Nondeterminism vs. Underspecification

Michał Walicki
Dept. of Informatics, University of Bergen, Bergen, Norway. michal@ii.uib.no. *

Sigurd Meldal
Dept. of CS, CalPoly, San Luis Obispo, CA, USA. smeldal@csc.calpoly.edu

ABSTRACT

The paper illustrates how nondeterministic operators supply a useful, and even *necessary*, abstraction mechanism, and how the common alternative, *underspecification* is insufficient even in quite simple examples. We show how underspecification actually forces one to *overspecify* by requiring additional assumptions not implied by the statement of the problem to be described. We indicate the tight relation between inherently unordered data types, like sets, and the nondeterminisitc (choice) operations which provide natural and adequate means for iteration over such data types.

Keywords: abstract data types, specification, nondeterminism, underspecification, iteration over sets.

1 INTRODUCTION

One of the aims of specifying a program is to express its functionality at a proper abstraction level, avoiding complications more properly addressed at a later stage in the operational and implementation refinement process. Abstract properties of the intended structure may (and should) allow different implementations, and one should avoid overspecifying the problems as this may exclude some, otherwise acceptable, attractive or even optimal, implementations.

The traditional means of abstraction in algebraic specifications is the use of *under*specification, which is often confused with nondeterminism or else meant to be a sufficient surrogate. We present the argument motivating the use of nondeterminism in specification, even when the systems (datastructures, algorithms) specified are intended to be implemented deterministically. An example will illustrate how nondeterminism may be an appropriate abstraction mechanism in writing a specification, and where underspecification cannot achieve the same (and reasonable) goals. Roughly, the example used to argue the case boils down to the observation that unordered structures (in particular *sets*) are very common and often very convenient abstract data structures – both in specification and programming. Equally common is the need for an *iteration* mechanism over such structures. But (deterministic) iteration introduces implicit ordering! We show by means of an example that the correct iterator over unordered structures should be specified as nondeterministic choice and that iteration along some implicitly assumed ordering, in spite of apparent plausibility, can easily lead to unintended results, collapsing otherwise non-equal data structures into identity.

As a vehicle for exploring these issues we shall make use of a choice operator $\sqcup$. Its intuitive meaning is simple: The result of $a \sqcup b$ is either a or b – no more, no less. In general, $\sqcup(S)$ should denote *some* member of S, and we do not care which one. This should allow us to discuss whether $c = a \sqcup b$ is valid or not, the consequences of introducing the equation as part of a specification, etc. Since the notion of *choice* is such a simple and intuitive one, we are striving for a semantic system which allows us to state its properties (and consequences of use) in a similarly straightforward and uncomplicated manner.

The presentation focuses on the concepts and general phenomena rather than technicalities. More technical details can be found in [7, 5]. In particular, the issue of formal reasoning with the specifications of nondeterminsm is not addressed at all – the reader is referred to [6, 1].

2 UNDERSPECIFICATION

The choice operation may serve as a (paradigmatic) example: assume that we want an operation $\sqcup : E \times E \to E$ which, for arguments x, y of a given sort E, should equal either x or y. A possible underspecification might be:

$$\begin{array}{llrrcl} SP = & \textbf{sorts}: & E & & & \\ & \textbf{ops}: & a, b: & & \to & E \\ & & \sqcup: & E \times E & \to & E \\ & \textbf{ax}.: & 1. & x \sqcup x & = & x \\ & & 2. & x \sqcup y & = & y \sqcup x \\ & & 3. & x \sqcup (y \sqcup z) & = & (x \sqcup y) \sqcup z \end{array}$$

*The author gratefully acknowledges the financial support received from the Norwegian Research Council, NFR.

where a, b are the two intended elements of sort E. Axioms 1.-3. intend to assert the general properties of $\sqcup$ as a join operator. The underspecification results in several nonisomorphic models for the specification, and the collection of models is taken as the ("loose") semantics of the specification.

However, SP is all too *loose* underspecification and $\sqcup$ has only a remote resemblance to the choice operation. Although SP will allow models where, for instance, $a \sqcup b = a$, and others where $a \sqcup b = b$, it will also allow plenty others where the result of $a \sqcup b$ is in no way related to a nor to b. To ensure more adequate properties of the intended choice operation, one would have to step a bit outside the tradition of using merely conditional equations and use disjunction to say something like:[1]

$$4. \quad x \sqcup y = x \quad \vee \quad x \sqcup y = y$$

Now, indeed, in every model of SP we will have either $a \sqcup b = a$ or $a \sqcup b = b$ (possibly also $a = b$). The point is, however, that in every model $\sqcup$ will be a deterministic operation, returning a result uniquely determined by the arguments. Thus, choosing a particular model as a basis for implementation, we will have to decide, whether to implement $a \sqcup b$ so that it always yields a, or else so that it always yields b. This may look innocent enough but it may actually have rather disastrous consequences. We will now give an example of possible problems.

Consider *graphs* as an example of an eminently practical abstraction. Graphs are typically presented as sets of vertices V and sets of edges G, and graph algorithms are rife with expressions such as "for each edge e from a vertex v do ...". The algorithms usually do *not* depend on the order in which the set of edges are traversed, leaving this choice to the implementor of either the set abstraction or the graph abstraction.

As an example, let us consider the problem of constructing the *depth-first traversal tree* of nodes reachable from a particular node of a directed, irreflexive graph. We assume that a (connected) graph is represented as a *set* G of edges, each edge being a pair $\langle x, y \rangle$ of the source and target vertices. (The set of all vertices in a graph can be easily obtained from G, assuming, for simplicity, that there are no isolated vertices.) Assume further that the function $N : Gr \times V \to \mathcal{P}(V)$ returns, for given a graph G and a vertex v in G, the set of v's neighbours in G.

Working with sets, we also assume some standard specification thereof, in particular, operations *empty set*, $\emptyset : \to \mathcal{P}(E)$, *adding* an element to a set, $+ : \mathcal{P}(E) \times E \to \mathcal{P}(E)$ and *removing* an element $\setminus : \mathcal{P}(E) \times E \to \mathcal{P}(E)$. In particular, the set constructor $+$ will satisfy the standard axioms:

$$\begin{array}{llcl} s1. & (S+x)+y & = & (S+y)+x \\ s2. & (S+x)+x & = & S+x \end{array}$$

ensuring that the order of insertions (s1) and the multiple insertions of the same element (s2) do not matter for the identity of sets.

A textbook presentation of the DFS algorithm is the following pair of procedures:

```
DFS (G, v) is
begin
  T := ∅;
  Traverse (G, v, T);
  return T;
end;
```

```
Traverse (G, v, T) is
begin
  mark v;
  for all edges ⟨v, x⟩ do
    if x is unmarked then
      T := T + ⟨v, x⟩;
      Traverse (G, x, T);
    end if;
  end for;
  return T;
end Traverse;
```

The DFS algorithm as stated is simple, and we should hope for a similar simplicity and level of abstraction when we give an algebraic specification for it. (Note in particular the use of the set iterator '**for all** edges ...' in the loop of *Traverse*.)

The following attempts to give an algebraic version of the specification of this standard algorithm for $DFS(G, v)$ starting from the vertex v in a graph G and constructing the DFS-tree by recursive calls to the operation $Tr : Gr \times V \times Gr \to Gr$, where the second Gr-argument is a set of edges representing the tree (graph) constructed so far.

$$\begin{array}{lrcl}
1. & & & DFS(G,v) = Tr(G,v,\emptyset) \\
2. & N(G,v) = \emptyset & \Rightarrow & Tr(G,v,T) = T \\
3. & N(G,v) = S + x, x \in |T| & \Rightarrow & Tr(G,v,T) = \\
 & & & Tr(G \setminus \langle v,x \rangle, v, T) \\
4. & N(G,v) = S + x, x \notin |T| & \Rightarrow & Tr(G,v,T) = \\
 & \multicolumn{3}{r}{Tr(G \setminus \langle v,x \rangle, v,\ Tr(G \setminus \langle v,x \rangle, x, T + \langle v,x \rangle)\ )}
\end{array}$$

The construction starts with the empty tree – axiom 1. Tr backtracks, returning the tree T constructed so far, if v has no neighbours (either because all were visited (marked by removing them from G) or because v is a leaf) – axiom 2. If the set $N(G,v)$ is not empty, i.e., it equals $S + x$ for some set S and vertex x, then there are two possibilities for each element of this set. If v's neighbour x is marked (indicated by $x \in |T|$, i.e., x is among the vertices included in (the edges of) T) then the edge $\langle v, x \rangle$ is simply removed from further consideration – axiom 3. If, on the other hand, the neighbour x was not visited before, then the edge $\langle v, x \rangle$ is added to T as well as being removed from G, and the construction proceeds recursively down through x – axiom 4.

[1] We do not focus here on the existence of initial models which is the traditional reason for restricting the language to conditional equations and excluding disjunctions. We are merely interested in the difference between underspecification and nondeterminism.

This should correspond to the intuitive understanding of the algorithm and may look plausible – but only as long as we do not inspect the definition of the sort of sets. The unordered structure of sets is expressed, at least, by axioms s1 and s2. In particular, s1 implies for any graph G' and edges $\langle v, a\rangle, \langle v, b\rangle$ the following equality:

$$(G' + \langle v,a\rangle) + \langle v,b\rangle \;=\; (G' + \langle v,b\rangle) + \langle v,a\rangle \qquad (1)$$

Taking as G' a single edge $\langle a, b\rangle$, the above expressions denote then the graph $G = \begin{tikzcd}[sep=small] & v \arrow[dl, dash] \arrow[dr, dash] & \\ a \arrow[rr] & & b \end{tikzcd}$. The set of v's neighbours, $N(G, v)$, can be written equivalently, again using s1, as:

$$(\emptyset + a) + b \;=\; N(G,v) \;=\; (\emptyset + b) + a \qquad (2)$$

But now, applying the axioms 1.-4. (in particular axiom 4.) to both these expressions, will yield an identification of two trees which nobody would like to say are the same:

$$\begin{tikzcd}[sep=small] & v \arrow[dl, dash] \arrow[dr, dash] & \\ a & & b \end{tikzcd} \;=\; Tr(\begin{tikzcd}[sep=small] & v \arrow[dl, dash] \arrow[dr, dash] & \\ a \arrow[rr] & & b \end{tikzcd}, v, \emptyset) \;=\; \begin{tikzcd}[sep=small] & v \arrow[dl, dash] & \\ a \arrow[rr] & & b \end{tikzcd} \qquad (3)$$

The problem is that the internal structure of the set values intrudes into the specification. The definition of Tr in terms of adding elements to a set tries, on the one hand, to respect the representation of graphs in terms of sets (of neighbours of each vertex) and, on the other hand, use the ordering of sets, implicit in their definition by means of adding elements. One might avoid the problem by choosing a *particular set representation* when specifying the DFS function, e.g., letting the neighbours $N(G, v)$ be represented as a sequence instead of as a set. However, this is quite contrary to the central tenet of (algebraic) specifications – it forces us into making representational or operational decisions when there is no conceptual need to do so: our abstract specification formalism forces us thus to abandon a key abstraction!

A more general statement of the problem would be to say that the means of abstraction offered by underspecification are insufficient (or, in any case, inadequate) to describe algorithms. An algorithm is to be understood here as a procedure yielding *a* result satisfying some definite property but which is not necessarily unique (viz. *a* DFS tree).

3 NONDETERMINISM

The DFS specification is by no means an artificial example. It illustrates a very common case of the need to pass from an abstract (which in this context means intuitive) description in terms of sets or, generally, unordered structures, to a more detailed description in terms of constructs available in a programming language presupposing always an ordering of data. Consider for instance a classical example: an abstract datatype *set,* with a choice operation. A natural implementation of such a datatype would be to represent sets as *lists*, and let the choice operation be implemented as the head function. However, in [4] it was shown how such an implementation would violate even the most minimal of *set* specifications, and how the problem finds a natural resolution by allowing for nondeterminism.

We will now sketch the main idea of the solution to the DFS problem using nondeterminism. A necessary generalization of the classical framework is to replace the algebras used as models of a specification by multialgebras, where nondeterministic operations are modeled by set-valued functions.

A *multialgebra* (over some signature Σ) is a Σ-algebra where operations return sets (possibly singleton, or even empty set) of elements. We introduce two primitive predicates for building atomic formulae: $s \doteq t$ (element identity – *equality*) requires that s and t denote the same, *one-element* set, and $s \prec t$ (*inclusion*) that the set interpreting s is included in that interpreting t. A Σ-multialgebra A is given by:

1. for each sort symbol S, the carrier set S^A
2. for each operation symbol $f : S_1 ... S_n \to S$, an operation $f^A : S_1^S ... S_n^A \to \mathcal{P}(S^A)$

where $\mathcal{P}(X)$ denotes the set of subsets of X. 2. makes it possible for a constant $c : \to S$ to denote a *set* of elements from S^A.

Given a Σ-term t (with variables X), any assignment $\alpha : X \to A$ of *individual* elements from the carrier of A to X, induces a unique interpretation $\overline{\alpha}(t)$ of t in A. Variables are assigned *individuals* and not sets.[2]

The formulae are sequents $a_1, ..., a_n \mapsto b_1, ..., b_m$, with a_i and b_j being atomic (equalities or inclusions). A multialgebra A satisfies such a formula ϕ under assignment α

3. $A \models_\alpha s \doteq t$ iff $\overline{\alpha}(s) = \overline{\alpha}(t) = \{e\}$ for some $e \in A$;
4. $A \models_\alpha s \prec t$ iff $\overline{\alpha}(s) \subseteq \overline{\alpha}(t)$;
5. $A \models_\alpha a_1, ..., a_n \Rightarrow b_1, ..., b_m$ iff $\exists 1 \leq i \leq n : A \not\models a_i$ or $\exists 1 \leq j \leq m : A \models b_m$.

As usual, $A \models \phi$ iff $A \models_\alpha \phi$ for all α. According to 3. $s \doteq t$ holds iff both terms denote the same 1-element set and not just the same set. Now, consider

[2] Motivations for these choices can be found in [7, 4, 6].

the following specification

$$
\begin{array}{llllll}
SPN = & \textbf{sorts}: & E & & & \\
& \textbf{ops}: & a, b, c: & & \to E & \\
& & \quad \sqcup: & E \times E & \to E & \\
& \textbf{ax}.: & 1. & z \prec x \sqcup y & \Rightarrow & z \doteq x \vee z \doteq y \\
& & 2. & c & \prec & a \sqcup b \\
& & 3. & a & \doteq & a \\
& & 4. & b & \doteq & b \\
& & 5. & c & \doteq & c
\end{array}
$$

3.-5. merely ensure that the constants, in particular c, are deterministic, i.e., denote individual elements. Note the difference between c and z. According to 5. c is *deterministic*, but *underspecified* according to 2. – in any model c will denote one element, either the same as denoted by a or the one denoted by b. The operation $x \sqcup y$, on the other hand, is nondeterministic and returns, according to 1., either x or y. The variable z is used to underspecify not the operation $\sqcup$ but any of its applications. The axiom 1. reads: whenever z is a result of $x \sqcup y$, then *either* $z \doteq x$ *or* $z \doteq y$. Thus, different applications of $\sqcup$ may return different results, and the equality $x \sqcup y \doteq x \sqcup y$ does *not* hold. Adding two axioms $x \prec x \sqcup y$ and $y \prec x \sqcup y$ would make this nondeterministic operation completely specified: in every model M, the interpretation of $x \sqcup^M y$ would be precisely the set $\{x, y\}$.

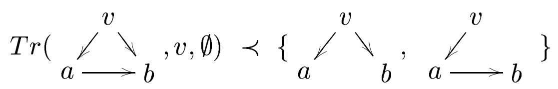
Now, an abstract definition of DFS should *not* assume any particular representation of the sets of neighbours. It also should *not* force the implementor to construct a *particular* DFS-tree but, on the contrary, determine only the necessary requirements, leaving the choice of a particular strategy (here particular way of ordering a set) to the implementor.

This objective can be achieved by the following, nondeterministic variant of the earlier specification. We assume the same representation of graphs and specification of sets as before. In addition, we assume a nondeterministic choice operation picking arbitrary element from a set $\sqcup : S \to E$. There are a number of ways to specify choice, for instance, by an axiom $z \prec \sqcup.(S + x) \Rightarrow (z \doteq x) \vee (z \prec \sqcup.S)$, or, perhaps, $S \doteq \emptyset \vee \sqcup(S) \in S$. The former can be read as "If we add an element x to a set S, then applying the choice operator to the resulting set we either get x or we get one of the values possible when applying choice to S." Irrespectively of the details of the specification of choice, the main specification is now as follows:

$$
\begin{array}{lrcl}
1. & DFS(G, v) & \prec & Tr(G, v, \emptyset) \\
2. & N(G, v) \doteq \emptyset & \Rightarrow & Tr(G, v, T) \doteq T \\
3. & x \prec \sqcup(N(G, v)), x \in |T| & \Rightarrow & Tr(G, v, T) \prec \\
& & & Tr(G \setminus \langle v, x \rangle, v, T) \\
4. & x \prec \sqcup(N(G, v)), x \notin |T| & \Rightarrow & Tr(G, v, T) \prec \\
& \multicolumn{3}{r}{Tr(G \setminus \langle v, x \rangle, v,\ Tr(G \setminus \langle v, x \rangle, x, T + \langle v, x \rangle)\ )}
\end{array}
$$

Axiom 1. indicates that DFS will return one of the trees Tr is capable of returning. Axiom 2. takes care of the backtracking case (which is deterministic). In axioms 3. and 4. one picks an arbitrary neighbour, and requires that Tr must allow for the edge to that neighbour being the next one to traverse. But the specification does not *require* that particular neighbour to be the next (except if it is the *only* neighbour, of course).

The operation Tr is now allowed to return any (all) DFS-trees – depending on the choice of x from the set $N(G, v)$ at each stage (axioms 3 and 4). Instead of the unintended collapse of different trees as in (3), we will now have:



i.e., the set of possible results of applying Tr to the graph is constrained by the set of two trees on the right-hand side of the inclusion.

Nondeterministic choice is *the* iterator for sets

As our DFS example illustrates the problem concerns the way of describing iteration over unordered structures, here sets. The deterministic underspecification forces one to assume some kind of ordering of such structures whenever one wants to iterate over them. This, as we pointed out, amounts actually to *over-specification* because, typically, there is no need for assuming such an ordering. The problem described in 2 signals the necessity of such an asumption – it was the attempt to treat sets as unordered (according to axiom s1) which caused the unintended identification of different result trees.

Our solution using the nondeterministic choice indicates that this operation provides the adequate notion of an iterator over unordered structures. It is adequate because yielding the correct results, it also preserves the intended level of abstraction. Indeed, having no ordering, it should be even intuitively clear that the only way of selecting elements is by picking them arbitrarily. The importance of our approach consists in that it offers the means for both modelling such operations and reasoning about them. (Although we do not discuss reasoning systems here, the reader may consult [6, 5, 1].) This reasoning preserves then the appropriate level of abstraction addressing only the properties which emerge under arbitrary choices or else under arbitrary deterministic implementations.

4 CONCLUSIONS

We have argued that nondeterminism provides adequate means for specifying operations on inherently unordered structures. In particular, nondeterministic choice is the natural operation for performing iteration over sets. In such context, deterministic specifications are bound to overspecify the operations and nondeterminism is simply the adequate abstraction mechanism.

The traditional abstraction mechanism – underspecification – is different from nondeterminism and cannot perform this function. It amounts to underspecifying a given operation requiring, nevertheless, its result to be unique in every model. We have shown its failure in capturing the appropriate level of abstraction. Nondeterminism, on the other hand, allows one operation to return different results within the same model. It can be thought of as underspecification not of an operation itself but of its particular applications.

Sets or, generally, unordered structures are most natural abstractions and most programming languages provide means for defining such data types. We suggest that this should be accompanied by the presence of a nondeterministic choice operator, unless one wants to design explicitly iterators for all such structures. But even if one does not include this operator into the programming language, it still has its place in defining the semantics for such a language which should be independent from any possible and particular implementation of the unordered data types.

Nondeterminism is a powerful abstraction mechanism which, actually, allows one to write programs and reason about them in a more intuitive way. This power of abstraction allows one also to model in a uniform way, except nondeterminism itself, a wide range of phenomena like partiality, parameterized data types, behavioural abstraction [6, 4, 2, 3].

References

[1] Valentinas Kriaučiukas and Michał Walicki. Reasoning and rewriting with set-relations ii: the non-ground case. In *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*. Springer, 1996.

[2] Yngve Lamo and Michał Walicki. Modeling partiality by nondeterminism – from abstract specifications to flexible error handling. Technical Report 178, Dept. of Informatics, University of Bergen, 1999.

[3] Yngve Lamo and Michał Walicki. Specification of parameterized programs: persistency revisited. *Nordic Journal of Computing*, 2001. [to appear].

[4] Michał Walicki and Manfed Broy. Structured specifications and implementation of nondeterminisitc data types. *Nordic Journal of Computing*, 2:358–395, 1995.

[5] Michał Walicki and Sigurd Meldal. A complete calculus for the multialgebraic and functional semantics of nondeterminism. *ACM TOPLAS*, 17(2), 1995.

[6] Michał Walicki and Sigurd Meldal. Multialgebras, power algebras and complete calculi of identities and inclusions. In *Recent Trends in Data Type Specification*, volume 906 of *Lecture Notes in Computer Science*. Springer, 1995.

[7] Michał Walicki and Sigurd Meldal. Algebraic approaches to nondeterminism-an overview. *ACM Computing Surveys*, 29, 1997.