using the `Variants` constructor.

Variants and contexts

Of course some types (such as integer types) cannot cope with an ambiguous condition in any other way than to somehow return both values. For such types a special type constructor `Variants` is provided; values of type `Variants a` are lists of values of type `a` coupled with formulas. It comes with its counterpart of `ite` function, defined for any type `a`:

```
iteV :: Formula -> a -> a -> Variants a
```

Thus one can implement conditional statements e.g. for integers: `iteV (eq a b) 1 2` will return a variant $1 : a = b \mid 2 : a \neq b$, akin to rule (18) in Fig. 1. The type of atoms `Atom` itself is actually defined as the variant type of variable names. Every variant type is an instance of the class `Conditional`.

However, not always all possible result variants of the program are desired. Sometimes the result is interesting only in a given context. In such cases the new class `Contextual` is useful. A function

```
when :: Contextual a => Formula -> a -> a
```

introduces a formula into the context of a computation. For example, expression

```
when (neq a b /\ neq b c /\ neq a c) size (fromList [a,b,c])
```

will display only the result for distinct atoms. This corresponds to adding formulas to contexts in rules (20) in Fig. 1.

Nominal types

The basic type class in Nλ is `NominalType` corresponding to types ranged over by the τ metavariable in our core language. This class is required by several functions of the language and is important for three reasons:

- it provides an implementation of the equality predicate `eq`,
- it has functions that operate on atom variables (`mapVariables` and `foldVariables`) and are used internally for resolving conflicts between atom variable names, and for collecting all or free atom variables that occur in a set expression,
- it helps split variant values into elements when inserting them to the set (to implement rule (21) in Fig. 1).

To operate on a set of elements of a given type, the type has to be an instance of `NominalType`. Additionally, all instances of this class must be instances of the standard Haskell class `Ord`. This is to improve performance.

Set types

The `Set` type constructor is an implementation of both infinite and finite sets. Generally, it is an alternative to the standard `Data.Set` module with most features that can be found there. These include core functions of Nλ such `map`, `filter` and `sum` and functions defined from them as in Section 3. One can find auxiliary functions to deal with pairs, triples or in general tuples and lists of set elements.

Notable omissions among functions provided by `Data.Set` are those that rely on an ordering of set elements, such as `elemAt`, `toList` but also `foldl` and `foldr`. There seems to be no meaningful way to interpret these functions on infinite, definable sets.

One additional function that *is* provided calculates the size of a set:

```
size :: NominalType a => Set a -> Variants Int
```

Certainly one can expect the answer in finite time only for finite sets. This function for consecutive natural numbers tries to find a list of distinct elements with a given length. This procedure is rather inefficient for large sets and does not terminate for infinite ones.

Hulls, supports and orbits

As mentioned in Section 5 and as will become apparent in Section 7 sometimes it is useful to the programmer to be able to operate on orbits of definable sets. For this purpose, functions listed in Section 5 have been added to the language. The implementation of all these functions is derived from two basic ones:

```
support :: NominalType a => a -> [Atom]
groupAction :: NominalType a => (Atom -> Atom) -> a -> a
```

The first returns a list of free atom variables in the argument (this list also serves as a support of it), the second applies a function to all free atom variables. Both functions invoke functions `foldVariables` and `mapVariables` that must be provided in instances of the `NominalType` class.

Based on `support` and `groupAction` we implement the function

```
orbit :: NominalType a => [Atom] -> a -> Set a
```

which computes the orbit of an element e under the action of all automorphisms of $\mathscr{A}$ that fix all elements of a given support $[a_1,\ldots,a_n]$. This function computes the list of free atoms $[b_1,\ldots,b_k]$ in e, and filters all lists of atoms of length k:

$$\{[x_1,\ldots,x_k] : \text{ for } x_1,\ldots,x_k \in \mathbb{A}\}$$

to obtain only these in the same orbit as $[b_1,\ldots,b_k]$. To this end, a conjunction formula is built as follows:

$$\bigwedge_{\substack{1\le i,j\le k\\ i\neq j}} r(x_i,x_j) \iff r(b_i,b_j) \ \wedge \ \bigwedge_{\substack{1\le i\le k\\ 1\le j\le n}} r(x_i,a_j) \iff r(b_i,a_j)$$

for every relation r in the signature of $\mathscr{A}$. (For equality atoms, it is just the equality relation.) In the last step, the filtered set of lists is mapped with a function that replaces every atom b_i in the element e by x_i for $1 \le i \le k$.

Using `orbit` an implementation of `hull` and other functions listed in Section 5 is now easy, for example:

```
hull :: NominalType a => [Atom] -> Set a -> Set a
hull supp = sum . map (orbit supp)
```

7 Examples

We demonstrate the potential and limitations of Nλ on two simple examples: computing transitive closures of relations and graph k-colorability. Although both examples can be implemented in Nλ, they are rather different. In the former one, standard Haskell code for calculating transitive closures of finite relations can be reused almost verbatim for the first-order definable case, sparing the programmer from considerations regarding finite vs. infinite sets. In the latter example, standard Haskell code for finding k-colorings in finite graphs does not transport to the infinite setting. Instead, one partitions a given graph into its orbits, and looks for an *equivariant* coloring, where all nodes in the same orbit get the same color. Both the program and the proof of its correctness depend on the programmer's knowledge of first-order definable sets and their mathematical theory.

Transitive closures and cycles

We begin by recalling the example presented in Section 1. To compute the composition of two relations one can define a function compose as follows:

```
compose :: (NominalType a, NominalType b, NominalType c) =>
           Set (a,b) -> Set (b,c) -> Set (a,c)
compose r s = sum (map (\(a,b) ->
                         map (\(_,c) -> (a,c))
                             (filter (eq b . fst) s))
                      r)
```

This function can be written down more concisely, using some auxiliary functions. In Nλ we provide some functions similar to the standard Haskell zip and zipWith:

```
pairs :: (NominalType a, NominalType b) => Set a -> Set b -> Set (a, b)
pairsWith :: (NominalType a, NominalType b, NominalType c) =>
             (a -> b -> c) -> Set a -> Set b -> Set c
```

There are also functions that help filtering pairs:

```
pairsWithFilter :: (NominalType c, NominalType b, NominalType a) =>
     (a -> b -> NominalMaybe c) -> Set a -> Set b -> Set c
maybeIf :: Ord a => Formula -> a -> NominalMaybe a
```

Using these one can implement compose in a single line:

```
compose r s = pairsWithFilter (\(a, b) (c, d) -> maybeIf (eq b c) (a, d)) r s
```

Now, one can code a function transitiveClosure computing the transitive closure of a given relation:

```
transitiveClosure :: NominalType a => Set (a,a) -> Set (a,a)
transitiveClosure r = let r' = union r (compose r r)
                      in ite (eq r r') r (transitiveClosure r')
```

It should be noted that the implementation of compose and transitiveClosure is similar to the finite version with only two differences: eq instead of (==) and ite instead of an if...then...else... statement.

Consider a datatype that describes directed graphs with vertices of any type and edges represented as pairs of vertices:

```
data Graph a = Graph {vertices :: Set a, edges :: Set (a,a)}
```

To check whether a graph has a cycle one could use the function transitiveClosure in the following way:

```
hasCycle :: NominalType a => Graph a -> Formula
hasCycle (Graph vs es) = exists (uncurry eq) (transitiveClosure es)
```

When only odd-length cycles are requested, one could define a function hasOddLengthCycle as presented below:

```
hasOddLengthCycle :: NominalType a => Graph a -> Formula
hasOddLengthCycle (Graph vs es) = intersect (map swap es)
                                            (transitiveClosure (compose es es))
```

where `(transitiveClosure (compose es es))` returns the set of all pairs of vertices connected with even-length paths. If some pair of vertices from this set is also connected with an edge from the original graph, it means that there is an odd-length cycle.

Note how the above fragments of code are essentially the same as ones that would be used for computing transitive closures or cycle finding on finite graphs.

Graph coloring

Recall that a graph coloring is a valuation of its nodes such that no two adjacent vertices share the same value. The verification whether a given function is a valid coloring looks as follows:

```
isColoringOf :: (NominalType a,NominalType b) => (a -> b) -> Graph a -> Formula
isColoringOf c g = forAll (\(v1,v2) -> c v1 ‘neq‘ c v2) (edges g)
```

A *k*-coloring is a graph coloring with *k* colors. In order to check whether a graph is *k*-colorable in the finite setting, one could generate all *k*-partitions of a set of *n* vertices:

```
partitions :: Int -> Int -> Set [Int]
partitions n 1 = singleton (replicate n 0)
partitions n k | k < 1 || n < k = empty
partitions n k | n == k = singleton [0..n-1]
partitions n k = union (map (k-1:) $ partitions (n-1) (k-1))
                       (pairsWith (:) (fromList [0..k-1]) (partitions (n-1) k))
```

For example, (`partitions` 3 2) evaluates to a set of three partitions: {`[0,0,1], [1,0,0], [1,0,1]`}. For each such partition one could examine if the valuation that arises from it is a valid coloring.

In the world of definable sets the situation is much more complicated. One cannot enumerate and collect all partitions because the set of partitions of a definable set might not be first-order definable or even countable. Indeed, at first sight it is not clear that colorability of definable graphs is a decidable problem. For example, consider the undirected graph:

$$\begin{aligned}\texttt{Graph}\ \{&\texttt{vertices} = \{(a_1,a_2) : a_1 \neq a_2 \text{ for } a_1,a_2 \in \mathbb{A}\},\\ &\texttt{edges} = \{\{(a_1,a_2),(a_2,a_3)\} : a_1 \neq a_2 \wedge a_1 \neq a_3 \wedge a_2 \neq a_3 \text{ for } a_1,a_2,a_3 \in \mathbb{A}\}\}\end{aligned} \tag{23}$$

This graph, used as an example in [9], is not 3-colorable. However, its smallest finite non-3-colorable graph has as many as 10 vertices and 20 edges. One may try to check larger and larger finite subgraphs of a given definable graph and check their colorability using the standard code above, but it is not clear when one can stop and declare the entire graph colorable.

One may make some additional assumptions, for example consider only *equivariant* colorings, where nodes in the same orbit must get the same color. (For example, the graph in (23) has no equivariant colorings, as it only has one orbit of vertices and it has edges.) The problem then reduces to coloring the finite set of orbits. For a given list of orbits and a list of its partitions one can create a coloring function that determines which orbit contains a given element and returns the color assigned to such an orbit.

```
coloring :: NominalType a => [Set a] -> [Int] -> a -> Variants Int
coloring [] [] _ = variant 0
coloring (o:os) (p:ps) a = ite (member a o) (variant p) (coloring os ps a)
```

Then it remains to check whether a coloring function created by a partition of orbits is a proper coloring of the graph. This can be implemented as follows:

```
hasEquivariantColoring :: NominalType a => Graph a -> Int -> Formula
hasEquivariantColoring g k = member true $
    pairsWith (\os ps -> (coloring os ps) ‘isColoringOf‘ g)
              (replicateSet n orbits)
              (partitions n k)
    where orbits = setOrbits (vertices g)
          n = maxSize orbits
```

where `replicateSet :: NominalType a => Int -> Set a -> Set [a]` returns the set of lists with a given length and elements from a set.

This solves the problem of finding equivariant colorings of definable graphs. As it turns out it solves the problem of general k-colorability as well: in [9], it was proved that *over ordered atoms* a definable graph has a k-coloring if and only it has an equivariant one. That result relies on deep theorems in topological dynamics. As we can see, the programmer needs to know the mathematics of first-order definable structures not only to write the program for k-colorability, but even more so to prove its correctness.

It is worth noting that the problem of finding an equivariant k-coloring may have different solutions depending on the structure of atoms. For example, the graph:

$$\begin{aligned}\texttt{g} = \texttt{Graph}\,\{\texttt{vertices} &= \{(a_1,a_2) : a_1 \neq a_2 \text{ for } a_1,a_2 \in \mathbb{A}\},\\ \texttt{edges} &= \{((a_1,a_2),(a_2,a_1)) : a_1 \neq a_2 \text{ for } a_1,a_2 \in \mathbb{A}\}\}\end{aligned}$$

does not have an equivariant 2-coloring when equality atoms are considered. But for ordered atoms, a function `(uncurry lt)` with type: `(Atom, Atom) -> Formula` is a correct coloring. So for these two structures of atoms the expression `(hasEquivariantColoring g 2)` will evaluate to `false` and `true` respectively.

Note that 2-colorings can be looked for in a way very similar to the one used for finite graphs; indeed, a graph is 2-colorable if and only if it has no cycle of odd length, and an Nλ program to check that was shown above. The expression `(hasOddLengthCycle g)` will evaluate to `false` both over equality and ordered atoms, indicating that a 2-coloring (not necessarily equivariant) of g exists.

These are only selected examples of programs in Nλ. We have also solved problems such as reachability, finding weakly or strongly connected components in graphs, the emptiness problem of automata [3] and a minimization algorithm of automata. None of these require the programmer to explicitly use orbits and other structure of definable sets. However, as the example of graph k-coloring (for $k > 2$) shows, certain problems do seem to require that. We do not understand precisely what it means for a problem to “require the use of orbits” or where the division lies between problems that do or do not. A possible connection to descriptive complexity theory and the celebrated “quest for PTIME logic” [6] could be imagined but this is left for future work.

References

[1] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The SMT-LIB Standard: Version 2.0*. Technical Report, University of Iowa.

[2] Mikołaj Bojańczyk, Laurent Braud, Bartek Klin & Sławomir Lasota (2012): *Towards nominal computation*. In: *Procs. POPL 2012*, pp. 401–412, doi:10.1145/2103656.2103704.

[3] Mikołaj Bojańczyk, Bartek Klin & Sławomir Lasota (2014): *Automata theory in nominal sets*. *Log. Meth. Comp. Sci.* 10, doi:10.2168/LMCS-10(3:4)2014.

[4] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Procs. of TACAS'08*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[5] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and Types*. Cambridge University Press.

[6] Martin Grohe (2008): *The quest for a logic capturing PTIME*. In: *Procs. LICS'08*, pp. 267–271, doi:10.1109/LICS.2008.11.

[7] Wilfried Hodges (1993): *Model theory*. Cambridge University Press, doi:10.1017/CBO9780511551574.

[8] Jan Hubička & Jaroslav Nešetřil (2005): *Universal partial order represented by means of oriented trees and other simple graphs*. *European Journal of Combinatorics* 26, pp. 765–778, doi:10.1016/j.ejc.2004.01.008.

[9] Bartek Klin, Eryk Kopczyński, Joanna Ochremiak & Szymon Toruńczyk (2015): *Locally Finite Constraint Satisfaction Problems*. In: *Procs. LICS 2015*, pp. 475–486, doi:10.1109/LICS.2015.51.

[10] Eryk Kopczyński & Szymon Toruńczyk: *Looping over infinite sets*. To appear.

[11] Rüdiger Loos & Volker Weispfenning (1993): *Applying Linear Quantifier Elimination*. *The Computer Journal* 36(5), pp. 450–462, doi:10.1093/comjnl/36.5.450.

[12] Tobias Nipkow (2008): *Linear Quantifier Elimination*. In Alessandro Armando, Peter Baumgartner & Gilles Dowek, editors: *Automated Reasoning*, *Lecture Notes in Computer Science* 5195, Springer, pp. 18–33, doi:10.1007/978-3-540-71070-7_3.

[13] *Nλ*. Available from `http://www.mimuw.edu.pl/~szynwelski/nlambda/`.

[14] Joanna Ochremiak (2016): *Extended constraint satisfaction problems*. Ph.D. thesis, University of Warsaw.

[15] Andrew M. Pitts (2013): *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, doi:10.1017/CBO9781139084673.

[16] F. Rossi, P. van Beek & T. Walsh, editors (2006): *Handbook of Constraint Programming*. Elsevier.

[17] J. T. Schwartz, R. B. Dewar, E. Schonberg & E. Dubinsky (1986): *Programming with Sets; an Introduction to SETL*. Springer-Verlag.

[18] Mark R. Shinwell (2006): *Fresh O'Caml: Nominal Abstract Syntax for the Masses*. *Electr. Notes Theor. Comput. Sci.* 148(2), pp. 53–77, doi:10.1016/j.entcs.2005.11.040.

[19] M. Takahashi (1995): *Parallel reductions in λ-calculus*. *Information and Computation* 118(1), pp. 120 – 127, doi:10.1006/inco.1995.1057.