

A Typed Model for Encoding-Based Protocol Interoperability *

Adam D. Bradley, Azer Bestavros, and Assaf J. Kfoury
Boston University Department of Computer Science
artdodge,best,kfoury@cs.bu.edu

Abstract

Documentation of the HTTP protocol includes precise descriptions of the syntax of the protocol, but lacks similarly precise specification of the semantics of messages and message bodies. Semantics are stated in English prose; while this makes the document more intuitively accessible, it makes any sort of formal claims of correctness or interoperability difficult to derive from the specification itself. We propose “layered types”, a formal description of the interpretive semantics of HTTP message bodies based upon the stacked type syntax. This model allows us to formally declare semantics for content-related HTTP headers and offers a precise way of characterizing interoperability between current and future protocol revisions and extensions.

1. Introduction

The fundamental premise of the web architecture is quite straightforward: the web provides a syntactically consistent and semantically predictable interface to arbitrary network-connected resources via the exchange of *representations* (documents which *represent* the state of their creators) [8].

While this core is fairly straightforward, the HTTP protocol itself [7] has integrated a variety of (largely performance-oriented) features [10] which have been integrated with the protocol’s syntax and specification in a largely *ad hoc* manner. The result is a specification with uneven precision, making it difficult to directly reason about its correctness [3, 4].

One area of imprecision in the specification is with respect to the *content model*, *i.e.*, the semantics of representations (message payloads) in their raw and decoded forms and the relationship between these semantics and the protocol’s syntax. While Mogul has offered an excellent intuitive content model [14] incorporated into a standards-track RFC [15], it is not presented as a

formalism which can be precisely and unambiguously related to the syntax of the protocol. The purpose of this paper is to formally define and describe the content model of the HTTP protocol, and to use this model to establish syntax-directed semantics for HTTP messages. The increased precision simplifies the assessment of the protocol’s correctness and the interoperability between multiple revisions and extensions.

This paper is organized as follows: Section 2 presents an overview of the HTTP protocol with particular emphasis upon the definition of its data model. Section 3 presents *layered types*, an unambiguous formalism representing the HTTP data model; layered types are employed in Section 4 as the basis for specifying *syntax-directed semantics* for HTTP messages under several versions of the HTTP protocol, and techniques for assessing correctness and interoperability of HTTP revisions and extensions are then discussed.

2. An Overview of HTTP

HTTP is a stateless client-server request-response protocol which roughly adheres to the REST (Representational State Transfer) architectural style [8], meaning (intuitively) that clients interact with servers by exchanging documents which represent the state of the applications employing the protocol. These *representations* may have arbitrary underlying media types (text, audio, video, *etc.*) and may be encoded or transformed in a variety of ways to support performance optimizations and other features [10].

A *resource* is any object which has a name (a URI) and can be accessed through an HTTP server. The HTTP protocol is built around the exchange of request and response *messages*. A message consists of a single control line which specifies the basic action or result represented by the message, zero or more headers which modify or refine the meaning of the message, and an optional *message body* carrying a (possibly encoded or transformed) representation of the state of some resource. A message may be either a *request* (a message asking that some action be performed upon a particular resource) or a *response* (a message reflecting the result of a request); *clients* are agents which transmit requests

*This research was supported in part by grants from the National Science Foundation (NSF), numbers ANI-9986397, ANI-0095988, CCR-9988529, ITR-0113193, ANI-0205294, and EIA-0202067.

and receive responses, while *servers* process requests by performing the requested action upon a resource and sending responses reflecting the results. A *proxy* is both a server and a client, forwarding requests and responses from its own clients to upstream servers and back. A *cache* is a store of responses to past requests, usually connected to a proxy or a client to reduce the number of interactions with servers and thereby improve performance.

2.1. Data Model

An HTTP message body is a transformed and encoded media object. The particular sequence of encodings and transformation applied to that object make up what is called the message's *data model* or (equivalently) *content model*; the set of all such encoding/transform sequences supported by the protocol is the protocol's data model or (equivalently) content model.

In addition to structuring the order of encodings and transforms, the data model attaches a structured set of semantics to various points within the encoding sequence. Specifically, we say that there are four *phases* to the HTTP data model: *variant*, *instance*, *entity*, and *message*;¹ each corresponds with a distinct set of semantics for agents which wish to interpret and use the content of a message body, as described below.

Variants A variant is a primitive value produced by a resource intended for direct presentation to a user, a non-HTTP-specific software component, or a resource. Variants usually take the form of individual documents, images, audio or video files in some well-known encodings (e.g., HTML, JPEG, MP3, and RealMedia, respectively), and no processing beyond that which is intrinsic to that encoding is needed to render the variant to the appropriate output device(s) or present it to the targeted application logic.

Instances An instance is a variant with zero or more *content-codings* applied to it. Content-codings are transforms which preserve the wholeness or semantic integrity of a variant/instance while altering the byte-wise content of its presentation; for example:

- Compression (e.g., `gzip`, `compress`, `deflate`)
- Wrapping content (encrypted, signed, or both) in a cryptographic envelope
- Integrity checks (e.g., MD5 checksums)

Because instances remain semantically whole representations, they are suitable (in principle) for shared caches to store and use to answer subsequent requests.

¹Our work builds upon the four-phase data model proposed by Mogul *et al* [14, 15] as a refinement of RFC2616's three-phase model.

Also note that a variant can always be coerced to become an instance, but not *vice versa*.

Entities An entity is an instance with zero or more instance-manipulations applied to it. An instance-manipulation is an operation which does not necessarily preserve the semantic integrity or "wholeness" of its operand. For example:

- Range selection (Content-Range header, `multipart/byteranges` Content-Type)
- Delta coding [11, 15]
- Cyclone coding [16]²
- Compression³
- Wrapping content in a cryptographic envelope
- Integrity checks (Content-MD5 header)

While an instance represents a whole thing (a representation which is *of itself* expressive of the state of the resource which produced it), an entity is not necessarily useful *of itself*, but rather may provide an application with enough information to be able to construct a complete instance from other information it also possesses (e.g., by providing a delta against a previous instance or filling a gap in a previous aborted download). More formally, content-codings will always be "lossless" with respect to the essential information of a representation, while instance-manipulations *may* be "lossy" transforms (such that the lost information must be acquired via a separate request or other out-of-band means). Because of their potential irreversibility (and the resulting potential inability to recover a usable instance or variant without additional out-of-band information), entities are not suitable for conventional shared caching.

Another distinction between an instance and an entity is that an instance is an *end-to-end* object from the application's perspective; upon successful completion of an HTTP transaction, the client and the origin server should (if the system has behaved properly) agree on the bitwise content of the instance in question. Entities, in contrast, are not necessarily end-to-end in this same sense, as proxies may interpret an entity and then send a very different entity on to their next inbound or outbound peer (e.g., patching a hole in the cache's

²Cyclone coding was proposed as a media-coding, *i.e.*, the variant would itself be a cyclone-coded object; see [2] for a critique of this design decision and motivation for classifying it as an instance-manipulation.

³Compression is one of several operators also mentioned as a content-coding and a transfer-coding; intuitively, this is required because a "compressed entity", a "compressed instance", and a "compressed message" have different meanings (inasmuch as entity, instance, and message have different meanings); e.g., a compressed delta must be handled differently than a compressed variant.

copy of an instance with one byterange, then transmitting a different but overlapping byterange requested by the client).

While in principle there is nothing wrong with layering multiple instance-manipulations upon one another, doing so is simply not supported by the HTTP/1.1 protocol [7], which supports only one transformative instance-manipulation (byterange selection, *i.e.*, “partial downloads”) and one annotative instance-manipulation (instance checksumming via the Content-MD5 header), and those two applied only in that order. This restriction is relaxed by a standards-track proposed extension to HTTP [15].

Messages As the final phase, a message is an entity with zero or more transfer-codings applied to it; this process is often called *serialization*. Transfer-codings are annotations and transformations of an entity or a message which allow a recipient to reliably delineate the underlying content, *e.g.*:

- Message Length (Content-Length header)
- Chunking (chunked Transfer-coding)
- Wrapping content in a self-delimiting (*e.g.*, ASCII armored) cryptographic envelope
- Compression (using self-delimiting encodings)

The message length transform (we model it as a transform, as it fixes unambiguously the boundaries of the stream) is simple enough to understand; in order to use whatever object is length-described, the recipient must receive the stated number of bytes. “Chunking” is a technique introduced in HTTP/1.1 to allow servers to begin transmission of messages before they can determine their length (*e.g.*, when a representation is being produced by a script) [10].

While it is possible in principle to send an entity without applying any meaningful transfer-codings, this is not advisable, as it prevents the recipient from having a reliable HTTP-level indicator of the end of the *message*; as such, it must treat the message as an unlimited byte source which happens to eventually be terminated by an out-of-band event (closure of the transport connection).

2.2 Syntax and Semantics

The intent of an HTTP message is to use headers to syntactically express the data model of the message body to allow a recipient to correctly interpret the message body; while the syntax of the HTTP protocol is well-defined [7] and the above data model has been clearly stated for the HTTP engineering community [15], the relationship between syntax (headers and their values) and some precise model of semantics (*e.g.*,

layered types) is stated strictly with english prose, opening the door to potential ambiguity, incompleteness, or contradiction within the specification.

In formal programming language theory, such precise relationships are often established by defining the *syntax-directed semantics* of a language, usually as a function which takes a block of syntax and returns some representation of the result, effects, or meaning of that syntax. We use layered types as a reference model for the semantics of HTTP message bodies and present several examples of syntax-directed semantics for different versions of the HTTP protocol. We are also able to present an example of a *semantic-directed syntax* function which translates models of desired semantics back into the standard HTTP syntax.

This exercise has several benefits. First, a precise model of semantics gives us a way to formally reason about such desirable properties as *interoperability* between revisions and extensions, something which has not always been diligently maintained⁴.

Second, the rules we develop for comparing particular message data models offer a useful run-time tool for assessing the usefulness of a given message as input to a given application; for example, if an application expresses its willingness to accept any message of the type “gzipped XML document” and it is sent a message of the type “gzipped XHTML document”, the traditional HTTP rules would rule this as an unacceptable mismatch;⁵ by contrast, we develop a system in which *subtyping* rules can be used to infer the suitability of one representation for use by an application requiring a representation with a differing description, and even to (in some cases) identify strategies for transforming a representation to make it usable to the application.

3. Layered Types

Our representation of the HTTP data model is based upon *layered types* [2]. The layered type system is a precise description of the composition of transformative and annotative operations in the construction of representations for use in protocols like (but not limited to) HTTP. Layered types describe latent or internal semantics and properties of data as it is operated upon in ways which preserve (perhaps in part) that meaning but alter the form, presentation, or encoding of the data, or which alter the set of annotations attached to that data.

In essence, a layered type preserves the history of transformative and descriptive operations performed

⁴This is evidenced by the potential deadlock conditions which emerged between HTTP/1.0 and HTTP/1.1 agents implementing the now-obsolete RFC2068 [3] as well as in initial proposals for persistent connections [13].

⁵More precisely, for this situation to proceed successfully would require the application to exhaustively list all MIME-types which have been assigned to XML-based encodings.

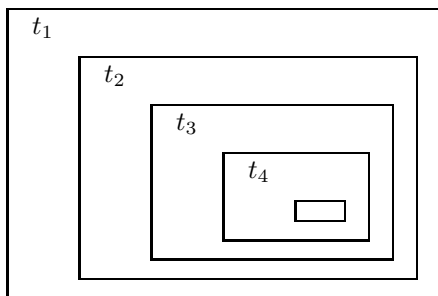


Figure 1. Layered Type $[t_1[t_2[t_3[t_4[]]]]]$

upon a representation with an unambiguous ordering of their application. Thus, it describes (and thereby restricts) not only the immediate interpretation of the representation, but also provides an implicit strategy for the management of the representation, its comparison and composition with other representations, and (in many cases) a strategy for applying inverted transforms to derive the more basic underlying data forms.

3.1. Syntax of Layered Types

Layered types are expressed using the *stacked type syntax* (developed and employed for several applications in [2]). Rather than representing the type of a value with a single symbol (e.g., int or bool), the stacked type syntax expresses the type of a value with a stack of such symbols, where the head of the stack represents the most immediate “layer” of semantics which must be correctly handled in order to reach the inner “layers” of semantics (the stack’s tail). This is represented graphically in Figure 1, where t_1 is the head (outermost) type which must be correctly interpreted to reach a value of type t_2 and so forth.

Layered types have the syntax:

$$\sigma ::= [stackel \ \sigma] \mid []$$

Intuitively, a layered type (σ) is a stack (list) of type elements (stack elements or *stackels*), each denoting a particular transformation or annotation, terminated by the empty-stack symbol ($[]$) at the tail. For example, $[t_1[t_2[t_3[t_4[]]]]]$ (the type illustrated in Figure 1) is a stack consisting of four *stackels* with t_1 as its head.

A useful construct for layered types is concatenation of type σ' onto the head of σ , written as $\sigma'\sigma$ and defined as:

$$\sigma'\sigma = \begin{cases} \sigma' & \text{if } \sigma = [] \\ \sigma & \text{if } \sigma \neq [] \text{ and } \sigma' = [] \\ [v \ \sigma'' \ \sigma] & \text{if } \sigma \neq [] \text{ and } \sigma' = [v \ \sigma''] \end{cases}$$

3.2. Layered Type Basics

When a value (a representation by another name) has a type $[s \ \sigma]$ for some s and some σ , it can be thought of as belonging to the set s . The value in question will (in general) be transformable into one of type σ by applying a function which removes, undoes, or decodes whatever transform is indicated by s .

For example, consider an HTML document compressed using the *gzip* algorithm. The contents of this file constitute a simple value (call it F); the layered type of this value would be $[gzipped[HTML[]]]$, indicating that the value must be *gunzipped* before the *HTML* can be interpreted.

For most conventional meanings of subtypes, the tail clearly encodes a subtype of whatever type information is identified by the head. Consider F above; in the set-theoretic sense, its type indicates not only that the value is a member of the set of valid *gzipped* streams (the head element), but that it is one of a subset of such streams which, when run through the *gunzip* algorithm, produce *HTML* documents (the tail).

We express subtypes using the conventional reflexive subtype operator $<$, where “ $a < b$ ” means “ a is a subtype of b ”. One could think of $<$ as having a similar meaning to \subseteq , in that “ $a < b$ ” means “the set of values described by a is a subset (not necessarily strict) of the set of values described by b ”.

We define the empty stack symbol $[]$ as carrying no type information, and thus the universal supertype. For all *stackels* s and all layered types σ , we state as axioms that

$$\sigma <: [] \quad (1)$$

and that

$$[s \ \sigma] <: [s \ []] . \quad (2)$$

A natural corollary of Equation 1 is that $[] <: []$ (which also follows from the reflexivity of $<:$).

We can establish subtype relationships among individual *stackels* of the form $s_1 <: s_2$, e.g.,

$$HTML3.2 <: HTML$$

and

$$byterange_{32KB@64KB} <: byterange_{8KB@72KB}$$

The latter example may seem counterintuitive at first, as the first type carries a *superset* of the bytes of the latter. Consider it instead in terms of *information content*: of all of the messages carrying the 8KB beginning at offset 72KB, those which carry 32KB beginning at offset 64KB are a strict subset.

We compose subtype relationships between layered types of the form $\sigma_1 <: \sigma_2$ using Equations 1 and 2 and

by induction using the rule

$$[s_1 \sigma_1] <: [s_2 \sigma_2] \quad (3)$$

where $s_1 <: s_2$ and $\sigma_1 <: \sigma_2$. To illustrate this, consider two gzipped XML documents, X_1 and X_2 . Suppose X_2 's underlying content is actually XHTML, a sub-language (subtype) of XML. Also suppose that X_2 is compressed using the most aggressive gzip setting (9). Then these two values are typed

$$X_1 : [\text{gzipped} [\text{XML} []]]$$

and

$$X_2 : [\text{gzipped}_9 [\text{XHTML} []]].$$

Intuitively, the type of X_2 is a subtype of X_1 , in that all XHTML documents are XML documents and all level-9 gzipped documents are gzipped documents. Since

$$\text{XHTML} <: \text{XML}$$

and

$$\text{gzipped}_9 <: \text{gzipped}$$

it follows from 3 that

$$[\text{XHTML} []] <: [\text{XML} []]$$

and subsequently (also from 3) that

$$[\text{gzipped}_9 [\text{XHTML} []]] <: [\text{gzipped} [\text{XML} []]].$$

This agrees with our intuition, that the information both about and within the outer *gzipped* layer allows us to establish a subtype relationship between types which are distinct at every layer.

3.3. Varieties of Stack Elements

The layered type system is useful for specifying a total ordering of operations performed in the preparation of a representation. Such operations are divided into three categories, which reflect distinct attributes of the way HTTP operates upon and describes message bodies: *transformative* operations, *annotative* operations, and *phase labels*.

Transformative operations are those which actually alter the content of the object being operated upon; the obvious example is the *gzip* compression algorithm. In essence, a transformative operation is any function t for which $t(x) \neq x$ for at least one representation x ; as a result, reliably retrieving x from $t(x)$ requires access to the inverse function, t^{-1} , such that $t^{-1}(t(x)) = x$ for all x (e.g., *gunzip* for *gzip*). Transformative operations are used in HTTP to support a number of features from compression to byterange selection to chunking and delta coding.

Annotative operations are those which provide ad-

ditional *side-band* information about a value which is not necessary to interpret the value itself. Annotative operations can be computationally trivial (counting the number of bytes in an object) or highly involved (computing public-key signatures or message digests), but they never change the content of the representation itself. HTTP supports several annotative operations, most notably the computation of MD5 checksums.

If a message body x has the type $[s \sigma]$ and s represents a transformative operation, then retrieving the underlying value of type σ will require that the inverse of that transform be applied to x . However, if s denotes an annotative operation, then s (intuitively) simply offers subtype information to the type σ ; in this case, x can safely be treated as having type σ . Formally, for every stackel a which is annotative and for every layered type σ ,

$$[a \sigma] <: \sigma. \quad (4)$$

Taken together with Equation 3, this can be thought of as a limited variation on the concept of *multiple inheritance* (a controversial concept in object-oriented programming) or alternately as an *intersection type* [5], by which the type of the expression is describable as a subtype both of the stack's head and of its tail. (Of course, this only works for annotative heads, not transformative ones.)

Furthermore, because annotative operations are idempotent with respect to message content, it is also true that for all annotative stack elements a_1 and a_2 and for every layered type σ that

$$[a_1 [a_2 \sigma]] <: [a_2 [a_1 \sigma]] \quad (5)$$

(i.e., annotative *stackels* are commutative).

Phase Labels Recall in Section 2.1 our description of the four *phases* of the HTTP data model: *variant*, *instance*, *entity*, and *message*. We represent each of the four phases by pushing a corresponding identifier (a *phase label*) onto a layered type stack to denote its relationship with the transforms and annotations applied to the representation. Under Mogul's four phase model, all HTTP messages can be described with layered types of the form:

$$\sigma_1 [\text{message } \sigma_2 [\text{entity } \sigma_3 [\text{instance } \sigma_4 [\text{variant } \sigma_5]]]].$$

Intuitively, an agent which receives a typed message need only interpret enough layers of that type to reach the phase-identifying element corresponding to the phase of the data model with which that particular agent is concerned. For example, caches are interested in instances, so for a message of the above type a cache need only possess capabilities to interpret $\sigma_1[\text{message } \sigma_2 [\text{entity } \sigma_3 []]]$ (and can be completely ambivalent to σ_4 and σ_5); a user-agent is interested in

variants and would need to also understand the data model of σ_4 , while a “pure” proxy (with no attached cache) is simply interested in receiving and forwarding entities and so would only need to comprehend messages of the type σ_1 [message σ_2 []].

Sublayers Recall our definition of a variant as “an instance with zero or more content-codings applied to it”. This intuitively suggests that $variant <: instance$ would be a reasonable subtype rule to introduce, and similarly $instance <: entity$ and $entity <: message$. However, such rules would be problematic in our system because their application *removes* one phase marker from the stack and replaces it with another. For example, consider a *variant* (call it V) with the type

$$[variant [Validated_{(URI)} [XML/1.1 []]]].$$

Using the subtype rule $variant <: instance$, we can also safely treat V as having the type

$$[instance [Validated_{(URI)} [XML/1.1 []]]].$$

Now, imagine that we apply the *gzip* operation to V , yielding a new representation V' with type

$$[instance [gzipped [Validated_{(URI)} XML/1.1 []]]].$$

This type has lost an important piece of information, namely, at what point in the stack we stop describing an *instance* and start describing the underlying *variant*: is the *variant* itself *gzip*-compressed, or was the compression performed as a content-coding?

Layer subtype rules like $variant <: instance$ are undesirable because type promotion rules would (for example) discard the *variant* element from the stack where it should be preserved to indicate the content model of the object at the time of its “type promotion”. To accomplish this, we introduce *sublayers* as a variation on traditional subtyping.

We write that a phase label t_1 is a sublayer of another phase label t_2 as:

$$t_1 \prec t_2.$$

The interpretation of this declaration is that t_1 always appears in the tail of any layered type with t_2 as its head. (Note that, unlike $<:$, the \prec relation is *not* reflexive.)

Sublayer declarations are used by the type inference rule:

$$\frac{x : [t_1 \sigma] \quad t_1 \prec t_2}{x : [t_2 [t_1 \sigma]]} \quad (6)$$

An equivalent statement would be that $t_1 \prec t_2$ implies $[t_1 \sigma] <: [t_2 [t_1 \sigma]]$; intuitively, a representation of type $[t_1 \sigma]$ can be promoted to have a type of the form $[t_2 \dots]$ by pushing t_2 onto the head of its type stack, but *not* by replacing t_1 with t_2 .

There are three sublayer relations defined for HTTP

layered types: $variant \prec instance$, $instance \prec entity$, and $entity \prec message$.

Other Phase Label Issues As mentioned in Section 3.3, HTTP’s practical features require that we distinguish between transformative and annotative stack elements. This raises an important question: are the phase labels (*variant*, *instance*, *entity*, and *message*) transformative or annotative, or should they be treated as something distinct?

As we have already stated, value-wise every *variant* can be thought of as also being an *instance*; this suggests that the phase labels are purely annotative. This supports several useful conclusions; *e.g.*,

$$[instance [variant \sigma]] <: [variant \sigma]$$

(*i.e.*, an instance with no content-codings can be treated as a variant). Taken together with the $variant \prec instance$ rule, this suggests that

$$[instance [variant \sigma]] \equiv [variant \sigma]$$

(*i.e.*, that the two types are equivalent). Treating phase labels as annotations also allows us to take advantage of commutativity of annotations in interesting ways, *e.g.*,

$$\sigma [instance [checksum_{(0xfedcba98)} \sigma']] <: \sigma [checksum_{(fedcba98)} [instance \sigma']]$$

for all σ and σ' , *i.e.*, an annotation applied to an *instance* equivalently describes an *entity* to which no transformative content-codings have been applied.

However, the commutativity of annotations would also lead to such nonsensical conclusions as

$$[variant [instance \sigma]] <: [instance [variant \sigma]].$$

Therefore, we prefer to define phase labels as neither annotative nor transformative, but as their own third class of stack elements, and specify subtype relationships which retain the desirable properties above while avoiding problems with commutativity. Specifically, for any phase labels p and p' and any σ ,

$$[p [p' \sigma]] <: [p' \sigma], \quad (7)$$

and for any phase label p , any annotative stack element a , and any σ ,

$$[p [a \sigma]] <: [a [p \sigma]] \quad (8)$$

$$[a [p \sigma]] <: [p [a \sigma]] \quad (9)$$

Thus, properties like $[instance [variant \sigma]] \equiv [variant \sigma]$ and commutativity of phase labels with annotations are retained without allowing phase labels to commute with each other.

4. Syntax-Directed HTTP Semantics

In this section we formalize the data models of different versions of the HTTP protocol through the use of a syntax-directed type inference algorithms defined over the HTTP messages syntax. We refer to these inference algorithms as *interpreters*, as they (in a real sense) reflect an abstract model of the process an HTTP agent goes through in *interpreting* a message to derive its own internal representation of that message's data model.

Taken together with the typing rules given in the previous section, this unambiguously declares the set of correct interpretations for a given message, allowing us to reason about correctness and interoperability with precision. A complete presentation of a suite of such formalisms, along with interpreters for Mogul's proposed modifications [15, 12] and a novel HTTP variant explicitly serializing layered types (and thus substantially broadening the HTTP data model), can be found in [2].

4.1. Interpreting HTTP/1.0 Messages

HTTP/1.0 [1] has a very simple data model for representations; a representation has a basic media type, no more than one content-coding, and delimits messages with a byte count.⁶ A suitable interpreter $\mathcal{I}_{1.0}[\cdot \cdot \cdot]$ can be defined as in Figure 2 (with its helper function $\mathcal{I}'_{1.0}[\cdot \cdot \cdot]$ in Figure 3); these functions are evaluated using a first-match strategy to simplify their presentation.

4.2. Interpreting HTTP/1.1 Messages

Because of the *ad hoc* way HTTP headers have been added and mapped onto its data model, the rules for interpreting a conventional HTTP/1.1 header block must of necessity be far more complicated than those for HTTP/1.0 messages. The function $\mathcal{I}_{1.1}[\cdot \cdot \cdot]$ interprets HTTP/1.1 headers to construct a layered type for the attached message body. Headers provide the bulk of the type information, although for response messages the request method (GET, HEAD, PUT, POST, *etc*) and response status code (200, 304, 404, 500, *etc*) can both significantly alter the meaning of the response message and the type it should be assigned.

The function $\mathcal{I}_{1.1}[\cdot \cdot \cdot]$ wraps the interpretation algorithm by taking two arguments: the syntax of the message to be interpreted and a boolean flag indicating whether this message is a response to a HEAD request. (Naturally, this boolean value will always be

⁶HTTP/1.0 also allows response messages to be delimited by closing the transport connection; as noted previously, this makes a complete representation indistinguishable from an aborted or incomplete one, so we do not include it in this model.

false when interpreting a request.) This function then returns the message body with an appropriate type decoration. To simplify the specification of this function, the first-match rule is used when identifying the applicable case for any given arguments. $\mathcal{I}_{1.1}[\cdot \cdot \cdot]$ is specified in Figure 4; the symbol “*either*” matches both *true* and *false* values.

The helper function, $\mathcal{I}'_{1.1}[\cdot \cdot \cdot]$ (stated in Figure 5), takes as an argument a block of HTTP syntax (headers followed by a potentially empty message body) decorated with a layered type, and returns the message body with the complete layered type appropriate to the initial type and the headers present in the syntax block. Use of the first-match rule again simplifies the specification.

These functions actually reflect several disambiguations of the HTTP standard (RFC2616). For example, there is an apparent conflict in the specification of the 206 response code, in that it is defined only as a valid response to the GET method; this (arguably) excludes it as a valid response to HEAD, but that in turn raises a contradiction with the definition of HEAD (which is supposed to duplicate the results of a GET with the message body repressed, and “SHOULD” bear the same meta-data). Our formalization errs on the side of allowing 206 responses to HEADs; this allows us to understand responses to HEAD requests indicating that the server would have transmitted a message with partial content had the request been a GET.

In a similar spirit, while HTTP/1.1 seems to define byterange instance-manipulations strictly for 206 responses, our interpreter follows the IETF interoperability principle (“be conservative in what you send, liberal in what you accept”) by including rules by which the Content-Range header or multipart/byteranges media type can be used to signal the use of this instance manipulation regardless of the response status-code, provided no other instance-manipulations have been applied. This specifies an interpretation of messages for which HTTP/1.1's semantics are simply undefined; as stated, it also enables our interpreter to attach a type to messages which are explicitly forbidden by the specification (*e.g.*, a response with the 416 status code must not use the multipart/byteranges type), but this can be explicitly excluded with a fairly verbose (but not terribly illuminating) change to the function.

4.3. Formalizing Interoperability

This section presents a model for partially assessing the interoperability of HTTP agents' content models based upon the layered type model. When speaking of interoperability between HTTP agents, we have three criteria in mind: completeness, soundness, and agreement.

Capability Completeness For two agents to interoper-

$$\begin{aligned}
& \mathcal{I}_{1.0} \llbracket \text{Request-Line Headers MessageBody, false} \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : [message [entity [instance [variant []]]]] \rrbracket \\
& \mathcal{I}_{1.0} \llbracket \text{HTTP-Version SP "204" SP Reason-Phrase CRLF Headers MessageBody, either} \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : [T-repress [message [entity [instance [variant []]]]] \rrbracket \\
& \mathcal{I}_{1.0} \llbracket \text{HTTP-Version SP "304" SP Reason-Phrase CRLF Headers MessageBody, either} \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : [T-repress [message [entity [instance [variant []]]]] \rrbracket \\
& \mathcal{I}_{1.0} \llbracket \text{Status-Line Headers MessageBody, false} \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : [message [entity [instance [variant []]]]] \rrbracket \\
& \mathcal{I}_{1.0} \llbracket \text{Status-Line Headers MessageBody, true} \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : [T-repress [message [entity [instance [variant []]]]] \rrbracket
\end{aligned}$$

Figure 2. The $\mathcal{I}_{1.0} \llbracket \cdot \cdot \cdot \rrbracket$ function

$$\begin{aligned}
& \mathcal{I}'_{1.0} \llbracket \text{"Content-Type:" media-type CRLF Headers MessageBody : } \sigma_1 \llbracket \text{variant []} \rrbracket \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : } \sigma_1 \llbracket \text{variant [T-content}_{\text{media-type}} \llbracket \cdot \cdot \cdot \rrbracket \rrbracket \rrbracket \\
& \mathcal{I}'_{1.0} \llbracket \text{"Content-Encoding:" content-coding CRLF Headers MessageBody : } \sigma_1 \llbracket \text{instance [variant } \sigma_2 \rrbracket \rrbracket \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : } \sigma_1 \llbracket \text{instance [T-content-coding [variant } \sigma_2 \rrbracket \rrbracket \rrbracket \\
& \mathcal{I}'_{1.0} \llbracket \text{"Content-Length:" digits CRLF Headers MessageBody : } \sigma_1 \llbracket \text{message } \sigma_2 \rrbracket \rrbracket \\
& = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : } \sigma_1 \llbracket \text{T-length}_{\text{digits}} \llbracket \text{message } \sigma_2 \rrbracket \rrbracket \rrbracket \\
& \\
& \mathcal{I}'_{1.0} \llbracket \text{Header Headers MessageBody : } \sigma \rrbracket = \mathcal{I}'_{1.0} \llbracket \text{Headers MessageBody : } \sigma \rrbracket \\
& \mathcal{I}'_{1.0} \llbracket \text{CRLF MessageBody : [T-rcr } \sigma \rrbracket \rrbracket = \mathcal{I}'_{1.0} \llbracket \text{CRLF MessageBody : } \sigma \rrbracket \\
& \mathcal{I}'_{1.0} \llbracket \text{CRLF MessageBody : [message [entity [instance [variant []]]]] \rrbracket = \llbracket \text{T-nil []} \rrbracket \\
& \mathcal{I}'_{1.0} \llbracket \text{CRLF MessageBody : } \sigma \rrbracket = \sigma
\end{aligned}$$

Figure 3. The $\mathcal{I}'_{1.0} \llbracket \cdot \cdot \cdot \rrbracket$ helper function

ate, each must be able to produce a message type which can be correctly interpreted by the other, *i.e.*, if A produces a message M of type τ_A , then B must be able to understand a message of type τ_B where $\tau_A <: \tau_B$.

Interpretive Soundness Possession of a particular capability (*e.g.*, the ability to decompress a gzip-encoded stream) does not mean that the agent will necessarily know if or when to employ that capability (*e.g.*, to identify the appropriate point in the interpretation pipeline to apply the decompression).

We define a *serializer* as a software agent which transforms some representation of a message's data model into an actual block of HTTP syntax (*i.e.*, a semantic-directed syntax function). For two agents to interoperate, one must *serialize* the message in such a way that the recipient can both (syntactically) understand the description and (semantically) discern that the message body is of an agreeable type; *i.e.*, if A produces from the type τ_A

a message using a serializer $\mathcal{S}_A \llbracket \cdot \cdot \cdot \rrbracket$ and B uses the interpreter $\mathcal{I}_B \llbracket \cdot \cdot \cdot \rrbracket$, then

$$\tau_A <: \mathcal{I}_B \llbracket \mathcal{S}_A \llbracket M : \tau_A \rrbracket \rrbracket$$

Intuitively, the serialization/deserialization pipeline may lose type information about the message so long as it does not introduce spurious or unintended type information which would incorrectly alter the interpretation of the message's contents.

Agreement between Interpretation and Capabilities

Notice that an agent may be able to discern the intended semantics of a message without possessing the capabilities to then interpret the content of the message. (*e.g.*, the agent may recognize that the content is *bzip*-compressed but lack the necessary algorithm to decompress it.) For two agents to interoperate, they must share the capabilities understood by the recipient as the model of the

```

 $\mathcal{I}_{1,1}$  [Request-Line Headers MessageBody, false]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [message [entity [instance [variant []]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "204" SP Reason-Phrase CRLF Headers MessageBody, either]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-repress [message [entity [instance [variant []]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "205" SP Reason-Phrase CRLF Headers MessageBody, either]
= [T-nil []]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "206" SP Reason-Phrase CRLF Headers MessageBody, false]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [message [entity [T-range [instance [variant []]]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "206" SP Reason-Phrase CRLF Headers MessageBody, true]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-repress [message [entity [T-range [instance [variant []]]]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "304" SP Reason-Phrase CRLF Headers MessageBody, either]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-repress [message [entity [instance [variant []]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "416" SP Reason-Phrase CRLF Headers MessageBody, false]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-rcr [message [entity [instance [variant []]]]]]
 $\mathcal{I}_{1,1}$  [HTTP-Version SP "416" SP Reason-Phrase CRLF Headers MessageBody, true]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-rcr [T-repress [message [entity [instance [variant []]]]]]]]
 $\mathcal{I}_{1,1}$  [Status-Line Headers MessageBody, false]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [message [entity [instance [variant []]]]]]
 $\mathcal{I}_{1,1}$  [Status-Line Headers MessageBody, true]
=  $\mathcal{I}'_{1,1}$  [Headers MessageBody : [T-repress [message [entity [instance [variant []]]]]]]]

```

Figure 4. The $\mathcal{I}_{1,1} [\dots]$ function

message's content, *i.e.*,

$$\mathcal{I}_B [\mathcal{S}_A [M : \tau_A]] <: \tau_B$$

for some τ_B comprehensible to B . In words, B can soundly discover the intended semantics of M and possesses the capabilities necessary to interpret M itself correctly.

In summary, for any messages M of type τ_A sent by agent A , the necessary and sufficient condition for agent B to be able to correctly interpret them is precisely B 's ability to interpret messages of some type τ_B where

$$\tau_A <: \mathcal{I}_B [\mathcal{S}_A [M : \tau_A]] <: \tau_B$$

For example, with respect to capability completeness a message of type τ_1 can (theoretically) always be correctly interpreted by an HTTP agent which understands messages of type τ_2 if $\tau_1 <: \tau_2$. So if a server sends a message with the type

```

[message [T-chunked [entity [T-delta-coded
[instance [A-MD5 [T-gzipped [variant
[T-contentapplication/xml []]]]]]]]]]]

```

it must be careful only to do so when communicating with a client which understands messages of the type

```

[message [T-chunked [entity [T-delta-coded
[instance [T-gzipped [variant
[T-contentapplication/xml []]]]]]]]]

```

or it will risk the client incorrectly interpreting and presenting the message.

4.4. Protocol Correctness

There are two spheres of correctness we wish to use this formalism to assess. The first is with respect to a protocol itself; the specification (call it A) defines a canonical message interpreter $\mathcal{I}_A [\dots]$ for converting HTTP syntax into layered types and a generic serializer $\mathcal{S}_A [\dots]$ which accepts as an argument every layered type supported by the protocol's data model and outputs (non-deterministically) every possible syntactic representation of each. Given these two functions, the correctness of the protocol itself rests upon these two criteria:

1. The specification does not allow construction of any messages which the canonical interpreter cannot handle, *i.e.*, the range of $\mathcal{S}_A [\dots]$ must be a subset of the domain of $\mathcal{I}_A [\dots]$.

$$\begin{aligned}
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Content-Type:" multipart/byteranges" ";" "boundary=" BoundaryString CRLF} \\ \text{Headers MessageBody: } \sigma_1 [T\text{-range } \sigma_2] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [T\text{-range}_{\text{BoundaryString}}^{\text{multipart}} \sigma_2] \right] \\
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Content-Type:" multipart/byteranges" ";" "boundary=" BoundaryString CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{entity } [\text{instance } \sigma_2]] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{entity } [T\text{-range}_{\text{BoundaryString}}^{\text{multipart}} [\text{instance } \sigma_2]]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Type:" media-type CRLF Headers MessageBody: } \sigma_1 [\text{variant } []] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{variant } [T\text{-content}_{\text{media-type}} []]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Range:" content-range-spec CRLF Headers MessageBody: } \sigma_1 [T\text{-rcr } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 \sigma_2 \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Range: */" instance-length CRLF Headers MessageBody: } \sigma_1 [\text{instance } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [T\text{-length}_{\text{instance-length}} [\text{instance } \sigma_2]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Range:" byte-range-resp-spec "/"* CRLF Headers MessageBody: } \sigma_1 [T\text{-range } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [T\text{-range}_{\text{byte-range-resp-spec}} \sigma_2] \right] \\
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Content-Range:" byte-range-resp-spec "/" instance-length CRLF} \\ \text{Headers MessageBody: } \sigma_1 [T\text{-range } \sigma_2 [\text{instance } \sigma_3]] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [T\text{-range}_{\text{byte-range-resp-spec}} \sigma_2 [T\text{-length}_{\text{instance-length}} [\text{instance } \sigma_3]]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Range:" byte-range-resp-spec "/"* CRLF Headers MessageBody: } \sigma_1 [\text{entity } [\text{instance } \sigma_2]] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{entity } [T\text{-range}_{\text{byte-range-resp-spec}} [\text{instance } \sigma_2]]] \right] \\
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Content-Range:" byte-range-resp-spec "/" instance-length CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{entity } [\text{instance } \sigma_2]] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{entity } [T\text{-range}_{\text{byte-range-resp-spec}} [T\text{-length}_{\text{instance-length}} [\text{instance } \sigma_2]]]] \right] \\
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Content-Encoding:" content-coding "," 1#content-coding CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{instance } \sigma_2] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\begin{array}{l} \text{"Content-Encoding:" 1#content-coding CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{instance } [T\text{-content-coding } \sigma_2]] \end{array} \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Encoding:" content-coding CRLF Headers MessageBody: } \sigma_1 [\text{instance } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{instance } [T\text{-content-coding } \sigma_2]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-MD5:" md5-digest CRLF Headers MessageBody: } \sigma_1 [\text{entity } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [A\text{-md5}_{\text{md5-digest}} [\text{entity } \sigma_2]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Content-Length:" digits CRLF Headers MessageBody: } \sigma_1 [\text{message } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [T\text{-length}_{\text{digits}} [\text{message } \sigma_2]] \right] \\
\mathcal{I}'_{1,1} & \left[\begin{array}{l} \text{"Transfer-Encoding:" transfer-coding "," 1#transfer-coding CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{message } \sigma_2] \end{array} \right] \\
& = \mathcal{I}'_{1,1} \left[\begin{array}{l} \text{"Transfer-Encoding:" 1#transfer-coding CRLF} \\ \text{Headers MessageBody: } \sigma_1 [\text{message } [T\text{-transfer-coding } \sigma_2]] \end{array} \right] \\
\mathcal{I}'_{1,1} & \left[\text{"Transfer-Encoding:" transfer-coding CRLF Headers MessageBody: } \sigma_1 [\text{message } \sigma_2] \right] \\
& = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma_1 [\text{message } [T\text{-transfer-coding } \sigma_2]] \right] \\
\mathcal{I}'_{1,1} & \left[\text{Header Headers MessageBody: } \sigma \right] = \mathcal{I}'_{1,1} \left[\text{Headers MessageBody: } \sigma \right] \\
\mathcal{I}'_{1,1} & \left[\text{CRLF MessageBody: } [T\text{-rcr } \sigma] \right] = \mathcal{I}'_{1,1} \left[\text{CRLF MessageBody: } \sigma \right] \\
\mathcal{I}'_{1,1} & \left[\text{CRLF MessageBody: } [\text{message } [\text{entity } [\text{instance } [\text{variant } []]]]] \right] = [T\text{-nil } []] \\
\mathcal{I}'_{1,1} & \left[\text{CRLF MessageBody: } \sigma \right] = \sigma
\end{aligned}$$

Figure 5. The $\mathcal{I}'_{1,1} [\dots]$ helper function

$$\begin{aligned}
S_{1.1} \llbracket \sigma [T-x [\text{variant } \sigma']] \rrbracket &= \text{"Content-Encoding: } x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{variant } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [A-x [\text{variant } \sigma']] \rrbracket &= S_{1.1} \llbracket \sigma [\text{variant } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [T-length_x [\text{message } \sigma']] \rrbracket &= \text{"Content-Length: } x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{message } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [A-x [\text{message } \sigma']] \rrbracket &= S_{1.1} \llbracket \sigma [\text{message } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [A-x [\text{entity } \sigma']] \rrbracket &= S_{1.1} \llbracket \sigma [\text{entity } \sigma'] \rrbracket \text{ for } x \neq md5 \\
S_{1.1} \llbracket \sigma [A-md5_x \sigma' [\text{entity } \sigma'']] \rrbracket &= S_{1.1} \llbracket \sigma \sigma' [\text{entity } \sigma''] \rrbracket \text{ for } \sigma' \neq [] \\
S_{1.1} \llbracket \sigma [A-md5_x [\text{entity } \sigma']] \rrbracket &= \text{"Content-MD5: } x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{entity } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [T-length_x [\text{instance } \sigma']] \rrbracket &= \text{"Content-Range: } */x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{instance } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [\text{entity } [T-range_x [T-length_y [\text{instance } \sigma']]]] \rrbracket &= \text{"Content-Range: } x/y \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{entity } [\text{instance } \sigma']] \rrbracket \\
S_{1.1} \llbracket \sigma [\text{entity } [T-range_x [\text{instance } \sigma']]] \rrbracket &= \text{"Content-Range: } x/* \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{entity } [\text{instance } \sigma']] \rrbracket \\
S_{1.1} \llbracket \sigma [T-range_x^{\text{multipart}} [\text{instance } \sigma' T-content_x]] \rrbracket &= \\
&\text{"Content-Type: multipart/byteranges"} \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{instance } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [T-content_x] \rrbracket &= \text{"Content-Type: } x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma \rrbracket \\
S_{1.1} \llbracket \sigma [T-x [\text{entity } \sigma']] \rrbracket &= \text{"Transfer-Encoding: } x \cdot \text{CRLF} \cdot S_{1.1} \llbracket \sigma [\text{entity } \sigma'] \rrbracket \\
S_{1.1} \llbracket \sigma [A-x \sigma'] \rrbracket &= S_{1.1} \llbracket \sigma \sigma' \rrbracket \\
S_{1.1} \llbracket \sigma [T-x \sigma'] \rrbracket &= \text{Error: unsupported content model} \\
S_{1.1} \llbracket [] \rrbracket &= \text{CRLF}
\end{aligned}$$

Figure 6. The $S_{1.1} \llbracket \cdot \cdot \cdot \rrbracket$ function

2. $S_A \llbracket \cdot \cdot \cdot \rrbracket$ and $\mathcal{I}_A \llbracket \cdot \cdot \cdot \rrbracket$ are sound (as defined above).

The broadness of $S_A \llbracket \cdot \cdot \cdot \rrbracket$ may give the impression that proving these two properties will be highly difficult, but this is in fact not the case. HTTP declares that many syntactically differing messages are semantically equivalent, and if we can prove these equivalences for $\mathcal{I}_A \llbracket \cdot \cdot \cdot \rrbracket$ we can significantly reduce the size of the message set to be considered to include only a single *canonical form* for every such equivalence class. For example:

- Header names are case-insensitive. We assume the rules capture this, and consider only the canonical spellings used in the RFC.
- There is tremendous liberty in the use of white space in headers. We assume the rules capture this, and consider only a single space as representing all linear white space strings.
- Value-list headers may present their lists using one or several headers, *i.e.*,

Transfer-Encoding: gzip, chunked

is equivalent to

Transfer-Encoding: gzip
Transfer-Encoding: chunked

Notice (for example) Figure 5's rules for Content-Encoding and Transfer-Encoding capture this, so it is safe to consider only one-value-per-line presentation.

- Header ordering is very liberal; so long as instances of value-list headers (see immediately above) are kept in order, there are no ordering rules between headers. Our use of the $\sigma [match-string \sigma']$ form in the interpretive rules makes them largely agnostic to ordering, so it is safe to consider only an alphabetical header ordering.

A portion of the function $S_{1.1} \llbracket \cdot \cdot \cdot \rrbracket$ is presented in Figure 6 as a first-match rule set as a partial so-reduced model of HTTP/1.1 serialization. (We have omitted the sections dealing, *e.g.*, with unusual request methods like HEAD and unusual response status codes like 304, and instead present only those rules which pertain to header construction in normal request and response messages.) The proof that the complete form of this function and $\mathcal{I}_{1.1} \llbracket \cdot \cdot \cdot \rrbracket$ satisfy the above two correctness properties is straightforward but voluminous, and is omitted here for want of space.⁷

⁷The proof, along with a complete declaration of $S_{1.0} \llbracket \cdot \cdot \cdot \rrbracket$ and $S_{1.1} \llbracket \cdot \cdot \cdot \rrbracket$, will be published in a technical report version of this paper.

Interoperability These same techniques can be used to test the backward-compatibility of a protocol. Consider HTTP/1.0 and 1.1, represented by the function pairs $\langle \mathcal{S}_{1.0}[\cdot\cdot\cdot], \mathcal{I}_{1.0}[\cdot\cdot\cdot] \rangle$ and $\langle \mathcal{S}_{1.1}[\cdot\cdot\cdot], \mathcal{I}_{1.1}[\cdot\cdot\cdot] \rangle$, respectively. We can say that 1.1 is backward-compatible with 1.0 if for every message content model σ in the domain of $\mathcal{S}_{1.0}[\cdot\cdot\cdot]$,

$$\sigma <: \mathcal{I}_{1.0}[\mathcal{S}_{1.1}[\sigma]]$$

and

$$\sigma <: \mathcal{I}_{1.1}[\mathcal{S}_{1.0}[\sigma]]$$

(i.e., a 1.1 agent can always understand a 1.0 agent, and a 1.0 agent can always understand a 1.1 agent when capability completeness is satisfied), which is in fact the case.

4.5. Implementation Correctness

Similarly, we are concerned with assessing whether a *particular implementation* of the protocol is correct with respect to the formalized specification. Call the implementation Z , and the standard it aspires to R ; its correctness rests upon both of the following criteria:

1. Z 's syntax interpreter $\mathcal{I}_Z[\cdot\cdot\cdot]$ agrees with the reference interpreter $\mathcal{I}_R[\cdot\cdot\cdot]$ (e.g., $\mathcal{I}_{1.1}[\cdot\cdot\cdot]$) for all messages within the scope of the protocol. Formally, for every syntax block x in the range of $\mathcal{S}_R[x]$, either $\mathcal{I}_R[x] <: \mathcal{I}_Z[x]$ or $\mathcal{I}_Z[x]$ is undefined (the interpreter rejects the message).
2. The implementation's serializer $\mathcal{S}_Z[\cdot\cdot\cdot]$ is a correct instantiation of the reference serialization model $\mathcal{S}_R[\cdot\cdot\cdot]$, i.e., for every x in the intersection of the domain of $\mathcal{S}_Z[\cdot\cdot\cdot]$ and $\mathcal{S}_R[\cdot\cdot\cdot]$, $\mathcal{S}_Z[x]$ only produces syntax that could have been produced by $\mathcal{S}_R[x]$.

Proving this for a given implementation requires extracting models of $\mathcal{S}_Z[\cdot\cdot\cdot]$ and $\mathcal{I}_Z[\cdot\cdot\cdot]$ directly from the code; while this task is certainly not trivial, neither does it seem intractable [6, 9].

5. Summary

We have proposed *layered types* as a formalism for specifying the content model semantics of representational protocols such as HTTP. By capitalizing upon well-understood concepts such as subtypes and introducing novel type constructs (such as *sublayers*) to represent particulars of the kernel of HTTP's content model, we are able to define both a type-based syntax-directed semantics for the payloads of HTTP protocol messages and a precise notion of correctness and interoperability against which new revisions and extensions to the protocol can be compared and tested.

References

- [1] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC1945, May 1996.
- [2] A. D. Bradley. *A Type-Disciplined Approach to Developing Resources and Applications for the World-Wide Web*. PhD thesis, Boston University, 2004.
- [3] A. D. Bradley, A. Bestavros, and A. J. Kfoury. Safe composition of web communication protocols for extensible edge services. In *Workshop on Web Caching and Content Delivery (WCW)*, Boulder, CO, Aug. 2002.
- [4] A. D. Bradley, A. Bestavros, and A. J. Kfoury. Systematic verification of safety properties of arbitrary network protocol compositions using CHAIN. In *IEEE Conference on Network Protocols (ICNP)*, Atlanta, GA, Nov. 2003.
- [5] A. B. Compagnoni and B. C. Pierce. Multiple inheritance via intersection types. Technical Report ECS-LFCS-93-275, LFCS, University of Edinburgh, Aug. 1993.
- [6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC2616, June 1999.
- [8] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, pages 115–150, May 2002.
- [9] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000. Also appeared 4th SPIN workshop, Paris, November, 1998.
- [10] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the WWW-8 Conference*, Toronto, May 1999.
- [11] J. Mogul, F. Douglis, A. Feldman, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM '97*, pages 181–194, Cannes, France, Aug. 1997. ACM.
- [12] J. Mogul and A. V. Hoff. Instance digests in HTTP, Jan. 2002. RFC3230.
- [13] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM '95*. ACM, Aug. 1995.
- [14] J. C. Mogul. Clarifying the fundamentals of HTTP. In *WWW-2002*, Honolulu, HI, May 2002.
- [15] J. C. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. an Hoff, and D. Hellerstein. Delta encoding in HTTP, Jan. 2002. RFC3229.
- [16] S. Rost, J. Byers, and A. Bestavros. The cyclone server architecture: Streamlining delivery of popular content. In *Sixth International Workshop on Web Caching and Content Distribution*, Boston, June 2001.