Towards a fully generic theory of data

Douglas A. Creager and Andrew C. Simpson

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD
United Kingdom

Abstract. Modern software systems place a large emphasis on heterogeneous communication. For disparate applications to communicate effectively, a generic theory of data is required that works at the *inter-application* level. The key feature of such a theory is *full generality*, where the data model of an application is not restricted to any particular modeling formalism. Existing solutions do not have this property: while any data can be encoded in terms of XML or using the Semantic Web, such representations provide only *basic generality*, whereby to reason about an arbitrary application's data model it must be re-expressed using the formalism in question. In this paper we present a theory of data which is fully generic and utilizes an extensible design to allow the underlying formalisms to be incorporated into a specification only when necessary. We then show how this theory can be used to investigate two common data equivalence problems — canonicalization and transformation — independently of the datatypes involved.

1 Introduction

Modern software systems must contend with many issues of communication and data exchange that did not exist previously. This raises an interesting class of new problems involving *data equivalence* — the question of whether two data somehow "mean" the same thing, taking into account the data's format, structure, semantics, and application. Two examples are canonicalization and transformation.

Canonicalization involves two equivalences that disagree. For example, in the world of XML [1], digital signatures are problematic, as cryptographic signature algorithms are defined in terms of byte streams. Since a single XML document has many possible binary encodings, a mismatched signature does not necessarily mean that an XML document was modified in transit — it may be a different sequence of bytes that represents the same document.

Transformation, on the other hand, involves maintaining an equivalence between two datatypes. This problem occurs frequently when two applications are linked with a communications channel: the data models of the two applications will likely not be the same, even though they refer to semantically equivalent concepts. Assuming that one cannot easily rewrite the applications, some form of transformation is needed to bridge the mismatch between the two datatypes.

The transformation, however, must ensure that it maintains the semantic equivalence between the two datatypes.

These problems are particularly troublesome, especially when applied to real-world applications, since they must take into account the low-level encoding details of the datatypes. This constrasts with most existing approaches to data modeling and data typing, which abstract away encoding details to simplify the formalism. This abstraction is beneficial to the application designer, since one can exploit *data independence* to separate the high-level application logic from the low-level data storage issues. However, when integrating multiple applications, these low-level issues must be considered.

Further complicating matters, these problems must handle multiple underlying data formalisms. Many data formalisms are *complete*, meaning that any feasible application data model can be represented in the formalism. Complete formalisms, at varying levels of abstraction, include XML, the relational model of data [2], algebraic and co-algebraic datatypes [3], and Shannon's information theory [4]. One can investigate canonicalization and transformation within one of these formalisms. The W3C, for instance, has developed solutions to both within the context of XML [5, 6]. However, these data equivalence issues are problems with data in general, and cross formalism boundaries. No XML transformation method can solve the transformation problem in general.

Fundamentally, existing solutions are not general because multiple data formalisms exist not just in theory, but in practice. There are applications that do not use XML, or do not use the relational model, for perfectly valid reasons. Though these formalisms are complete, they only maintain *basic generality* — to reason about an arbitrary application's data model, it must first be re-expressed in terms of the formalism in question. Instead, we strive for a theory of data with *full generality*, which would allow us to reason formally about an application's data model *as it exists in the application*. Requiring the application to present a separate, theory-compatible, view of its data is not a desirable solution.

This paper presents a fully generic theory of data. It has two key features. The first is that, in addition to the data itself, datatypes and data equivalences are both treated as first-class objects. This lets us reason about generic problems like canonicalization and transformation independently of the particular datatypes and underlying data formalisms used. The second is that the theory is designed in an extensible way; for instance, one can represent an XML datatype in this theory without requiring a complete description of the XML formalism. Of course, including an XML formalism increases the number of properties one can deduce about an XML datatype; however, as we will show, many interesting problems do not require this level of detail.

The remainder of this paper is organized as follows. Section 2 provides a basic description of datatypes and data equivalence. Section 3 provides a formal description of the data theory. We will present this formalism using the Z notation, introductions to which can be found in [7–9]. We will digress slightly from the standard notation, however, by allowing certain operators to be overloaded — to be defined, for instance, for both datatypes and sequences

of datatypes. Section 4 shows how we can use this formalism to investigate canonicalization and transformation. Finally, Section 5 presents our conclusions and suggests an area for future work.

2 Overview

In this section we provide an overview of our data theory. First, we highlight some of the complications that arise when considering the supposedly simple notion of "equivalence". Next, we mention an existing classification that can help illuminate some of the issues involved. Finally, we use this classification to present informal descriptions of several datatypes that we want our theory to support.

2.1 Data equivalences

A key feature to take into account when designing a data theory is *data equivalence*. What do we mean when we say that two data are "equivalent"? A naïve answer would be to define this based on binary equality — two program variables that both contain the 32-bit integer "73" are obviously equivalent. However, this does not capture the entire picture. We present a few obvious counterexamples.

First, we can consider low-level encoding details that can affect data equivalence. For instance, computer processors have a property called *endianness* that affects how multi-byte numbers are stored in memory. "Big-endian" processors store these numbers with their most-significant byte first, whereas "little-endian" processors store the number's least-significant byte first. As an example, consider the number 1,000, which can be encoded in hexadecimal as the 16-bit quantity `03E8`. As shown in Figure 1, when encoded on a big-endian machine, the number is represented by the byte string ⟨⟨`03 E8`⟩⟩. When encoded on a little-endian machine, however, the byte string becomes ⟨⟨`E8 03`⟩⟩. In one sense, that of binary equality, the data are not equivalent; in another, equally valid sense, that of integer equality, they are. This inconsistency holds in reverse, as well. Consider the byte string ⟨⟨`03 E8`⟩⟩. As before, on a big-endian machine, this evaluates to the integer 1,000. On the little-endian machine, however, this is interpreted as the hexadecimal number `E803`, or 59,395. In this case, the data are equivalent according to binary equality, but not according to integer equality.

	⟨⟨03 E8⟩⟩			⟨⟨E8 03⟩⟩	
	Signed	Unsigned		Signed	Unsigned
Big-endian	1,000	1,000	Big-endian	-6,141	59,395
Little-endian	-6,141	59,395	Little-endian	1,000	1,000

Fig. 1. Differing semantic interpretations of binary integers

To further complicate matters, both of the previous examples assumed that the integers were unsigned. Modern computers encode signed integers using

two's complement notation, which has the beneficial property that the same binary addition operator can be used for signed and unsigned numbers. This is a further inconsistency in how a particular byte string can be interpreted as an integer. For example, on a big-endian machine, the byte string ⟨⟨`E8 03`⟩⟩ is interpreted differently as an signed integer (-6,141) and unsigned integer (59,395). This is another case of the data being equivalent according to binary equality, but not according to integer equality.

Similar inconsistencies can appear at higher abstraction levels. For instance, in the HTML markup language [10], it is possible to specify the background color of a Web page with the `bgcolor` attribute of the opening `body` tag. To give a Web page a white background, for instance, one could use the following:

```
<body bgcolor="white">
```

This example represents the color using one of the values in the list of named color strings specified by the HTML standards. It is also possible to specify the color by giving an explicit color value in the RGB color space, such as:

```
<body bgcolor="#ffffff">
```

This example specifies a background color that has the maximum value of 255 ("`ff`" in hexadecimal) for its red, green, and blue components; this color happens to be the color white. These two examples are not equivalent according to binary equality, or even according to character string equality. However, the semantics of the `bgcolor` attribute, as defined by the HTML standard, are such that the character strings "`white`" and "`#ffffff`" represent equivalent colors.

Thus, it is easy to see that a true notion of data equivalence is very application-dependent. It is also a notion that is very dependent on the level of abstraction being used — two data that have different binary encodings might be semantically equivalent, and vice versa. Sometimes semantic equivalence will be more important; sometimes syntactic equivalence will.

2.2 S classification

As seen in the previous section, many different kinds of data equivalence exist, depending on the application and the desired level of abstraction. It will be useful to classify these different equivalences. Ouksel and Sheth identify one possible classification in their study of heterogeneity in information systems [11, 12]: *system*, *syntax*, *structure*, and *semantics*. System heterogeneity refers to the particular combination of hardware and software used to implement an application or datastore. Syntactic heterogeneity refers to the low-level representation of the data — usually in terms of a specific binary encoding. Structural heterogeneity refers to the underlying data primitives used to model an application domain. Semantic heterogeneity refers to the inherent meaning and interpretation of data — the terms *information* and *knowledge* are often used instead of *data* to refer to semantic content.

Ouksel and Sheth introduce this classification, which we will refer to as the *S classification*, to study heterogeneity of information systems — the applications that process data. It is equally effective at describing the data itself. Data equivalence is ambiguous because of its dependence on the desired level of abstraction. The S classification allows us to describe which level of abstraction we are using when analyzing a particular data equivalence, and to highlight differences between data equivalences.

2.3 Datatypes

Any study of data needs to think about datatypes. Broadly speaking, we define a *datatype* to be some set of data. Notionally, a datatype is different than an arbitrary set of data, because the data that constitute a type are supposed to be "similar" in some way. Exactly what form this similarity takes will be application-dependent, just like our notion of data equivalence. To illustrate this, we present several example datatypes, and show how the S classification helps classify them.

Integers. As our first example we can consider the integer types. This is a very low-level set of types; its syntax is a binary string, or sequence of bytes. As we have seen in previous sections, our interpretation of these bytes depends on several factors. At the system level, we must know the integer's endianness, as this affects the order of the bytes in the sequence. At the structural level, we must know the length (and therefore range) of the integer; this is necessary, for instance, to know how much space to reserve in memory for the integer value. At the semantic level, we must know whether the application intends to use this integer as a signed or unsigned value.

Each of these levels can be seen as imposing constraints on which particular data can appear in the datatype's set: an integer datatype contains all of the data that are encoded as a byte string of a particular length, and are interpreted as integers with a particular endianness and signedness. Taken together, this constraint-based definition of the datatype's set brings our original vague notion of "similarity" into focus — but only for this particular datatype.

Postal address (XML). Next we look at a higher-level type — a postal address encoded in XML. This data type might, for example, be used to send "electronic business cards" between address book applications. An instance of this datatype is shown in Figure 2.

At the semantic level, this datatype represents a postal address. As people who have grown up with a postal system, we are able to encapsulate quite a bit of semantic meaning into this concept. This datatype does not provide us with a means of directly encoding this semantic meaning in the data, but it will inform how we write applications that use this data.

At the syntax level, we are using the Extensible Markup Language (XML). Therefore, by extension, our datatype implicitly includes all of the syntactic

```
<address>
  <name>Douglas Creager</name>
  <company>Oxford University Computing Lab</company>
  <line1>Wolfson Building</line1>
  <line2>Parks Road</line2>
  <city>Oxford</city>
  <postcode>OX1 3QD</postcode>
  <country>UK</country>
</address>
```

Fig. 2. Example instance of the postal address XML type

assumptions and requirements of the XML standard [1]: for instance, a binary string that is not well-formed XML cannot be a valid instance of our datatype.

At the structural level, we have an XML schema (not shown) that specifies which XML tags must be used, the content of those tags, and the order in which the tags must appear. Like at the syntax level, this implicitly includes into the datatype definition all of the structural assumptions and requirements of our XML schema: a well-formed XML document that does not match our schema is not a valid instance of our datatype.

At the system level, things are more vague, and will depend in part on the details of the application that is accessing the data. Further, the different aspects of the system interpretation of the datatype are interrelated with the interpretations of the other three levels. Our application will need to have some sort of XML parser, which will handle the syntax level. It will also need application-level logic for parsing the abstract document tree, taking care of the structural level. The application itself will be written with some intuitive notion of what an address actually *is*, taking care of the semantic level. In addition, there will be the low-level details of the application itself, such as the hardware and operating system that it is running on, and any shared libraries that it uses.

Again, we can look at these levels as imposing constraints on the members of the datatype's set: the set contains all of those data that are encoded in XML, using this particular address schema, and that are used as "postal addresses" within the context of some application.

Postal address (database). As another example, we might decide to store these postal addresses in a relational database. This could correspond to an address book application's internal state of the various business cards that someone has collected. An instance of this datatype is shown in Figure 3.

Semantically, this datatype represents a postal address, just as in the previous example. Specifically, this means that the semantic-level constraints imposed on the corresponding sets are the same for both of these datatypes.

Structurally, however, they are obviously quite different. The tables used in this example are based on the relational model, which is quite different than the hierarchical model of XML. Instead of using an XML schema to define which

ADDRESSES table

ADDRESS_ID	13
NAME	"Douglas Creager"
COMPANY	"Oxford University Computing Lab"
LINE_1	"Wolfson Building"
LINE_2	"Parks Road"
CITY	"Oxford"
POST_CODE	"OX1 3QD"
COUNTRY_ID	30

COUNTRIES table

COUNTRY_ID	30
NAME	"United Kingdom"
ABBREV	"UK"

Fig. 3. Example instance of the postal address database type

tags must appear in the tree of XML data, we have a database schema that defines which relational tables we use, and how the tables relate to each other.

The system and syntax levels of this example are rather blurred. Relational databases do provide an application-visible syntax in the SQL query language, but this is not the syntactic representation of the data itself. In fact, we have several similar datatypes that are equivalent semantically and structurally, but different syntactically. We could be referring to the internal representation used by a particular database server, such as PostgreSQL or Oracle. We could be referring to the wire format used by the database server to send the results of a query back to the application. We could be referring to the equivalent SQL `INSERT` statement that could be used to reconstruct the data. We could be referring to the abstract notion of a relational tuple, in which case there is no actual low-level syntax that can be represented in a computer. Often, these syntactic differences will not matter, and we can exploit data independence by ignoring them. Other times, they will be important, and must be included in the datatype definition.

Postal address (Semantic Web). As one final example, we can describe a third postal address datatype, which uses the formalisms and notations of the Semantic Web [13]. The Semantic Web provides a data representation that is better able to express the semantics of the data involved. It does this by representing data using *subject-predicate-object* triples as defined by the Resource Description Framework (RDF) [14, 15]. One can envision these triples as edges in a graph, with the subject being a source node, the object being a destination node, and the predicate being a labeled edge connecting the two. This graph notation is used in Figure 4 to show how a postal address could be expressed in the Semantic Web. (Technically, we should give full URIs [16] for the labels of the edges and the address1 and uk nodes; we provide shorter labels for brevity.)

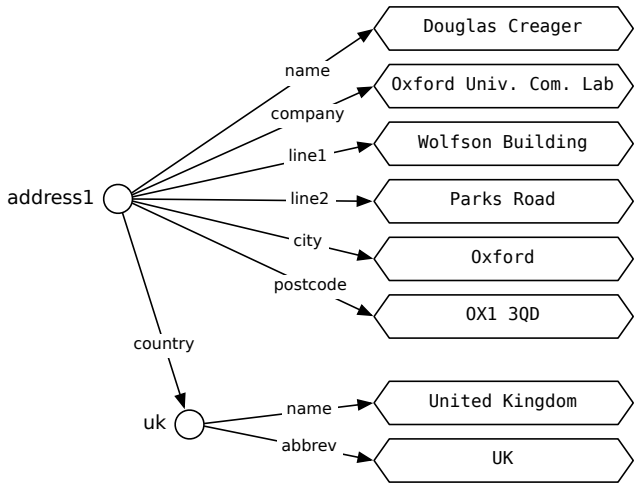

Fig. 4. Example instance of the postal address Semantic Web type

Semantically, this datatype once again represents a postal address; however, by using subject-predicate-object triples, we have encoded a version of these semantics into the data more directly.

Syntactically, the Semantic Web uses XML to encode these graphs of RDF triples, so in one very specific, low-level sense, this datatype is similar to the XML postal datatype described previously. Structurally, however, not just any XML data is allowed — Semantic Web data must exist in a well-formed RDF graph, encoded in XML in a specific way. So while the XML syntax is used for both datatypes, they differ greatly in structure. As with the previous examples, the Semantic Web provides a schema language, the Web Ontology Language (OWL) [17, 18], for stating which particular semantic structures are used. Our datatype would include an OWL ontology describing the overall structure of the graph in Figure 4. RDF graphs that do not match this ontology would not be instances of this datatype.

3 Formalization

The example datatypes described in the previous section were not particularly complex. Even so, they were able to incorporate several formalisms that represent data in completely different ways. A fully generic theory of data must be able to incorporate all of this data, regardless of the differences in the underlying formalisms. In this section we describe such a theory, using a simple running example to provide clarity.

In order to talk about data, we must first define it. Since we are aiming for full generality in this type theory, we cannot assume any kind of structure when referring to data — it must be considered completely opaque. We also define

equivalences, which are relations between data that are reflexive, symmetric, and transitive:

$$[Datum]$$

$$Equivalence : \mathbb{P}(Datum \leftrightarrow Datum)$$

$$\begin{aligned}
&\forall \mathrel{\hat{=}} : Datum \leftrightarrow Datum \bullet \\
&\quad \mathrel{\hat{=}} \in Equivalence \Leftrightarrow \\
&\qquad \forall\, d : \operatorname{dom} \mathrel{\hat{=}} \bullet\ d \mathrel{\hat{=}} d \land \\
&\qquad \forall\, d_1, d_2 : Datum \bullet\ (d_1 \mathrel{\hat{=}} d_2) \Rightarrow (d_2 \mathrel{\hat{=}} d_1) \land \\
&\qquad \forall\, d_1, d_2, d_3 : Datum \bullet\ (d_1 \mathrel{\hat{=}} d_2 \land d_2 \mathrel{\hat{=}} d_3) \Rightarrow (d_1 \mathrel{\hat{=}} d_3)
\end{aligned}$$

As mentioned above, datatypes are represented as sets of data. We can define a simple is-a relation between data and datatypes. Note that this definition says nothing about polymorphism; it is neither mandated nor prohibited.

$$Datatype == \mathbb{P}\, Datum$$

$$\_ \text{ is-a } \_ : Datum \leftrightarrow Datatype$$

$$\forall\, d : Datum;\ t : Datatype \bullet d \text{ is-a } t \Leftrightarrow d \in t$$

However, we have also said that a datatype is not just any set of data; the data in question must be similar in some way. We will express this similarity by defining *interpretations* and *constraints* for each datatype. The interpretations and constraints can both be classified using the S classification.

We can apply this to one of the integer types mentioned in Section 2.3. There are multiple integer datatypes, since bit length, endianness, and signedness all affect the integer interpretation. For simplicity, we will look at one integer datatype in particular: 16-bit, little-endian, and unsigned.

$$Integer_{16,L,U} : Datatype$$

Our first task is to specify the datatype's interpretations. In the case of the integer datatypes, there are two interpretations: its binary encoding, and its integer value. We use the $_{Syn}$ subscript to denote that the binary interpretation is syntactic, and the $_{Sem}$ subscript to denote that the integer interpretation is semantic. Both interpretations are defined as partial functions:

$$\begin{aligned}
&\text{binary}_{Syn} : Datum \nrightarrow ByteString \\
&\text{integer}_{Sem} : Datum \nrightarrow \mathbb{Z}
\end{aligned}$$

In the first case, we define the binary interpretation using the byte string type specified in Appendix A. Similarly, we define an integer interpretation in terms of Z's integer type ($\mathbb{Z}$). It is important to point out that this integer interpretation is not the same as any concrete representation of an integer — rather, it is an abstract mathematical concept that fully captures the semantics of an "integer".

With these interpretations in place, we can formalize our notion of binary equivalence and integer equivalence. Two data are binary-equivalent if their binary interpretations are equal; a similar definition applies to integer equivalence.

$$
\begin{array}{|l}
\_ \stackrel{\circ}{=}_{\mathrm{bin}} \_ : \mathit{Equivalence} \\
\_ \stackrel{\circ}{=}_{\mathrm{int}} \_ : \mathit{Equivalence} \\
\hline
\forall\, d_1, d_2 : \mathit{Datum} \bullet \\
\quad (d_1 \stackrel{\circ}{=}_{\mathrm{bin}} d_2) \Leftrightarrow (\mathrm{binary}_{\mathsf{Syn}}\ d_1 = \mathrm{binary}_{\mathsf{Syn}}\ d_2) \land \\
\quad (d_1 \stackrel{\circ}{=}_{\mathrm{int}} d_2) \Leftrightarrow (\mathrm{integer}_{\mathsf{Sem}}\ d_1 = \mathrm{integer}_{\mathsf{Sem}}\ d_2)
\end{array}
$$

In both cases, we have defined the interpretation as a generic property that can be applied to any *Datum*, since there are many other datatypes that might be encoded in binary or interpreted as an integer. They are both partial functions, though, because not every *Datum* has a binary or integer interpretation. We must then apply these generic properties to our specific datatype:

$$
\begin{array}{l}
\mathit{Integer}_{\mathsf{16,L,U}} \subseteq \mathrm{dom}\, \mathrm{binary}_{\mathsf{Syn}} \\
\mathit{Integer}_{\mathsf{16,L,U}} \subseteq \mathrm{dom}\, \mathrm{integer}_{\mathsf{Sem}}
\end{array}
$$

After defining the interpretations, we must also specify the datatype's constraints. Each of these constraints will depend in some way on at least one of the interpretations. First we have the structural constraint that our integer type is 16 bits long. This is defined in terms of the datatype's binary interpretation. Note that this is a two-way constraint; we must not only say that each of our integers is 16 bits long, but also that every 16-bit binary string can be interpreted as an integer of this type.

$$\forall\, i : \mathit{Integer}_{\mathsf{16,L,U}} \bullet \mathrm{binary}_{\mathsf{Syn}}\ i \in \mathsf{Bytes}\, 2$$

$$\forall\, b : \mathsf{Bytes}\ 2 \bullet \exists_1\, i : \mathit{Integer}_{\mathsf{16,L,U}} \bullet \mathrm{binary}_{\mathsf{Syn}}\ i = b$$

Our other constraint states how the binary and integer interpretations relate to each other, which we can calculate using the functions in Appendix A. This constraint is informed by both the system-level endianness property and the semantic-level signedness property. As before, the constraint is two-way: we must explicitly state that every integer interpretation in the correct numeric range has a corresponding $\mathit{Integer}_{\mathsf{16,L,U}}$.

$$\forall\, i : \mathit{Integer}_{\mathsf{16,L,U}} \bullet \mathrm{integer}_{\mathsf{Sem}}\ i = \mathsf{unsignedInt}\ \mathrm{binary}_{\mathsf{Syn}}\ i$$

$$\forall\, z : 0..(2^{16} - 1) \bullet \exists_1\, i : \mathit{Integer}_{\mathsf{16,L,U}} \bullet \mathrm{integer}_{\mathsf{Sem}}\ i = z$$

This completes a formal specification of this particular integer type. The other integer types can be defined analagously.

The amount of detail that went into the description of this integer datatype highlights an important distinction in our formalism. The $\mathit{Integer}_{\mathsf{16,L,U}}$ datatype had a *full specification* — we provided a complete, formal description of both of the datatype's interpretations, and of the constraints that relate them. In

this particular case, this full specification was not overly verbose. We were able to use Z's existing mathematical integer type ($\mathbb{Z}$) to model the semantics of an integer, and it was relatively straightforward to provide a formal definition of binary data (*ByteString*) in Appendix A.

Often a complete formal description is not readily available, and the effort involved in developing a precise definition might not be worth the benefit gained from doing so. In these cases, it is possible to provide a datatype with a *partial specification*, where we define some of the interpretations and constraints as abstract entities. This becomes especially useful when considering how multiple partially-specified datatypes relate to each other.

For instance, we can revisit the postal address types, which have new interpretations that were not used by the integer datatype. However, whereas we provided (or were given) full definitions of the $\mathbb{Z}$ and *ByteString* types, we will leave these new interpretations abstract:

$[XMLDocument, XMLSchema]$
$[RelationalTuple, RelationalSchema]$
$[PostalAddress]$

XMLDocument represents the logical document tree of an XML document, while *RelationalTuple* represents a row from some relational table. In both cases, we have also mentioned a type that represents the schema that describes the data's structure. *PostalAddress* represents the semantic meaning of a postal address. We can now define interpretations and equivalences for these three Z types, similarly to the integer example:

$$\mathrm{xml}_{\mathsf{Struct}} : Datum \pfun XMLDocument$$
$$\mathrm{relational}_{\mathsf{Struct}} : Datum \pfun RelationalTuple$$
$$\mathrm{address}_{\mathsf{Sem}} : Datum \pfun PostalAddress$$

$$\_ \mathrel{\dot{=}}_{\mathrm{xml}} \_ : Equivalence$$
$$\_ \mathrel{\dot{=}}_{\mathrm{rel}} \_ : Equivalence$$
$$\_ \mathrel{\dot{=}}_{\mathrm{addr}} \_ : Equivalence$$

$$\forall\, d_1, d_2 : Datum \bullet$$
$$(d_1 \mathrel{\dot{=}}_{\mathrm{xml}} d_2) \Leftrightarrow (\mathrm{xml}_{\mathsf{Struct}}\ d_1 = \mathrm{xml}_{\mathsf{Struct}}\ d_2) \land$$
$$(d_1 \mathrel{\dot{=}}_{\mathrm{rel}} d_2) \Leftrightarrow (\mathrm{relational}_{\mathsf{Struct}}\ d_1 = \mathrm{relational}_{\mathsf{Struct}}\ d_2) \land$$
$$(d_1 \mathrel{\dot{=}}_{\mathrm{addr}} d_2) \Leftrightarrow (\mathrm{address}_{\mathsf{Sem}}\ d_1 = \mathrm{address}_{\mathsf{Sem}}\ d_2)$$

With these interpretations defined, we can define the types themselves. The XML address datatype will have binary, XML, and address interpretations; the relational address datatype will have relational and address interpretations. (We ignore the syntax of the relational datatype to maintain data independence.)

$$Address_{\mathsf{XML}} : Datatype$$

$$Address_{\mathsf{XML}} \subseteq \mathrm{dom}\, \mathrm{binary}_{\mathsf{Syn}}$$
$$Address_{\mathsf{XML}} \subseteq \mathrm{dom}\, \mathrm{xml}_{\mathsf{Struct}}$$
$$Address_{\mathsf{XML}} \subseteq \mathrm{dom}\, \mathrm{address}_{\mathsf{Sem}}$$

$$Address_{\mathsf{Rel}} : Datatype$$

$$Address_{\mathsf{Rel}} \subseteq \mathrm{dom}\, \mathrm{relational}_{\mathsf{Struct}}$$
$$Address_{\mathsf{Rel}} \subseteq \mathrm{dom}\, \mathrm{address}_{\mathsf{Sem}}$$

Next we specify the constraints, for which we will need several helper functions and relations, which, again, we do not provide full definitions for:

$$\mathsf{encodes}: ByteString \nrightarrow XMLDocument$$

$$\mathsf{instanceof}: XMLDocument \leftrightarrow XMLSchema$$
$$\mathsf{instanceof}: RelationalTuple \leftrightarrow RelationalSchema$$
$$AddressSchema_{\mathsf{XML}} : XMLSchema$$
$$AddressSchema_{\mathsf{Rel}} : RelationalSchema$$

$$\mathsf{interpret}: XMLDocument \nrightarrow PostalAddress$$
$$\mathsf{interpret}: RelationalTuple \nrightarrow PostalAddress$$

The encodes function maps a byte string to the XML document that it represents. (The function is partial since not all byte strings represent valid XML documents.) The two flavors of the instanceof relation allow us to verify that an XML document or relational tuple matches its corresponding schema. We also mention the particular schemas used by our XML and relational datatypes. The two flavors of interpret allow us to determine the semantic meaning of an XML document or relational tuple. These are then applied to the datatypes as constraints:

$$\forall\, d : Address_{\mathsf{XML}} \bullet$$
$$(\mathrm{binary}_{\mathsf{Syn}}\ d)\ \mathsf{encodes}\ (\mathrm{xml}_{\mathsf{Struct}}\ d) \land$$
$$(\mathrm{xml}_{\mathsf{Struct}}\ d)\ \mathsf{instanceof}\ AddressSchema_{\mathsf{XML}} \land$$
$$(\mathrm{xml}_{\mathsf{Struct}}\ d)\ \mathsf{interpret}\ (\mathrm{address}_{\mathsf{Sem}}\ d)$$

$$\forall\, d : Address_{\mathsf{Rel}} \bullet$$
$$(\mathrm{relational}_{\mathsf{Struct}}\ d)\ \mathsf{instanceof}\ AddressSchema_{\mathsf{Rel}} \land$$
$$(\mathrm{relational}_{\mathsf{Struct}}\ d)\ \mathsf{interpret}\ (\mathrm{address}_{\mathsf{Sem}}\ d)$$

This provides a formal rendering of the datatype definitions in Section 2.3. For an XML postal address, its binary encoding must match its logical XML document; this XML document must match the postal address schema; and the document must have some valid real-world interpretation as a postal address. Similar constraints apply to relational postal addresses.

4 Canonicalization and transformation

The formalism presented in the previous section allowed us to give formal definitions for the datatypes from Section 2.3. However, we only provided a partial specification for the postal address types. If we were so inclined, it would certainly have been possible to give them full specifications. This would have required a formal specification of each of the datatypes' interpretations. For XML,

it would be relatively straightforward to define in terms of trees of data nodes; for relational data, we have the underlying relational model to work with. The real-world semantics could have been modeled using a knowledge-representation framework such as the Semantic Web. All of these specifications are possible; however, they would also be much more verbose than what we have presented, and time-consuming to produce and verify. As will be shown in this section, we can still describe and reason about useful properties of these datatypes with partial specifications, rendering this cost unnecessary much of the time. We look specifically at canonicalization and transformation.

4.1 Canonicalization

One example that highlights the importance of differing notions of equivalence is *data canonicalization*. A well-known current example of canonicalization involves XML documents and digital signatures [5, 19, 20].

The problem stems from the fact that every XML document has many different encodings as a concrete sequence of bytes. Three aspects of the XML syntax, in particular, affect the encoding of a document: attributes, namespaces, and whitespace. In most XML applications, these differences are not a problem, since the application works with a high-level view of the XML content, often in the form of the Document Object Model API [21], which represents an XML document by its abstract tree structure. However, one application area where these differences are important is *digital signatures*. Briefly, digital signatures are a more cryptographically-secure version of checksums and error-correcting codes. They provide a means of attesting that the content of a document has not been modified in transit between two parties. This is an important security feature in modern applications that helps prevent, among other things, man-in-the-middle attacks.

The algorithms used to implement digital signatures are not constrained to XML documents; they work on any binary payload. Alice can send an XML document to Bob, signing it before sending it along the communications channel. However, there might be communications gateways in between Alice and Bob that modify the binary representation of an XML document without modifying the document structure. When Bob receives the document, its binary representation will have changed, and Alice's signature will no longer match the document.

Looking at this in terms of our datatype formalism, we can define a function that can sign a byte string:

$$[Signature]$$

$$\mathsf{sign} : ByteString \rightarrow Signature$$

$$\forall\, b_1, b_2 : ByteString \bullet (\mathsf{sign}\ b_1 = \mathsf{sign}\ b_2) \Leftrightarrow (b_1 = b_2)$$

This captures the essence of a digital signature: if the signatures match, the byte strings most likely match as well; conversely, if the signatures do not match, the

byte strings are different. (It should be noted that is not technically a true equivalence. The number of signatures is much smaller than the number of binary strings, so some overlap is inevitable. Rather than a full guarantee, matching signatures *strongly imply* that the binary strings are the same. However, for the purposes of this example, this distinction is not important, and we will treat it as an equivalence.)

We can define a similar function for signing data that simply signs a datum's binary interpretation; signatures then work for arbitrary data, too, *but only under binary equivalence*:

$$\mathsf{sign} : \mathit{Datum} \pfun \mathit{Signature}$$

$$\forall\, d : \mathit{Datum} \bullet \mathsf{sign}\ d = \mathsf{sign}\, \mathrm{binary}_{\mathsf{Syn}}\ d$$
$$\forall\, d_1, d_2 : \mathit{Datum} \bullet (\mathsf{sign}\ d_1 = \mathsf{sign}\ d_2) \Leftrightarrow (d_1 \mathrel{\dot{=}}_{\mathrm{bin}} d_2)$$

We run into a problem in the case of XML. Alice's and Bob's applications do not care about binary equivalence; they care about XML equivalence. The hope, then, is that the signature predicate holds for XML equivalence, too:

$$\forall\, d_1, d_2 : \mathit{Datum} \bullet (\mathsf{sign}\ d_1 = \mathsf{sign}\ d_2) \stackrel{?}{\Leftrightarrow} (d_1 \mathrel{\dot{=}}_{\mathrm{xml}} d_2)$$

For this to be the case, we would need the following implication to hold:

$$\forall\, d_1, d_2 : \mathit{Datum} \bullet (d_1 \mathrel{\dot{\neq}}_{\mathrm{bin}} d_2) \stackrel{?}{\Rightarrow} (d_1 \mathrel{\dot{\neq}}_{\mathrm{xml}} d_2)$$

However, we know this is not true; two different byte strings *can* represent the same XML document.

What is needed is a *canonicalization function*. In the case of XML documents, we need to choose one particular binary encoding for each logical XML document. We would then define a function $\mathsf{canon}_{\mathrm{xml}}$ that maps an XML datum to its canonical binary encoding. The required property would then hold:

$$\forall\, d_1, d_2 : \mathit{Datum} \bullet (\mathsf{canon}_{\mathrm{xml}}\ d_1 \mathrel{\dot{=}}_{\mathrm{bin}} \mathsf{canon}_{\mathrm{xml}}\ d_2) \Leftrightarrow (d_1 \mathrel{\dot{=}}_{\mathrm{xml}} d_2)$$

Two XML documents that have the same logical structure, when canonicalized, would also have the same binary encoding. Expressed another way, two data that are XML-equivalent, when canonicalized, would also be binary-equivalent. In fact, we can define canonicalization as a generic property that a function might provide between any two equivalences:

$$\mathit{DataFunction} == \mathit{Datum} \pfun \mathit{Datum}$$

$$\_\ \mathsf{canonicalizes}\ [\_/\_] : \mathit{DataFunction} \rel (\mathit{Equivalence} \times \mathit{Equivalence})$$

$$\forall\, f : \mathit{DataFunction};\ \dot{=}_1, \dot{=}_2 : \mathit{Equivalence} \bullet$$
$$f\ \mathsf{canonicalizes}\ [\dot{=}_1/\dot{=}_2] \Leftrightarrow$$
$$\forall\, d_1, d_2 : \mathit{Datum} \bullet (d_1 \mathrel{\dot{=}}_1 d_2) \Leftrightarrow (f\ d_1 \mathrel{\dot{=}}_2 f\ d_2)$$

With this generic property defined, we can easily state that the $\mathsf{canon}_{\mathrm{xml}}$ function canonicalizes XML equivalence in terms of binary equivalence:

$$\mathsf{canon}_{\mathrm{xml}} : \mathit{DataFunction}$$
$$\mathsf{canon}_{\mathrm{xml}}\ \mathsf{canonicalizes}\ \left[\ \stackrel{\circ}{=}_{\mathrm{xml}} / \stackrel{\circ}{=}_{\mathrm{bin}}\ \right]$$

It should be noted that this formalism does not help us find a detailed definition of the $\mathsf{canon}_{\mathrm{xml}}$ function. In general, the definition of a canonicalization function will be highly dependent on the details of the underlying data formalism and how this relates to its binary encodings.

4.2 Transformations

The canonicalizations described in the previous section provide one category of special data function. *Transformations* provide another. They differ in how they relate to the data equivalences that hold on particular types. In the case of canonicalization, the function is used to ensure that two equivalences agree with each other. A transformation, on other hand, only deals with a single equivalence; the function provides a bridge between two datatypes that maintains this equivalence.

One situation where transformations are useful arises often in application integration: linking two heterogeneous applications with a communications channel. These applications will often have completely different datatypes for their inputs and outputs; however, as implied by the fact that we want them to communicate, there is at least some semantic equivalence between the datatypes. (If there were not, what communication would be possible?) We can return once again to the postal address example, and consider two address book applications: one which uses the relational datatype, and one which uses the XML datatype. Since the datatypes both refer to postal addresses, they are semantically equivalent; therefore, in theory, the two applications can communicate. However, before we can even begin to consider the details of the communications channel itself, we must reconcile the difference in datatypes. Assuming that rewriting the applications is impossible or too expensive, some transformation is needed to link the applications. This transformation would translate data from one datatype to another, while maintaining the semantic equivalence.

We can model this situation similarly to the canonicalization example and reuse the *DataFunction* type from that section. We need to introduce the notion of *typing* the data functions, however:

$$\_\ \mathsf{source}\ \_ : \mathit{Datatype} \leftrightarrow \mathit{DataFunction}$$
$$\_\ \mathsf{dest}\ \_ : \mathit{Datatype} \leftrightarrow \mathit{DataFunction}$$
$$\forall\, t : \mathit{Datatype};\ f : \mathit{DataFunction} \bullet$$
$$\quad t\ \mathsf{source}\ f \Leftrightarrow \mathrm{dom}\, f \subseteq t \land$$
$$\quad t\ \mathsf{dest}\ f \Leftrightarrow \mathrm{ran}\, f \subseteq t$$

We can define the *source* and *destination* datatypes for a data function; this simply states that all of the function's input or output values come from the

respective datatype. Since we have not prohibited polymorphism, we must define this as a relation — i.e., there might be many datatypes that encompass the input values for a particular function; all of them can be said to be sources of the function. A data function *links* each of its sources to each of its destinations:

$$
\begin{array}{|l}
\_ \ \mathsf{links} \ [\_ \rightharpoonup \_] : \\
\quad \mathit{DataFunction} \leftrightarrow (\mathit{Datatype} \times \mathit{Datatype}) \\
\hline
\forall f : \mathit{DataFunction};\ t_S, t_D : \mathit{Datatype} \bullet \\
\quad f \ \mathsf{links} \ [t_S \rightharpoonup t_D] \Leftrightarrow (t_S \ \mathsf{source} \ f) \land (t_D \ \mathsf{dest} \ f)
\end{array}
$$

Lastly, a data function *maintains* an equivalence if that equivalence holds between each of the function's inputs and the corresponding output:

$$
\begin{array}{|l}
\_ \ \mathsf{maintains} \ \_ : \mathit{DataFunction} \leftrightarrow \mathit{Equivalence} \\
\hline
\forall f : \mathit{DataFunction};\ \hat{=} : \mathit{Equivalence} \bullet \\
\quad f \ \mathsf{maintains} \ \hat{=} \Leftrightarrow \\
\qquad \forall d : \mathrm{dom}\, f \bullet d \ \hat{=} \ (f\ d)
\end{array}
$$

With these definitions in place, we can state the existence of the required transformation: it links the XML and relational postal address datatypes, and maintains the postal address semantic equivalence.

$$
\begin{array}{|l}
\mathsf{xform}_{\mathrm{Address}} : \mathit{DataFunction} \\
\hline
\mathsf{xform}_{\mathrm{Address}} \ \mathsf{links} \ [\mathit{Address}_{\mathsf{XML}} \rightharpoonup \mathit{Address}_{\mathsf{Rel}}] \\
\mathsf{xform}_{\mathrm{Address}} \ \mathsf{maintains} \ \hat{=}_{\mathrm{addr}}
\end{array}
$$

The $\mathsf{xform}_{\mathrm{Address}}$ function is a transformation since it links the $\mathit{Address}_{\mathsf{XML}}$ and $\mathit{Address}_{\mathsf{Rel}}$ datatypes while maintaining the $\hat{=}_{\mathrm{addr}}$ equivalence. Note that once again, we have abstracted away a lot of unnecessary detail — we have said nothing about how $\mathsf{xform}_{\mathrm{Address}}$ performs this transformation.

Since transformations are modeled as functions between data, they are also composable. This allows us to consider sequences of datatypes, and sequences of data functions:

$$
\begin{array}{l}
\mathit{TypeSequence} == \mathrm{seq}_1 \ \mathit{Datatype} \\
\mathit{FunctionSequence} == \mathrm{seq}_1 \ \mathit{DataFunction}
\end{array}
$$

$$
\begin{array}{|l}
\_ \ \mathsf{types} \ \_ : \mathit{TypeSequence} \leftrightarrow \mathit{FunctionSequence} \\
\_ \ \mathsf{source} \ \_ : \mathit{Datatype} \leftrightarrow \mathit{FunctionSequence} \\
\_ \ \mathsf{dest} \ \_ : \mathit{Datatype} \leftrightarrow \mathit{FunctionSequence} \\
\hline
\forall ts : \mathit{TypeSequence};\ fs : \mathit{FunctionSequence} \bullet \\
\quad ts \ \mathsf{types} \ fs \Leftrightarrow \\
\qquad \#ts = \#fs + 1 \land \\
\qquad \forall i : 1 \mathinner{..} \#fs \bullet ts(i) \ \mathsf{source} \ fs(i) \land ts(i+1) \ \mathsf{dest} \ fs(i) \land \\
\qquad (\mathit{head}\ ts) \ \mathsf{source} \ fs \land \\
\qquad (\mathit{last}\ ts) \ \mathsf{dest} \ fs
\end{array}
$$

A sequence of functions is *well-typed* if the destination type of each data function matches the source type of its successor. We can then define the source and dest operators for sequences, much as they are defined for individual functions: the source (destination) of a function sequence is the source (destination) of the first (last) function in the sequence.

With these definitions, we can define a compose operator on function sequences:

$$
\begin{array}{|l}
\mathsf{compose}\ \_ : \mathit{FunctionSequence} \rightarrow \mathit{DataFunction} \\
\hline
\forall f : \mathit{DataFunction} \bullet \mathsf{compose}\ \langle f \rangle = f \\
\forall \mathit{fs}_1, \mathit{fs}_2 : \mathit{FunctionSequence} \bullet \\
\quad \mathsf{compose}\ \mathit{fs}_1 \frown \mathit{fs}_2 = (\mathsf{compose}\ \mathit{fs}_2) \circ (\mathsf{compose}\ \mathit{fs}_1) \\
\forall t : \mathit{Datatype};\ \mathit{fs} : \mathit{FunctionSequence} \bullet \\
\quad t\ \mathsf{source}\ \mathit{fs} \Leftrightarrow t\ \mathsf{source}\ (\mathsf{compose}\ \mathit{fs}) \wedge \\
\quad t\ \mathsf{dest}\ \mathit{fs} \Leftrightarrow t\ \mathsf{dest}\ (\mathsf{compose}\ \mathit{fs})
\end{array}
$$

The operator is defined in the obvious way using structural induction, exploiting the fact that functional composition is associative. Note the order reversal; for the $\circ$ operator, the function to apply first is on the right, whereas in a function sequence, it is on the left.

5 Discussion

In this paper we have provided a formalism for an inter-application theory of data. This formalism features *full generality*, in that any application data model can be represented as is, without requiring conversion to another data formalism. This theory represents data as abstract entities with several *interpretations* and *constraints*, with the constraints defining how the interpretations of a datatype relate to each other. Underlying formalisms such as the XML or relational models can be incorporated into a specification to give a precise meaning to an interpretation; however, this is optional. It is also valid for an interpretation to remain abstract.

Our formalism is able to represent the low-level encoding details of a datatype in addition to the usual high-level semantic descriptions. At first glance, this seems to violate the data independence principle. However, this is not the case. Data independence can be maintained, when necessary, simply by leaving the datatype's low-level syntactic and structural interpretations abstract, as in the case of the $\mathit{Address}_{\mathsf{Rel}}$ datatype.

However, data independence is not useful when studying problems like canonicalization and transformation in a fully generic way. First, we must be able to handle different data formalisms, and cannot rely on a single abstraction to provide data independence. Second, we must be able to handle the data's binary encoding, which are exactly the details that are hidden by data independence. By allowing (but not requiring) our formalism to include descriptions of these low-level details, we are able to reason about this class of problems.

The similar notion of data refinement [9, Chapter 16] tackles many of these issues from a slightly different viewpoint. Looking at the example of integer endianness, one would consider the mathematical set of integers ($\mathbb{Z}$) to be a datatype that happens to be defined at a high, abstract level. One could then define a lower-level type, such as $\mathit{Integer}_{\mathsf{16,L,U}}$, that represents a binary string interpreted in a particular way. One would then prove that $\mathit{Integer}_{\mathsf{16,L,U}}$ *refines* $\mathbb{Z}$ — that a specification written in terms of $\mathbb{Z}$ could use $\mathit{Integer}_{\mathsf{16,L,U}}$ as a drop-in replacement without affecting the specification.

Data refinement is also possible with our framework. Instead of writing an application specification in terms of the $\mathbb{Z}$ "datatype", it is written in terms of any *Datum* that has an $\text{integer}_{\mathsf{Sem}}$ interpretation. The refinement proof then consists of showing how a datatype's $\text{integer}_{\mathsf{Sem}}$ interpretation correctly relates to one of its other interpretations. Our approach is different in two ways. First, $\mathbb{Z}$ is defined as an interpretation rather than as a first-class datatype, which allows multiple datatypes to have an integer interpretation. This difference does not mean much in terms of expressiveness; with data refinement, it is just as easy to define multiple types that all refine $\mathbb{Z}$. It does, however, make possible the second difference: that data equivalences are first class objects that might be defined abstractly. This allows problems like canonicalization and transformation to be investigated generically in terms of abstract equivalences, without having to rely on the details of the datatypes involved.

It is important to note that this data theory is not meant to be a replacement for any of the other data formalisms that have been mentioned. For instance, our description of canonicalization is not meant to replace the work of the XML Digital Signature initiative [5]; rather, it is meant to provide a higher-level framework in which to ground the XML-specific canonicalization. We envision this framework serving two purposes: as a bridge between data formalisms, and an abstraction away from them. Again, this allows us to reason about generic data without being forced to consider the particular formalism that it is defined in.

Further work in this area will focus on the transformation formalism mentioned in Section 4.2. The current type theory allows one to state the existence of transformations, and to provide specifications of these transformations at whatever detail is necessary. However, the theory provides no mechanism for *discovering* transformations. We hope to exploit the composability of transformation to develop a transformation framework that supports efficient discovery, while retaining the full generality of the type theory.

Acknowledgments

Doug Creager's work is funded by the Software Engineering Programme of the Oxford University Computing Laboratory. The authors would like to thank David Faitelson for providing valuable feedback on the manuscript of this paper. The comments of the anonymous reviewers were also very helpful in improving the readability and content of the paper.

References

1. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F., eds.: Extensible Markup Language. W3C (2004) http://www.w3.org/TR/REC-xml/.
2. Codd, E.F.: A relational model of data for large shared data bases. Communications of the ACM **13** (1970) 377–387
3. Jacobs, B., Rutten, J.: A tutorial on (co) algebras and (co) induction. EATCS Bulletin **62** (1997) 222–259
4. Shannon, C., Weaver, W.: The Mathematical Theory of Communication. University of Illinois (1963)
5. Eastlake, D., Reagle, J., Solo, D., eds.: XML-Signature Syntax and Processing. W3C (2002) http://www.w3.org/TR/xml-dsigcore/.
6. Clark, J., ed.: XSL Transformations (XSLT). W3C (1999) http://www.w3.org/TR/xslt/.
7. Spivey, J.M.: An introduction to Z and formal specification. Software Engineering Journal **4** (1989) 40–50
8. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall International (1989)
9. Woodcock, J.C.P., Davies, J.W.M.: Using Z: Specification, refinement, and proof. Prentice Hall (1996)
10. Raggett, D., Le Hors, A., Jacobs, I.: HTML 4.01 Specification. W3C (1999) http://www.w3.org/TR/html.
11. Ouksel, A.M., Sheth, A.: Semantic interoperability in global information systems. SIGMOD Record **28** (1999) 5–12
12. Sheth, A.: Changing focus on interoperability in information systems: From system, syntax, structure to semantics. In Goodchild, M.F., Egenhofer, M.J., Fegeas, R., Kottman, C.A., eds.: Interoperating Geographic Information Systems, Kluwer Publishers (1998)
13. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001) 29–37
14. Klyne, G., Carroll, J.J., eds.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C (2004) http://www.w3.org/TR/rdf-concepts/.
15. Beckett, D., ed.: RDF/XML Syntax Specification. W3C (2004) http://www.w3.org/TR/rdf-syntax-grammar/.
16. Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identifier (URI): Generic syntax. IETF Requests for Comments **3986** (2005) http://www.ietf.org/rfc/rfc3986.txt.
17. McGuinness, D.L., van Harmelen, F., eds.: OWL Web Ontology Language Overview. W3C (2004) http://www.w3.org/TR/owl-features/.
18. Smith, M.K., Welty, C., McGuinness, D.L., eds.: OWL Web Ontology Language Guide. W3C (2004) http://www.w3.org/TR/owl-guide/.
19. Boyer, J.: Canonical XML. W3C (2001) http://www.w3.org/TR/xml-c14n/.
20. Boyer, J., Eastlake, D.E., Reagle, J.: Exclusive XML Canonicalization. W3C (2002) http://www.w3.org/TR/xml-exc-c14n/.
21. Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) Level 3 Core Specification. W3C (2004) http://www.w3.org/TR/DOM-Level-3-Core/.

A Z Specification for Binary Integers

Any description of binary data must first define *bits*. Bits are simple — there are exactly two of them: $\mathtt{0}$ and $\mathtt{1}$. We can also define a *bit string*, which is an ordered sequence of bits.

$$Bit ::= \mathtt{0} \mid \mathtt{1}$$

$$BitString == \operatorname{seq} Bit$$

We will often need to translate a binary string into its integer equivalent. (This is not to be confused with interpreting integer datatypes; this is a low-level helper function to get the decimal interpretation of a base-2 integer.)

$$\begin{array}{|l} \mathsf{int}_{\mathsf{Bit}} : Bit \rightarrowtail \mathbb{N} \\ \mathsf{int}_{\mathsf{Bits}} : BitString \rightarrow \mathbb{N} \\ \hline \mathsf{int}_{\mathsf{Bit}}\ \mathtt{0} = 0 \\ \mathsf{int}_{\mathsf{Bit}}\ \mathtt{1} = 1 \\ \mathsf{int}_{\mathsf{Bits}}\ \langle\rangle = 0 \\ \forall\, b : Bit;\ bin : BitString \bullet \mathsf{int}_{\mathsf{Bits}}\ \langle b \rangle \frown bin = (\mathsf{int}_{\mathsf{Bits}}\ bin) * 2 + (\mathsf{int}_{\mathsf{Bit}}\ b) \end{array}$$

This allows us to define a *byte*, which is an 8-bit value, and a *byte string*, which is an ordered sequence of bytes. We also define a Bytes function which returns the set of all byte strings of a given length.

$$Byte == \{\, b : BitString \bullet \#b = 8 \,\}$$

$$ByteString == \operatorname{seq} Byte$$

$$\begin{array}{|l} \mathsf{Bytes}\ \_ : \mathbb{N} \rightarrowtail \mathbb{P}\, ByteString \\ \hline \forall\, n : \mathbb{N} \bullet \mathsf{Bytes}\ n = \{\, b : ByteString \bullet \#b = n \,\} \end{array}$$

When referring to literal byte strings, we will denote the bytes by their numeric (specifically hexadecimal) values, as in $\langle\!\langle \mathtt{48\,6F} \rangle\!\rangle$.

The integer representation of a byte string is more complicated, because we must contend with signedness and endianness issues. In this paper we are only considering unsigned, little-endian numbers; this is the simplest case, since we can use distributed concatenation to turn the little-endian byte string into an equivalent little-endian bit string. This bit string can be evaluated using the $\mathsf{int}_{\mathsf{Bits}}$ function.

$$\begin{array}{|l} \mathsf{unsignedInt} : ByteString \rightarrow \mathbb{N} \\ \hline \forall\, b : ByteString \bullet \mathsf{unsignedInt}\ b = \mathsf{int}_{\mathsf{Bits}}\ (\frown / b) \end{array}$$