Verifying and Generating WP Transformers for Procedures on Complex Data

Patrick Michel and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_michel,poetzsch}@cs.uni-kl.de

Abstract. We present the formalized theory of a weakest precondition calculus for procedures on complex data with integrity constraints. The theory defines the assertion language and the wp-transformer. It contains the proofs for soundness and "weakestness" of the preconditions. Furthermore, we formalize a normalization process that eliminates all elementary updates from preconditions. This *normalization* property is important for *efficient checking* of the preconditions in programs. The theory is completely realized in Isabelle/HOL and used for generating the Haskell implementation of the wp-transformer and the normalization process.
The wp generation is developed for procedures on complex data with integrity constraints, for example XML documents satisfying a schema. Efficient checkability allows maintaining the constraints with acceptable computing resources. It is a central motivation of our work and has influenced many design decisions.

Keywords: Interactive verification, Isabelle/HOL, structured data, paths, invariants, precondition generation, language semantics

1 Introduction

We present the formalized theory of a weakest precondition calculus for procedures on complex data with integrity constraints. The theory defines the assertion language and the wp-transformer. It contains the proofs for soundness and "weakestness" of the preconditions. Furthermore, we formalize a normalization process that eliminates all elementary updates from preconditions. This *normalization* property is important for *efficient checking* of the preconditions in programs. The theory is completely realized in Isabelle/HOL and used for generating the Haskell implementation of the wp-transformer and the normalization process.

Our main application area are procedures on complex structured data with integrity constraints, for example XML documents satisfying a schema. Such data is used in many different areas of computing, for example as objects satisfying a class invariant, as parameters of complex types used in web services, or as data stored in a database. Accordingly, the data should only be manipulated by procedures pc maintaining the integrity constraints. That is, the constraints play the role of data invariants C_{inv}. Each procedure consists of a sequence of elementary modifications of the data where the invariant might be violated in between. The goal is to compute preconditions $C_{pre}(pc)$ of these procedures, i.e., assertions ranging over the current data state and the parameters of pc, with the following properties:

In "ITP 2012" (LNCS 7406), http://dx.doi.org/10.1007/978-3-642-32347-8_10
The original publication is available at www.springerlink.com.

1. If $C_{pre}(pc) \wedge C_{inv}$ holds in the prestate of pc, then C_{inv} holds in the poststate.
2. $C_{pre}(pc)$ can be checked efficiently in a given prestate.

Efficient checkability has two aspects. First, $C_{pre}(pc)$ should avoid existential quantifiers and only use read operations on the complex data. Second, $C_{pre}(pc)$ should not express properties that are covered by the invariant, because this would lead to redundant checking. These properties allow us to efficiently maintain even complex and large invariants by only checking $C_{pre}(pc)$ whenever pc is called (cf. Sect. 2).

Related to our work are wp-transformers for programs with heap allocated data [12, 17, 8] and language settings designed for XML or tree updates [11, 2, 18, 3]. Whereas these approaches either result in *inefficient* preconditions or work only for *structural* constraints, we aim to generate efficiently checkable preconditions for complex data with integrity constraints going beyond structure. In particular, preconditions should be normalized such that they only read the data in the prestate and do not contain updates. The central contribution of this paper is a theory for a core wp-calculus satisfying the described requirements. It contains:

- A path-based representation of structured data together with a suitable assertion and update language
- A wp-transformer together with proofs that it is sound and produces weakest preconditions
- A normalization process for assertions
- A theory for infinite multisets and a three-valued logic as semantical foundation for the assertion language, the transformer, and the normalization
- A proof technique and concept for syntactic transformations under semantic equivalence, using explicit partiality and a concept of safe formulas

The theory is formalized in Isabelle/HOL [16] (over 9000 lines) and is used to generate Haskell programs for the wp-transformer and the normalization (about 2200 lines of generated Haskell code). This tool-based approach proved to be indispensable, in particular for the rather complex normalization process.

Overview. Section 2 explains our approach to complex data in more detail. Section 3 presents the data representation and the core assertion language. Sections 4 to 6 summarize and discuss important aspects of the wp-theory. Section 7 discusses related work and Sect. 8 concludes.

2 Approach

In this section, we describe and motivate our approach by two small examples. The first shows how the different aspects of the approach work together, focusing on preconditions. The second is about a more realistic data type. To illustrate that our approach is not tied to a specific language setting we use a programming language syntax for the first example and an XML-related syntax for the second.

Preconditions. By *complex data*, we refer to hierarchical data structures with integrity constraints going beyond purely structural schemas. As first example, we consider a record type `container` with two components: the component `items` is

an array of numbers storing for item i its weight; the component `owght` stores the overall sum of the weights of all items. As invariant of `containers` c, we have:

$$C_{inv} \equiv sum(c.\text{items}) = c.\text{owght} \ \wedge \ c.\text{items}[i] > 0 \ \wedge \ c.\text{owght} < 1000$$

where i quantifies over the defined indices of the array. A basic incremental procedure on containers could, for instance, modify the weight of an item:

```
PROC modifyWeight(container c, int ix, int wght)
  c.owght = c.owght - c.items[ix] + wght;
  c.items[ix] = wght;
```

Using the techniques from [12], the generated weakest precondition for C_{inv} is

$$sum(update(c.\text{items}, ix, wght)) = c.\text{owght} - c.\text{items}[ix] + wght \ \wedge$$
$$update(c.\text{items}, ix, wght)[i] > 0 \ \wedge \ c.\text{owght} - c.\text{items}[ix] + wght < 1000$$

where *update* describes an array update and where we ignore index out of bounds problems for simplicity. Even for this simple example, the generated precondition is not suitable for efficient checking. The update-operation causes unneccessary overhead and the precondition essentially forces us to check the invariant C_{inv} for the prestate although we may assume that it holds. A much nicer precondition avoiding updates and rechecking of the invariant would be:

$$wght > 0 \ \wedge \ c.\text{owght} - c.\text{items}[ix] + wght < 1000$$

Generating such preconditions for efficient checking is the central goal of wp-transformation with normalization. Our normalization technique eliminates all update-operations. Furthermore, in the normalized form, the invariant can be factored out from the precondition using regrouping and simplification. We focus on loop-free, atomic procedures for incremental updates on complex data. For details on this design decision and how our approach can be used in practice see [14, 15].

Schemas for Complex Data. To illustrate the kinds of complex data that our approach supports, we consider an extension of the container datatype above. We use a schema notation in the style of the XML schema language RelaxNG [10]. The (extended) container type has a capacity attribute, contains a collection of items and a collection of products. Collections are multisets of elements (or attributes), i.e., have unordered content; elements in collections are referenced by a *key* that has to be provided on entry. Each item in the extended container type is considered to be an instance of a specific product and refers to the product entry using the corresponding key. Products have a positive weight less or equal to the capacity of the container.

```
element container {
  attribute capacity { integer [. > 0] },
  [ ./capacity ≥ sum (./product[./item/productref]/weight) ]

  element item * {
    attribute productref { key [ //product[.] ∈ $ ] }
  },
  element product * {
    attribute weight { integer [. > 0] [. ≤ //capacity] }
} }
```

To formulate constraints, we use path-based expressions. The dot '.' refers to the value of the current attribute or element. The '$'-sign refers to the overall document. For example, the constraint `//product[.] ∈ $` of attribute `productref` states that there has to be a product with this key in the current document. The global constraint in line 3 enforces that the capacity of the container is larger or equal to the sum of the weights of all items; i.e., the expression `./product[./item/productref]/weight` represents the *multiset* of all paths obtained by placing a key at `./item/productref` into `./product[_]/weight`. A more detailed presentation of our data description approach is given in [15].

We used our approach to define the schema, integrity constraints, and procedures for the persistent data of a mid-size web application [13]. The definition of the schema and integrity constraints is about 100 lines long. We developed 57 basic procedures for the manipulation of the data.

3 Core Assertions

This section defines the syntax and semantics of our core assertion language for complex data structures. This language is the heart of our theory, as the choice of the core operators play a central role to achieve our goals. On the one hand, syntactic transformations and normalizations require operators with homomorphic and further semantical properties. On the other hand, to define useful invariants, to handle partiality of operators, to manage the wp-transformations, and to support certain normalization steps within the language, we need enough expressive power. The developed language is kind of a sweet spot between these partially conflicting requirements. It essentially follows the ideas of [5] (cf. Sect. 7). It is a core language in the sense that syntactical extensions and further operators having the required transformation properties can be added on top of the language.

Data representation in the assertion language is based on the concepts of paths and the idea that every basic element of a hierarchical data structure should be addressable by a unique path. A path is essentially a sequence of labels l, which can be thought of as element names in XML or attribute names in Java, etc. As inner nodes of a hierarchical data structure might represent collections, the path has to be complemented by a *key* for each step to select a unique element of the collection at each point. Following XML, we call a hierarchical data structure a *document*.

Definition 1 (Paths and Documents). *A **path** is a sequence of label-key pairs. The special key null is used for path steps which need no disambiguation. The sequence of labels of a path alone defines the so-called **kind** of the path. A **document** is a finite mapping from a prefixed-closed set of paths to values.*

Syntax and Semantics. Based on the concept of documents, we define the syntax of the assertion language as follows:

$$\begin{array}{rlrl}
\textbf{value expr.} & V ::= c \mid v \mid \$(P) \mid V_{SI} \odot V_I \mid V \oplus V \mid \mathit{sum}\ V_I \mid \mathit{size}\ V \mid \mathit{count}\ V\ V_S \mid \mathit{tally}\ T\ V & & \\
\textbf{path expr.} & P ::= \mathit{root} \mid P/l[V_K] \mid P/l & \textbf{types} & T ::= \mathit{key} \mid \mathit{int} \mid \mathit{string} \mid \mathit{unit} \\
\textbf{relations} & R ::= V_S = V_S \mid V_{SI} < V_{SI} & \textbf{disjunctions} & D ::= \mathit{false} \mid L \vee D \\
\textbf{literals} & L ::= R \mid \neg R & \textbf{conjunctions} & C ::= \mathit{true} \mid D \wedge C
\end{array}$$

Basic *values* are of type *key*, *int*, *string*, or *unit* where *unit* is a singleton type with the constant (). Expressions and assertions B are interpreted with respect to an environment E, denoted by $\| B \|_E$. The environment assigns keys to the variables and a document to the \$-symbol that refers to the document underlying the assertion.[1]

Path expressions denote the root node, select children by label-key pairs, or select all children for a label. The latter is called a *kind step*, as it selects all children of this particular kind. The semantic domain of path expressions are (possibly infinite) multisets of paths. The semantic equations are as follows:

$$\begin{aligned} \| \mathit{root} \|_E &= \mathit{root} \\ \| P/l[V_K] \|_E &= \{\, p/l[k] \mid p \leftarrow \| P \|_E,\ k \leftarrow \| V_K \|_E\ ,\ k \in \mathit{Univ}(\mathit{key}) \,\} \\ \| P/l \|_E &= \{\, p/l[k] \mid p \leftarrow \| P \|_E,\ k \leftarrow \mathit{Univ}(\mathit{key}) \,\} \end{aligned}$$

where we denote the universe of keys by $\mathit{Univ}(\mathit{key})$ and use multiset comprehensions to denote all paths $p/l[k]$ where p and k are generated from multisets.

Basic value expressions are constants c, such as $1, -1, 0$ and *null*, and variables v for keys. Variables are implicitly universally quantified at top level. To handle single values and multisets of values uniformly, we identify single values with the singleton multisets containing exactly that value and vice versa. The read expression $\$(P)$ selects all values from the underlying document \$ that are stored at paths P:

$$\| \$(P) \|_E = \{\, E(\$)(p) \mid p \leftarrow \| P \|_E,\ p \in \mathit{dom}\ E(\$) \,\}$$

On multisets, we provide four aggregate functions: $\mathit{sum}\ V_I$ returns the sum of all integer elements of V_I; $\mathit{size}\ V$ returns the number of elements in V; $\mathit{count}\ V\ V_S$ returns the number of occurrences of singleton V_S in V; $\mathit{tally}\ T\ V$ returns the number of occurrences of elements of type T in V. Furthermore, we support the scalar multiplication $\odot$ of a single integer with a multiset of integers and a union operation $\oplus$ for multisets with $\mathit{count}\ (V_1 \oplus V_2)\ V_S = \mathit{count}\ V_1\ V_S + \mathit{count}\ V_2\ V_S$.

The rest of the syntax defines boolean formulas in conjunctive normal form with the polymorphic equality of singleton values and the ordering relation on singleton integers. The constants *false* and *true* are used for empty disjunctions or empty conjunctions. To handle partiality, all semantic domains include the bottom element $\bot$. All operators except for $\vee$ and $\wedge$ are strict: $\vee$ evaluates to true if one of the operands evaluates to true; otherwise its evaluation is strict (similar for $\wedge$).

In summary, the assertion language allows us to formulate document properties. Every hierarchical document model, in which values are accessed using a path concept, can be supported by our approach. As the core syntax does not support document updates, checking whether an assertion holds for a given document only needs to read values in the document by following access paths.

Syntactic Extensions. We extended the core language in two ways. Firstly, we support assertions that are not in conjunctive normal form. These more general forms are automatically transformed back into the core syntax. Secondly, we added further operators and abbreviations. We have taken great care in the design of the operators and their theory to avoid large blow-ups of the formula size as a result of eliminating the new operators (for details, we refer to the accompanying Isabelle/HOL theory).

[1] The document corresponds to the state of the heap in assertions for programming languages.

In particular, we provide the following abbreviations and operators:

$$
\begin{array}{rcl@{\qquad}rcl}
\{\} & \equiv & 0 \odot null & empty\ V & \equiv & size\ V = 0 \\
\bot & \equiv & null \odot 0 & unique\ V & \equiv & size\ V = 1 \\
-V & \equiv & -1 \odot V & unique\ P & \equiv & \ldots \qquad count\ P_2\ P_1 \equiv \ldots \\
V_1 + V_2 & \equiv & sum\ (V_1 \oplus V_2) & P_1 \in P_2 & \equiv & count\ P_2\ P_1 > 0 \\
V_1 - V_2 & \equiv & V_1 + (-V_2) & P_1 \notin P_2 & \equiv & count\ P_2\ P_1 = 0 \\
V\ is\ T & \equiv & tally\ T\ V = size\ V & P \in \$ & \equiv & unique\ P \wedge unique\ \$(P) \\
V\ is - not\ T & \equiv & tally\ T\ V < size\ V & P \notin \$ & \equiv & unique\ P \wedge empty\ \$(P)
\end{array}
$$

The left column shows simple abbreviations for the empty multiset, bottom, an embedding of integer arithmetic, and type tests. The right column adds some predicates for value multisets, as well as containment relations for paths regarding path multisets and the document. *unique*(P) yields true if P evaluates to a singleton path; *count* denotes the count-operator on path expression (both operators are realized by syntactic transformation into a value expression of the core syntax).

4 WP-Transformer for Linear Programs

As explained in Sect. 2, the basic goal of our approach is to generate preconditions for procedures manipulating complex data or documents. The preconditions should be efficiently checkable and should guarantee that integrity constraints are maintained. This section defines the imperative language for document manipulation and the weakest precondition generation.

Language for document manipulation. Procedures have a name, a list of parameter declarations, and a statement of the following form as body:

$$
\begin{array}{c}
\textbf{statements}\ S ::= skip \mid S;S \mid p := V_S \mid if\ L\ then\ S\ else\ S\ fi \mid assert\ C \mid \\
insert\ P_S\ l\ V_{SK} \mid update\ P_S\ V_S \mid delete\ P_S
\end{array}
$$

The skip-statement and sequential statement composition are as usual. The assignment has a local variable or parameter on the left-hand side and a (singleton) value expression free of (logical) variables as right-hand side. The boolean expression in the if-statement is restricted to literals. The assert-statement uses arbitrary assertions and allows strengthening of the precondition which can simplify it. There are three basic update-operations for documents:

- *insert* $P_S\ l\ V_{SK}$ assumes that the singleton path P_S exists in \$, but not the singleton path $P_S/l[V_{SK}]$; it inserts path $P_S/l[V_{SK}]$ with default value ().
- *update* $P_S\ V_S$ changes the value at an existing singleton path P_S to the singleton value V_S. In combination with *insert*, it allows us to insert arbitrary paths and values into a document.
- *delete* P_S removes all values in the document associated with the existing singleton path P_S or any of its descendants.

We do not provide a loop-statement, because we want the wp-generation to be fully automatic and because loops are available in programs in which the procedures for document manipulation are called. The statements are designed (and proven) in such a way that they maintain the property that the set of paths in a document is prefix-closed. The semantics of statements is defined in big-step operational style:

Definition 2 (Semantics of Statements). *The semantics of statements is inductively defined as a judgment* $S, E_1 \rightsquigarrow E_2$. *The judgment holds if the execution of statement* S *starting in environment* E_1 *terminates, without errors, with environment* E_2. *The environments capture the state of the document, the parameters of the procedure, and the local program variables.*

Based on the semantics, we define Hoare-Triples for *total correctness*, i.e., if the precondition holds, no errors occur and a poststate exists. We have proven that statement execution is deterministic, i.e., the post environment is unique, if it exists.

Definition 3 (Hoare-Triple). *Hoare-Triples* $\{C\}\ S\ \{C\}$ *have the following semantics:*

$$\vDash \{C_P\}\ S\ \{C_Q\} \quad \equiv \quad \forall E.\ \| C_P \|_E \ \rightarrow\ \exists E'.\ S, E \rightsquigarrow E' \ \wedge\ \| C_Q \|_{E'}$$

WP generation. The central goal of WP generation is to take an assertion C_Q and a statement S and generate the weakest assertion C_P such that $\vDash \{C_P\}\ S\ \{C_Q\}$. Unfortunately, the restriction of the assertion language to read operations in documents that is profitable for efficient checking has its downsides when it comes to wp generation. The classical approach to handle compound data in wp-calculi is to move updates in the programming language into updates in the assertions (as demonstrated by array c.items in Sect. 2). We solve this problem in two steps: First, we make the assertion language more expressive by adding document update operators such that wp generation can be expressed following the classical approach. Second, we show how these operators can be eliminated by a normalization process (see Sect. 6).

Besides document updates, we introduce program variables p (parameters, local variables) into the value expressions:

$$\begin{aligned} &\textbf{value expr.}\ V ::= c \mid v \mid p \mid M(P) \mid \ldots \\ &\textbf{documents}\ M ::= \$ \mid M[P_S \mapsto V_S] \mid M[P_S \mapsto] \end{aligned}$$

$M[P_S \mapsto V_S]$ combines insert and update: If $P_S \in dom\ M$, the value at P_S is changed; otherwise a new binding is created. $M[P_S \mapsto]$ deletes all bindings of P_S and all its descendants. With these additions, the wp-transformer can be defined as follows.

Definition 4 (WP-Transformer).

$$\begin{aligned}
&wp\ skip\ C && = C \\
&wp\ (S_1; S_2)\ C && = wp\ S_1\ (wp\ S_2\ C) \\
&wp\ (delete\ P)\ C && = P \in \$ \ \wedge\ C[\$/\$[P \mapsto]] \\
&wp\ (insert\ P\ l\ V)\ C && = P \in \$ \ \wedge\ C[\$/\$[P/l[V] \mapsto ()]] \ \wedge\ P/l[V] \notin \$ \\
& && \quad \wedge\ unique\ V \ \wedge\ V\ is\ key \\
&wp\ (update\ P\ V)\ C && = P \in \$ \ \wedge\ C[\$/\$[P \mapsto V]] \ \wedge\ unique\ V \\
&wp\ (if\ L\ then\ S_1\ else\ S_2\ fi)\ C && = (L \rightarrow wp\ S_1\ C) \ \wedge\ (\neg L \rightarrow wp\ S_2\ C) \ \wedge\ (L \vee \neg L) \\
&wp\ (p := V)\ C && = C[p/V] \\
&wp\ (assert\ C_a)\ C && = C \wedge C_a
\end{aligned}$$

The notation $C[a/b]$ defines the substitution of all occurrences of a in C with b. The third conjunct in the precondition of the conditional is needed to make sure that evaluation of L does not lead to an error.

For the normalization, it is important to note how the data invariant, i.e., the postcondition of the procedure is changed by the transformer. As it does not refer

to parameters or local variables, it is only modified at the document variable $ and gets augmented by new conjuncts. As core results for the wp-transformer, we have formally proven that the generated precondition is sound and is "weakest":

Theorem 1 (WP Sound and Weakest).
1. $\vDash \{wp\ S\ C\}\ S\ \{C\}$
2. $\vDash \{C_P\}\ S\ \{C_Q\} \ \wedge\ \| C_P \|_E = true \implies \| wp\ S\ C_Q \|_E = true$

5 Aspects of the Theory Formalization

In this section we discuss some of the techniques used to formalize the presented theory. Furthermore, we explain several of our design decisions, partly triggered by limitations of Isabelle/HOL, and introduce concepts needed for normalization.

Formalizing Semantic Domains. The lack of multi parameter type classes in Isabelle/HOL made it impossible to define a function symbol for the semantics, which is both polymorphic in the syntactic element and its semantic domain. To circumvent this, we defined a union type named *domain*, containing all semantic domains, as well as the bottom value. This makes the handling of bottom easier, because all semantic domains literally share the same bottom value. On the other hand, we had to prove that the semantics of a syntactic element is indeed either from its semantic domain or bottom. The resulting theorems in their different flavors (intro, elim and dest) are used extensively in the proofs of semantics related theorems.

Value, Path and Document Semantics. As the semantics of path expressions can be infinite path multisets, we developed a theory of infinite multisets.[2] More precisely, the multisets can contain infinitely many different values, but each value has only finitely many occurrences, i.e., the number of occurrences is still based on a function $\alpha \to$ nat. This decision allows to give a natural semantics to operators like kind steps and reads, yet it also makes dealing with multisets a bit more complicated in general. As an example consider the semantics of an operation *map* mapping a function f over all elements of a multiset and returning the multiset of the results. For instance, if f is constant for all elements, like the scalar multiplication with zero, the result of the mapping might not be covered by our multiset definition. Consequently, the definition and application of *map* has to take special care for such situations. However, in our framework, such a multiset theory is a good tradeoff, as it simplifies the treatment of paths which might lead to infinite multisets and gets restricted to finitely many values if used for value expressions.

The semantic domain of documents is based on the theory of partial functions as provided by the `Map` theory from Isabelle/HOL.

Mapping and Folding. With the abstract syntax defined for expressions, assertions, and statements, we want both to be able to make syntactic transformations and express properties. In a functional setting – and especially in Isabelle/HOL – a transformation is a bottom-up mapping of a family of functions onto the mutually recursive datatypes. Bottom-up is important in Isabelle/HOL to guarantee termination.

[2] The multiset theory shipped with Isabelle/HOL is for finite multisets only.

The family of functions has to contain an adequate function for every type involved, in our case P, V, M, L, R, D and C. We define the mapping function in a type class, such that it is polymorphic in the syntactic element.

A property on the other hand is described by a folding with a family of functions. We use the single parameter of the type class to make the function polymorphic in the type being folded, but we have to fix it with regard to the type being folded to. We are mostly interested in properties, i.e. boolean foldings, but also counting functions to support proving properties, i.e. nat foldings.

What we end up with are three functions *xmap*, *xprop* and *xcount*, which all take an adequate family of functions and then map or fold all the abstract syntax types we presented. We also defined instances for statements, so we can use the same properties on programs and their contained assertions and expressions.

Well-Formed Assertions. In the design of the assertion language, we decided for a very simple type system. In particular, we do not distinguish between singleton multisets and single values and paths. The main reason for this was to keep the assertion syntax as concise as possible and to avoid the introduction of similar operators for both cases. We can also avoid to deal with partiality from operators like read. Where we need the uniqueness property that an expression yields at most one value or path, we enforce it by checking expressions or assertions for well-formedness:

Definition 5 (Well-Formed Expressions and Assertions). *An expression is **statically unique**, if it denotes at most one singleton value in all environments. An expression or assertion is **well-formed**, if parameters of operators, whose semantics require a singleton value, are statically unique.*

On the core syntax, well-formedness can be checked based on the following criterion: all relations, as well as one of the parameters each in the count-operation and the scalar multiplication, must not directly contain the multiset plus or a read operation based on a path expression using kind steps. As paths and values form a mutual recursion, the property is also mutually recursive and excludes many more combinations. Once we introduce more operators to the syntax, the condition therefore gets more restrictive, yet we maintain the property at every step, so we can use it in the central steps of the normalization.

Safety. The main reason to include bottom into the semantics and use a three-valued logic was to handle partiality, i.e., exceptional behavior within the logic. It allows us to define the *domain* of assertions and expressions as the set of environments in which their semantics is not bottom. And, as we can use assertions as guards to widen the domains of the guarded assertions, we can construct assertions that are always defined. Such assertions are called *safe*. A function (or family of functions) f is considered to be *a domain identity* $id_E\, f$, if it does not change the semantics of an argument x within the domain of x.

Definition 6 (Safety and Domain Identities).

$$safe\, B \equiv \forall E.\ \| B \|_E \neq \bot \qquad id_E\, f \equiv \| B \|_E \neq \bot \longrightarrow \| f\ B \|_E = \| B \|_E$$

Having such a notion of failure and safety is beneficial for syntactic transformations and proving that such transformations are sound. Most transformations simply do not preserve semantics in all corner cases, as they might widen the domain of expressions. Using the notion of safety, we are able to prove, that *xmap* does not alter the semantics of a safe formula, if it uses a family of domain identities.

Theorem 2 (Safe Transformations).

$$id_E\ f\ \wedge\ safe\ C\ \implies \|\ xmap\ f\ C\ \|_E\ =\ \|\ C\ \|_E$$

6 Normalization

The goal of this section is to eliminate update-operations from generated preconditions and to get preconditions into a form where we can identify unchanged parts of the original invariant and separate them from new conjuncts. To reach this point, we need multiple, increasingly complex steps.

The wp generation with normalization starts with the invariant in core syntax and uses the *wp* function to generate a precondition with document updates. After establishing safety, we replace updates by conditionals and multiset arithmetic simulating their effects on expressions. We then remove the conditionals and in a final step eliminate the multiset arithmetic. This brings us back to a core syntax assertion such that new parts are separated from the invariant.

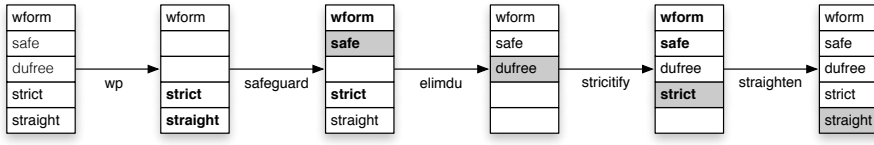
We first define conditionals and arithmetic operators on multisets:

$$\begin{aligned} \text{for } V, P, M \text{ and } R \quad & \alpha ::= \ldots \mid \mathit{if}\ L\ \mathit{then}\ \alpha\ \mathit{else}\ \alpha\ \mathit{fi} \\ \textbf{value expr.} \quad & V ::= \ldots \mid V_{SI} \otimes V \\ \textbf{path expr.} \quad & P ::= \ldots \mid V_{SI} \otimes P \mid P \oplus P \mid P \ominus P \end{aligned}$$

Every expression type and the relations get a conditional, guarded by a single literal. The operator $n \otimes V$ replicates the elements of V n-times, i.e., the first argument has to denote a positive integer. The operators $\oplus$ and $\ominus$ denote multiset plus and minus for paths. With these additions – and the concepts we discussed earlier – we can now define all necessary properties:

$$\begin{aligned} \mathit{wform} &\equiv \text{well-formed} \\ \mathit{dufree} &\equiv \text{does not contain document updates} \\ \mathit{strict} &\equiv \text{does not contain conditionals} \\ \mathit{straight} &\equiv \text{does not contain additional multiset operations} \\ \mathit{safe} &\equiv \text{does never yield bottom (see Def. 6)} \end{aligned}$$

Except for safety, all other properties are purely syntactic and are defined using *xprop* (cf. Sect. 5). The following figure summarizes the normalization process. The details of the individual steps are described in the remainder of the section.



6.1 Establishing Safety

Safety is important for the elimination of updates and multiset arithmetic. As the wp-transformer might return an unsafe assertion, we introduce guards. This is realized by the function *safeguard* that adds appropriate conjuncts to the assertion. This guarding step works because the assertion language allows to express these guards for all its operators. Our function *safeguard* assumes that assertions are strict and straight.

Theorem 3 (Secure).

1. $\mathit{strict}\ C \ \wedge\ \mathit{straight}\ C \ \wedge\ \| C \|_E \neq \bot \implies \| \mathit{safeguard}\ C \|_E = \| C \|_E$
2. $\mathit{strict}\ C \ \wedge\ \mathit{straight}\ C \ \wedge\ \| C \|_E = \bot \implies \| \mathit{safeguard}\ C \|_E = \mathit{false}$
3. $\mathit{strict}\ C \ \wedge\ \mathit{straight}\ C \implies \mathit{safe}\ (\mathit{safeguard}\ C)$

6.2 Elimination of Document Updates

The assertion language was designed in such way that document updates can only occur as parameters of read operations. Thus, we are concerned with expressions like $V_e \equiv M[P_u \mapsto V_?](P_r)$ where P_u is the singleton path at which the update occurs and P_r is a multiset of paths at which the document is read. To make the elimination work, we need to exploit the context of V_e and the specific properties of the assertion language. We distinguish the following two cases.

Unique path. If V_e does not appear in an expression context with a surrounding aggregate function, then we can prove that P_r denotes a singleton path, i.e., the read access is unique (using the safety and well-formedness property). Thus, we can use a case distinction to express the update:

$$\begin{aligned}
&\mathit{elim}_U\ M[P_u \mapsto V](P_r) &&= \mathit{if}\ P_r = P_u\ \mathit{then}\ V\ \mathit{else}\ M(P_r)\ \mathit{fi}\\
&\mathit{elim}_U\ M[P_u \mapsto](P_r) &&= \mathit{if}\ P_r \in \mathit{intersect}\ P_r\ P_u\ \mathit{then}\ \{\}\ \mathit{else}\ M(P_r)\ \mathit{fi}\\
&\mathit{elim}_U\ V &&= V
\end{aligned}$$

where $\mathit{intersect}\ P_r\ P_u$ is a syntactic function that calculates an expression representing the multiset of paths that are contained in P_r and are descendants of P_u.

Updates in Aggregates. Now, we consider document updates that have a context with a surrounding aggregate function. In such a context, P_r denotes a general multiset of paths. The basic idea is to split P_r into the multiset $P_r \ominus (\mathit{count}\ P_r\ P_u \otimes P_u)$ that is not affected by the update and the multiset of paths $\mathit{count}\ P_r\ P_u \otimes P_u$ that is affected. For the first set, we return the old values, for the other set we return new value. And similarly, for the delete operation:

$$\begin{aligned}
&\mathit{elim}_B\ M[P_u \mapsto V](P_r) &&= M(P_r \ominus (\mathit{count}\ P_r\ P_u \otimes P_u)) \oplus (\mathit{count}\ P_r\ P_u \otimes V)\\
&\mathit{elim}_B\ M[P_u \mapsto](P_r) &&= M(P_r \ominus \mathit{intersect}\ P_r\ P_u)\\
&\mathit{elim}_B\ V &&= V
\end{aligned}$$

Correctness properties. For the definitions of elim_U and elim_B we proved the following properties:

Theorem 4 (Basic Eliminations Semantics).

1. $wform\ V\ \wedge\ \|\ V\ \|_E\ =\ \{x\}\ \wedge\ V\ statically\ unique\ \implies\ \|\ elim_U\ V\ \|_E\ =\ \|\ V\ \|_E$
2. $wform\ V\ \wedge\ \|\ V\ \|_E\ \neq\ \bot\ \implies\ \|\ elim_B\ V\ \|_E\ =\ \|\ V\ \|_E$

Using $elim_U$ and $elim_B$, we develop a function *elimdu* that eliminates all occurrences of updates in an assertion. *elimdu* is based on the *xmap* facility in a non-trivial way. Putting this machinery together, we can prove the following central elimination properties:

Theorem 5 (Elim DM).

1. $wform\ C\ \wedge\ safe\ C\ \implies\ \|\ elimdu\ C\ \|_E\ =\ \|\ C\ \|_E$
2. $strict\ C\ \implies\ dufree\ (elimdu\ C)$

The proof of the semantic equivalence is based on properties of the *xmap* facility stated in Thm. 2. Besides eliminating updates, the function *elimdu* preserves well-formedness and safety.

6.3 Eliminating Conditionals

Conditionals in expressions are eliminated by pulling them up to the assertion level and replace them by corresponding logical operators. This is realized by the function *strictify* which is based on a polymorphic pull function and the *xmap* facility. The central properties are:

Theorem 6 (Strictify).

$\|\ strictify\ C\ \|_E\ =\ \|\ C\ \|_E\quad and\quad strict\ (strictify\ C)$

Whereas this elimination is simple in principle, the direct realization in Isabelle/HOL did not work, because it produced a huge combinatorial blow up in internally generated function equations (resulting from the patterns in the function definitions). Our solution was to make the pull operation polymorphic in a type class and realize instances for each abstract syntax type. Although well separated from all other aspects, the conditional elimination constitutes an Isabelle/HOL theory of 600 lines.

6.4 Eliminating Multiset Arithmetic

We are now ready for the final normalization step, which brings us back to the core syntax by eliminating the remaining multiset arithmetic. This elimination is quite complex and not only depends on the concept of safety and well-formed assertions, as was the case in the elimination of updates, but also on the choice of the core syntax operators.

We eliminate the remaining multiset arithmetic by pulling $\otimes$, $\oplus$ and $\ominus$ out of both value and path expressions, up to an enclosing aggregate function, which we know exists because of well-formedness. The *homomorphic* property of each aggregate then allows us to pull them out of the aggregate, i.e., we can express their effect on the aggregate by using only the core operators $\oplus$ and $\odot$ on value expressions. For

the pull-out process to work, we also designed the core language in such a way that all other operators which can contain multiset arithmetic are *homomorphic* too.

All these homomorphic properties are captured by a non-trivial family of functions, which consists of one function per aggregate, one for kind steps and two for path steps, as path steps can contain both path and value expressions with offending operators. As an example we show a selection of homomorphic properties, which demonstrate a lot of different cases:

$$\begin{array}{rcl@{\qquad}rcl} sum\,(V_1 \odot V_2) & \equiv & V_1 \odot sum\,V_2 & P/l[V_1 \oplus V_2] & \equiv & P/l[V_1] \oplus P/l[V_2] \\ sum\,(V_1 \otimes V_2) & \equiv & V_1 \odot sum\,V_2 & sum\,d(P_1 \ominus P_2) & \equiv & sum\,d(P_1) - sum\,d(P_2) \\ \multicolumn{3}{r@{\qquad}}{count\,(V_1 \odot (V_2 \oplus V_3))\,V_4} & \equiv & \multicolumn{2}{l}{count\,(V_1 \odot V_2)\,V_4 \;+\; count\,(V_1 \odot V_3)\,V_4} \end{array}$$

It starts with the simple case of the distribution law of multiplication over summation. Next, the path operator is homomorphic with regard to the multiset plus on values and paths. The third example shows how operators can change when pulled out, in this case the replication becomes a scalar multiplication. This example also shows, that all such equivalences only need to hold within the domain of the left side, as this is enough to use Thm. 2 to exploit safety.

The summation of a read operation containing a multiset minus shows that we sometimes need combinations of operators to be able to pull them out. In this case we simply don't have the minus on values, although the read operation itself is homomorphic with regard to it and the minus on paths. For this reason we also do not define a function for read, but distribute the read cases over the other aggregates. Last but not least, the count operation misses a usable homomorphic property regarding the multiplication, such that we have to pull offending operators out of both.

Based on the family of functions, we define the function *straighten* based on the *xmap* facility in a non-trivial way. We haven proven the following central elimination properties:

Theorem 7 (Normalize).

1. $safe\ C \implies \| straighten\ C \|_E = \| C \|_E$
2. $wform\ C \wedge strict\ C \implies straight\,(straighten\ C)$

As for *elimdu*, the proof of the semantic equivalence is based on safety. The well-formed property comes into play to guarantee that all operations can be eliminated. The function *straighten* of course also preserves all other properties established in the steps before.

6.5 Splitting the Generated Precondition

In Sect. 4 we remarked that the generated precondition strongly resembles the invariant and is only augmented with additional conjuncts and stacked updates at each \$. Making the result safe does not change the precondition at all, but only augments more conjuncts. The real transformation starts with *elimdu*, which removes the stacks of updates and replaces it with either conditionals or multiset arithmetic.

Conditionals are then eliminated using *strictify*, which leads to two versions of the surrounding disjunction, one which does not contain the read operation at all, but a value expression or empty set instead, and one which has exactly the form of the invariant, with added literals that only make it weaker. By this process, we gradually split up specific, much simpler versions of the original disjunction from the one staying generic – but getting weaker – which is still implied by the invariant and can be dropped at the end.

The part of the elimination based on $elim_B$ is more complicated, but looking at its two cases we notice that the original document M appears in a read operation with its original parameter P_r and some other path which is subtracted. As the *straighten* function uses homomorphisms to drag all multiset operations up to a surrounding aggregate, the original read $M(P_r)$ from the invariant is quickly recombined, split up from the other introduced expressions and its surrounding expression restored. It then depends on the context of the aggregate how the simplification can use this property. If the aggregate was embedded in an equality, for example, we can use the equality of the invariant to completely replace the aggregate in the precondition.

7 Related Work

The presented work is related to formalizations of wp-transformations and logics in interactive theorem provers, languages for data properties and schemas, and to literature about the modeling of semistructured data.

Formalizations. Weakest precondition generation has a well-established theory (see, e.g., [12]) and many researchers formalized the wp-generation in theorem provers. For example in [20], the proof assistant Coq has been used to formalize the efficient weakest precondition generation, as well as the required static single assignment and passification transformations, and prove all of them correct. In [19], Schirmer formalizes a generic wp-calculus for imperative languages in Isabelle/HOL. He leaves the state space polymorphic, whereas our contribution focuses on its design and the design and deep embedding of a matching assertion language. Related work is also concerned with formalizing the languages of the XML stack (e.g., Mini-XQuery [9]) and the soundness and correctness proofs for programming logics (e.g., [1, 17]).

Data properties and schemas. A lot of research on maintaining schema constraints has been done using regular languages and automata [2, 18, 3] or using complex modal logics, e.g., context logic [8, 11]. Both can handle structure more naturally and are more powerful in this regard compared to our approach. They can also handle references, but only global ones, as they lack the means to specify targets for constraints. It is therefore not surprising that it is shown in [7], that context logic formulas define regular languages. Incremental checks with these kinds of constraints are well researched, especially for automata.

The main difference to our work is our support for value-based constraints, including aggregates, even in combination with non-global references. To express these kinds of *context-dependent* constraints, we use a classical three-valued logic together with paths and a core set of aggregate functions and predicates, which allows us to combine type and integrity constraints. Precondition generation is also

used for context logic in [11], but they do not support update elimination and incremental checks.

Data models. Our work is based on a concept of paths with integrated local key constraints. With this design decision, we follow many of the ideas of Buneman et al. [4, 5, 6]. They argue for the importance of keys to uniquely identify elements, that keys should not be globally unique and that keys should be composed to "hierarchical keys" for hierarchical structures (see [5]). Their hierarchical keys resemble our core concept of paths. They advocate the usage of paths to specify key constraints and the need for a simpler language than XPath to reason about them, which in particular means paths should only move down the tree and should not contain predicates.

Although they are not in favor of document order – and argue against predicates which could observe it in paths – they use the position of elements to create the unique paths to elements they need for reasoning. By incorporating a simple variant of local key constraints in our paths, we can use element names, rather than positions, to uniquely identify elements. We believe that most of the more complex keys they suggest can be expressed in our logic. This decision, to uniformly handle typical type constraints and more complex integrity constraints within one formalism, is also discussed and backed up by [4].

8 Conclusion

Maintaining data invariants is important in many areas of computing. An efficient approach is to check invariants incrementally, i.e., make sure that invariants are established on data object creation and maintained when data objects are manipulated. To apply this approach one needs to compute the preconditions for all basic procedures manipulating the data. We are interested in automatic and efficient techniques for maintaining integrity constraints going beyond structural properties. To generate efficiently checkable preconditions for such integrity constraints, we needed a normalization process that eliminates update operations from the preconditions and splits the generated precondition into the invariant and the part to be checked.

First attempts to develop such a process by hand failed. The combination of (a) multisets resulting from kind steps in the integrity constraints, (b) intermediate syntax extensions, and (c) exceptional situation caused, e.g., by reading at paths not in the document, is very difficult to manage without tool support. The use of Isabelle/HOL was indispensable for developing succinct definitions, proving the correctess of the transformation, and generating the implementation. The resulting theory[3] consists of more than 9000 lines, the generated Haskell code for the wp-transformer about 2200. This includes the implementation and correctness proof of a first version of a simplifier for preconditions we developed.

Our next steps are the improvement of the simplifier and the development of an appropriate schema language on top of the core assertion language (similar to the one discussed in Sect. 2). With the first version of such a schema language and our approach, we made very positive experiences during the development of the student registration system [13]. This system is in practical use and very stable.

[3] The Isabelle/HOL theory files and PDFs can be found at `https://xcend.de/theory/`.

References

[1] A. W. Appel W. and S. Blazy. “Separation Logic for Small-step Cminor”. In: *TPHOLs 2007*. Vol. 4732. LNCS. Kaiserslautern, Germany: Springer, 2007, pp. 5–21.

[2] D. Barbosa et al. “Efficient Incremental Validation of XML Documents”. In: ICDE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 671–.

[3] B. Bouchou, M. Halfeld, and F. Alves. “Updates and Incremental Validation of XML Documents”. In: *DBPL ’03*. 2003, pp. 216–232.

[4] P. Buneman et al. “Constraints for semistructured data and XML”. In: *SIGMOD Rec.* 30 (1 2001), pp. 47–54.

[5] P. Buneman et al. “Keys for XML”. In: WWW ’01. Hong Kong: ACM, 2001, pp. 201–210.

[6] P. Buneman et al. “Reasoning about Keys for XML”. In: DBPL ’01. London, UK: Springer-Verlag, 2002, pp. 133–148.

[7] C. Calcagno and T. Dinsdale-Young. *Decidability of context logic*. 2009.

[8] C. Calcagno, P. Gardner, and U. Zarfaty. “Context logic and tree update”. In: POPL ’05. Long Beach, California, USA: ACM, 2005, pp. 271–282.

[9] J. Cheney and C. Urban. “Mechanizing the metatheory of mini-XQuery”. In: CPP’11. Kenting, Taiwan: Springer-Verlag, 2011, pp. 280–295.

[10] J. Clark. *RELAX NG Compact Syntax*. `http://www.oasis-open.org/committees/relax-ng/compact-20021121.html`. Nov. 2002.

[11] P. A. Gardner et al. “Local Hoare reasoning about DOM”. In: PODS ’08. Vancouver, Canada: ACM, 2008, pp. 261–270.

[12] D. Gries. *The Science of Programming*. Springer, 1981.

[13] P. Michel and C. Fillibeck. *STATS - Softech Achievement Tracking System*. 2011. URL: `http://xcend.de/stats/start`.

[14] P. Michel and A. Poetzsch-Heffter. “Assertion Support for Manipulating Constrained Data-Centric XML”. In: *PLAN-X ’09*. 2009.

[15] P. Michel and A. Poetzsch-Heffter. “Maintaining XML Data Integrity in Programs - An Abstract Datatype Approach”. In: *SOFSEM ’11, Špindleruv Mlýn, Czech Republic, January 23-29, 2010*. LNCS. Springer, 2010, pp. 600–611.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.

[17] D. von Oheimb. “Hoare logic for Java in Isabelle/HOL”. In: *Concurrency and Computation: Practice and Experience* 13.13 (2001), pp. 1173–1214.

[18] Y. Papakonstantinou and V. Vianu. “Incremental Validation of XML Documents”. In: ICDT. 2003, pp. 47–63.

[19] N. Schirmer. “A verification environment for sequential imperative programs in Isabelle/HOL”. In: LNAI ’05. Springer, 2005, pp. 398–414.

[20] F. Vogels, B. Jacobs, and F. Piessens. “A machine-checked soundness proof for an efficient verification condition generator”. In: *Symposium on Applied Computing*. Vol. 3. ACM, 2010, pp. 2517–2522.