Workgroup: Network Working Group
Internet-Draft:
draft-petithuguenin-computerate-specifying-17
Published: 7 August 2022
Intended Status: Experimental
Expires: 8 February 2023
Authors: M. Petit-Huguenin
Impedance Mismatch LLC

The Computerate Specifying Paradigm

Abstract

This document specifies a paradigm named Computerate Specifying, designed to simultaneously document and formally specify communication protocols. This paradigm can be applied to any document produced by any Standard Developing Organization (SDO), but this document targets specifically documents produced by the IETF.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 February 2023.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
2. Overview
3. Terminology
4. Basic Specification Tutorial
5. Verified Specification Tutorial
   5.1. Evidence-Based Answers
      5.1.1. Encoding Questions
         5.1.1.1. Any Value of a Type is Evidence of Yes
         5.1.1.2. Function Type As Implication
         5.1.1.3. Polymorphism
         5.1.1.4. Empty Type as No
         5.1.1.5. Sloppy Questions
         5.1.1.6. Product Type
         5.1.1.7. Sum Type
         5.1.1.8. Inductive Type
         5.1.1.9. Pi Type
         5.1.1.10. Sigma Type
         5.1.1.11. Equality Type
         5.1.1.12. Decidable Type
      5.1.2. How to Find Evidence
   5.2. PDU Descriptions
      5.2.1. PDU Examples
      5.2.2. Calculations from PDU
      5.2.3. PDU Representations
   5.3. State Machines
   5.4. Proofs
      5.4.1. Wire Type vs Semantic Type
      5.4.2. Data Format Conversion
      5.4.3. Postel's Law
      5.4.4. Implementability
      5.4.5. Termination
      5.4.6. Liveness
6. Standard Library Tutorial
   6.1. Internal Modules
      6.1.1. AsciiDoc Generation
      6.1.2. Bit-Vectors
      6.1.3. Abstract Numbers
      6.1.4. Denominate Numbers
      6.1.5. Typed Petri Nets
         6.1.5.1. Building a Typed Petri Net
         6.1.5.2. Adding Time to a Typed Petri Net
         6.1.5.3. Verifying a Typed Petri Net
         6.1.5.4. Deriving a Type from a Typed Petri Net
         6.1.5.5. Message Sequence Charts
   6.2. Formal Language Packages
      6.2.1. Formal Languages Defined in RFCs
         6.2.1.1. Augmented BNF (ABNF)
         6.2.1.2. Structured Field Values for HTTP
7. Specification Package Tutorial

8. Standard Library Reference
  8.1. Internal Modules
    8.1.1. Metanorma.Ietf
    8.1.2. BitVector
    8.1.3. Unsigned
    8.1.4. Dimension
    8.1.5. Tpn
      8.1.5.1. Building a TPN
      8.1.5.2. Verifying a TPN
      8.1.5.3. Deriving a Type From a TPN
  8.2. Formal Language Packages
    8.2.1. RFC 5234 (ABNF)
      8.2.1.1. Building an ABNF
      8.2.1.2. Generating and Verifying ABNF specifications
      8.2.1.3. Common Rules
    8.2.2. RFC 8941 (Structured Field Values for HTTP)
9. Informative References
Appendix A. Command Line Tools
  A.1. Installation
  A.2. Authoring a Computerate Specification
    A.2.1. Using the Templates
    A.2.2. Converting an Existing Document
    A.2.3. Bibliography
      A.2.3.1. Build a Bibliography with Zotero
      A.2.3.2. Build a Bibliography Manually
  A.3. Processing a Computerate Specification
    A.3.1. Outputs
  A.4. Other Commands
  A.5. Modified Tools
    A.5.1. Idris2
    A.5.2. asciidoctor
    A.5.3. metanorma
    A.5.4. metanorma-ietf
    A.5.5. mnconvert
    A.5.6. xml2rfc
    A.5.7. idris2-vim
  A.6. Bugs and Workarounds
  A.7. TODO List
Appendix B. Standard Library API Documentation
  B.1. Package computerate-specifying
    B.1.1. Module ComputerateSpecifying.BitVector
    B.1.2. Module ComputerateSpecifying.Dimension
    B.1.3. Module ComputerateSpecifying.Metanorma.Ietf
    B.1.4. Module ComputerateSpecifying.Tpn
    B.1.5. Module ComputerateSpecifying.Unsigned
  B.2. Package rfc5234
    B.2.1. Module RFC5234
    B.2.2. Module RFC5234.Core
  B.3. Package rfc8941
    B.3.1. Module RFC8941
Appendix C. Errata Statistics

Appendix D. Converting From a Colored Petri Net
  D.1. Convert Color Sets
    D.1.1. Simple Color Sets
      D.1.1.1. Unit Color Sets
      D.1.1.2. Boolean Color Sets
      D.1.1.3. Integer Color Sets
      D.1.1.4. Large Integer Color Sets
      D.1.1.5. Real Color Sets
      D.1.1.6. String Color Sets
      D.1.1.7. Enumerated Color Sets
      D.1.1.8. Index Color Sets
    D.1.2. Compound Color Sets
      D.1.2.1. Product Color Sets
      D.1.2.2. Record Color Sets
      D.1.2.3. List Color Sets
      D.1.2.4. Union Color Sets
      D.1.2.5. Subset Color Sets
      D.1.2.6. Alias Color Sets
    D.1.3. Timed Color Sets
    D.1.4. Size of Color Sets
  D.2. Convert Places
  D.3. Convert Transitions
    D.3.1. Convert Arcs
      D.3.1.1. Convert Free Variables
      D.3.1.2. Convert Input Arcs
      D.3.1.3. Convert Inhibitor Arcs
      D.3.1.4. Convert Reset Arcs
      D.3.1.5. Convert Output Arcs
    D.3.2. Convert Transition Inscription
      D.3.2.1. Unification
      D.3.2.2. Guards
  D.4. Convert Substitution Transitions
  D.5. Convert Fusion Places
Appendix E. A Distributed Package Manager for Computerate Specifications
  E.1. Distributed Source Repositories
    E.1.1. The "gits" Protocol
    E.1.2. The "mgit" and "mgits" Protocols
    E.1.3. Git Submodules as Dependencies
  E.2. Distributed Artifact Manager
    E.2.1. Reproducible Build
    E.2.2. Distributed Cache
  E.3. Recursive Build
    E.3.1. Indexing Commits
    E.3.2. Building a Submodule
    E.3.3. Pinned Down Builds
Appendix F. Git Layout for Computerate Specifications
Acknowledgements
Contributors
Changelog
Author's Address

1. Introduction

If, as the unofficial IETF motto states, we believe that "running code" is an important part of the feedback provided to the standardization process, then as per the Curry-Howard equivalence [Curry-Howard] (that states that code and mathematical proofs are the same), we ought to also believe that "verified proof" is an equally important part of that feedback. A verified proof is a mathematical proof of a logical proposition that was mechanically verified by a computer, as opposed to just peer-reviewed.

The "Experiences with Protocol Description" paper from Pamela Zave [Zave11] gives three conclusions about the usage of formal specifications for a protocol standard. The first conclusion states that informal methods (i.e. the absence of verified proofs) are inadequate for widely used protocols. This document is based on the assumption that this conclusion is correct, so its validity will not be discussed further.

The second conclusion states that formal specifications are useful even if they fall short of the "gold standard" of a complete formal specification. We will show that a formal specification can be incrementally added to a document.

The third conclusion from Zave's paper states that the normative English language should be paraphrasing the formal specification. The difficulty here is that to be able to keep the formal specification and the normative language synchronized at all times, these two should be kept as physically close as possible to each other.

To do that we introduce the concept of "Computerate Specifying" (note that Computerate is a British English word). "Computerate Specifying" is a play on "Literate Computing", itself a play on "Structured Computing" (see [Knuth92] page 99). In the same way that Literate Programming enriches code by interspersing it with its own documentation, Computerate Specifying enriches a standard specification by interspersing it with code (or with proofs, as they are the same thing), making it a computerate specification.

Note that computerate specifying is not specific to the IETF, just like literate computing is not restricted to the combination of TeX and Pascal described in Knuth's paper. What this document describes is a specific instance of computerate specifying that combines [AsciiDoc] as formatting language and [Idris2] as programming language with the goal of formally specifying IETF protocols.

2. Overview

Nowadays documents at the IETF are written in a format named xml2rfc v3 [RFC7991] but unfortunately making that format computerable is not trivial, mostly because there is no simple solution to mix code and XML together in the same file. Instead the [AsciiDoc] format was

selected as a layer on top of xml2rfc v3. The AsciiDoc format has some superficial commonalities with the Markdown formats, but provides extensibility within a unified syntax.

This is the extensibility of AsciiDoc that enables mixing code with text in a document, making it a candidate for literate programming [Knuth92]. But the most important feature added to AsciiDoc is the ability to insert in the document the textual result of the evaluation of the code in that same file.

The AsciiRFC [I-D.ribose-asciirfc] document states additional reasons why AsciiDoc is a superior format for the purpose of writing standards, so that will not be discussed further.

The AsciiDoc to xml2rfc v3 converter is itself an extension that is provided by the [Metanorma] project. Both AsciiRFC and the online documentation [Metanorma-IETF] for that extension are prerequisites to write AsciiDoc document that can be converted into an Internet-Draft or an RFC.

The code in a computerate specification uses the programming language [Idris2] in literate programming [Knuth92] mode. The Idris2 programming language has been chosen because its type system supports dependent and linear types [Brady17], and that type system is the language in which propositions are written. The Idris2 programming also has reflection capabilities and support for meta-programming, also known as elaboration, which are useful for generating code.

Following Zave's second conclusion, computerate specifying is not restricted to the specification of protocols, or to property proving. There is a whole spectrum of formalism that can be introduced in a specification, and we will present it in the remaining sections by increasing order of complexity. Note that because the specification language is a programming language, these usages are not exhaustive, and plenty of other usages can and will be found after the publication of this document.

WARNING
The IETF Trust licences [TLP5] do not grant permission to distribute a retrofitted computerate specification for an Internet-Draft or RFC as a whole so redistributing such specification would be a copyright license infringement. Appendix F shows how to use transclusions to work around that issue.

The installation and use of the tooling are explained in Appendix A.

The remaining of this document is divided in 3 parts:

After the Terminology (Section 3) section starts a tutorial on how to write a computerate specification. This tutorial is meant to be read in sequence, as concepts defined in early part will not be repeated later. On the other hand the tutorial is designed to

present information progressively and mostly in order of complexity, so it is possible to start writing effective specifications without reading or understanding the whole tutorial.

This tutorial begins by explaining how to write basic specifications (Section 4), which are specifications that use code to generate examples and similar parts of a document.

Then the tutorial continues by explaining how to write verified specifications (Section 5), which are specifications that contains generated examples and similar parts that are correct by construction.

Writing verified specifications is difficult and time-consuming, so the tutorial continues by explaining how to use the modules defined in the computerate specification standard library (Section 6). These modules are useful to build examples and parts that are most frequently used in IETF documents.

The tutorial ends with explanations on how to design a specification package (Section 7), which provides both code and documentation that can be used by other computerate specifications.

The second part of this document contains the description of all the packages and modules in the computerate specifying standard library (Section 8).

The third part contains various appendices:

Appendix A explains how to install and use the associated tooling, Appendix B contains the reference manual for the standard library, Appendix D explains how to convert Colored Petri Nets in a form that can be used in specifications, Section 5.1 is a tutorial on using Programs and Types to prove propositions, Appendix E explains the distributed architecture of the standard library, and Appendix F describes the format of the files distributed in the standard library.

3. Terminology

Computerate Specification, Specification: Literate Idris2 code embedded in an AsciiDoc document, containing both formal descriptions and human language texts, and which can be processed to produce documents in human language.

Document: Any text that contains the documentation of a protocol in the English language. A document is the result of processing a specification.

Retrofitted Specification: A specification created after a document was published such as the generated document coincides with the published document.

In this document, the same word can be used either as an English word or as an Idris identifier used inside the text. To explicitly differentiate them, the latter is always displayed like "this". E.g. "IdrisDoc" is meant to convey the fact that IdrisDoc in that case is an Idris module or type. On the other hand the word IdrisDoc refers to the IdrisDoc specification.

By convention an Idris function that returns a type and types themselves will always start with an uppercase letter. Functions not returning a type start with a lowercase letter.

For the standard library, the types names are also formed by taking the English word or expression, making the first letter of each word upper case, and removing any symbols like underscore, dash and space. Thus bitvector would become "Bitvector" after conversion as a type name but bit diagram would become "BitDiagram".

"Metanorma" is a trademark of Ribose Inc.

4. Basic Specification Tutorial

Creating a new computerate specification always start by creating a new AsciiDoc document, and there are different ways to do that. Writing it from scratch is difficult and error-prone, so a better alternative is to start by copying a template (Appendix A.2.1) provided as part of the tooling. Reusing a previous specification is not recommended as new features present in an updated version of the template may be missed. The last alternative is to convert (Appendix A.2.2) an existing RFC or Internet-Draft into an AsciiDoc document, which is the path to take to produce a retrofitted specification.

The next step is to start adding code to that document to make it a computerate specification. This generally requires to split the AsciiDoc document in multiple documents, mostly because different file extensions indicate how a file will be processed by the tooling:

* There should be a unique file with the extension ".lipkg" (lipkg file) that acts as the root document and as the Idris package description. It is recommended that the name of this file being either the name of the RFC or the part of the Internet-Draft name that will not change as it proceeds through the publication process. E.g. the name of the lipkg file for the computerate specification underneath this document is "computerate-specifying.lipkg". Each line of the package description in that file is must start with a Bird style mark ("> ") on the first column. A block of package description lines must be preceded by the beginning of the file or an empty line and followed by either the end of the file or an empty line.

* Any part of the specification that contains code needs to be in a separate file with the extension ".lidr" (lidr file). Code in lidr files uses the same rules than for the package description.

*It can be convenient to also have separate files for the parts of the specification that do not require code. These files must have the ".adoc" extension (adoc file).

These files are put back together by using, directly or indirectly, AsciiDoc "include::" statements in the lipkg file.

WARNING
The Bird style mark is also used by AsciiDoc as an alternate way of defining a blockquote [Blockquotes] which is consequently not available in a computerate specification.

The code will not appear in the rendered document, but self-inclusion can be used to copy part of the code into the document, thus guaranteeing that this code is verified by the type-checker:

```
> -- tag::proof[]
> total currying : ((a, b) -> c) -> (a -> b -> c)
> currying f = \x => \y => f (x, y)
> -- end::proof[]

....
 include::myfile.lidr[tag=proof]
....
```

Code can be evaluated and the resulting text inserted in the document by using one of the code macros.

The "code:[]" inline macro is used when the result is to be inserted as AsciiDoc text. To do so, the type of that result must implement the "Show" interface or the "Asciidoc " interface in the "ComputerateSpecifying.Metanorma.Ietf" module so the generated text is correctly escaped. For instance the following excerpt taken from the computerate specification of [RFC8489]:

```
> retrans' : Nat -> Int -> Maybe (List1 Int)
> retrans' rc = fromList . take rc . scanl (+) 0
>  . unfoldr (bimap id (*2) . dup)

> retrans : Nat -> Int -> String
> retrans rc = maybe "Error"
>   (foldr1By (\e, a => show e ++ " ms, " ++ a)
>     (\x => "and " ++ show x ++ "ms")) . retrans' rc

> timeout : Nat -> Int -> Int -> String
> timeout rc rto rm = maybe "Error"
>   (\e => show (last e + rm * rto)) (retrans' rc rto)

> rc : Nat; rc = 7
> rto : Int; rto = 500
> rm : Int; rm = 16

For example, assuming an RTO of code:[rto]ms, requests would be
sent at times code:[retrans rc rto].
If the client has not received a response after code:[timeout] ms,
the client will consider the transaction to have timed out.
```

is rendered as

"For example, assuming an RTO of 500ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500ms. If the client has not received a response after 39500 ms, the client will consider the transaction to have timed out."

Additionally using the "code:[rto]", "code:[rc]", and "code:[rm]" macros in lieu of the equivalent constants in the document ensures that the text stays consistent during the development of the specification.

The "code::[]" block macro (notice the double colon) is used when the result is to be pretty printed before being inserted as an AsciiDoc block. To do so, the type of that result must implement the "Pretty" interface in the "Text.PrettyPrint.Prettyprinter" module so the generated text fits in the available number of columns in the resulting document, even if the AsciiDoc block is indented.

A number of attributes can be added in inline and block code macros:

impl: Select a named implementation for the "Asciidoc", or "Pretty" interface.

eval: Select a different evaluation process for the code, generally because the default process is slow. Possible values are "exec" or "scheme".

Some attributes are specific to code block macros:

style:
: the style to use for the block.

WARNING
Code containing a comma in a macro must be enclosed in double-quotes.

Code macros are processed separately from the code itself which means that, at the difference of the code, macros can reference code that is defined further down in the document. Code macros can even reference code that is defined in a separate lidr or idr files, as long as the name of the lidr or idr file is inserted between the last colon and the opening square bracket. This allows the use of code macros in adoc or lipkg files

For instance the "code:basic.lidr[rto]" macro can be used in any file to insert the RTO value defined in the "basic.lidr" file.

Alternating paragraphs of text and code permits to keep both representations as close as possible and is an effective way to quickly discover that the code and the text are diverging. The convention is to insert the code beneath the text it is related to.

The code itself imposes an order in which it must be declared and used because it does not by default look for forward references. Because in that case the text will follow the order the code is organized, the document generated tends to be naturally easier to implement because it favors a workflow that parallels the software implementation of the documented protocol. [RFC8489] and [RFC8656] are examples of standards that were made easier to implement because they follow the order a software developer follows to develop each component.

On the opposite documents that are not generated from a specification do not always have a structure that follow the way a software developer will implement them. When that is the case it will be difficult to add the Idris code right after a paragraph describing its functionality, as the final code may not type-check in the presence of unsupported forward references.

It could be a hint that the text needs to be reorganized to be more software-development friendly, but for retrofitted specifications an alternative is to combine self-inclusion and conditions to change the order in which paragraphs will be rendered. A paragraph has to be enclosed in an "ifdef::" statement that use an attribute that is never set ("never"), then in a "tag::" statement on a comment line:

```
ifdef::never[]
// tag::para1[]

Text that describes a functionality

// end::para1[]
endif::never[]

> -- Code that implements that functionality.
```

Then the paragraph can be moved at the right place:

```
include::main.lidr[tag=para1]
```

5. Verified Specification Tutorial

WARNING
This whole section will be rewritten.

A verified specification is a specification where the generated examples and other parts are not just the result of evaluating code, but.

A specification uses Idris types to specify both how stream of bits are arranged to form valid Protocol Data Units (PDU) and how the exchange of PDUs between network elements is structured to form a valid protocol. In addition a specification can be used to prove or disprove a variety of properties for these types.

5.1. Evidence-Based Answers

This document uses a special interpretation of Programs and Types that permits to build evidence-based answers to the kind of questions that a network protocol designer would be asking of its designs.

Although that interpretation is not new, few textbooks are available to concretely learn it and even when available, these textbooks generally take the long road by choosing to teach first Constructive Logic and then apply these teaching to Programs and Types. As there is in fact an even longer road that would take from Fibred Category Theory to Constructive Logic and then to Programs and Types, it is reasonable to think that there should be a shortcut there that would permit to start directly with Programs and Types, especially when the target audience is programmers, a segment of the technical population that is known to dislike mathematics.

Still, the mathematically inclined or the non-programmer can look at [Nederpelt14], [Bornat05], or [Mimram20] for an approach based on mathematics.

Basically the goals of that interpretation of Program and Types are:

* To answer any question with either "Yes", "No", or "Don't know".
* To ensure that any "Yes" or "No" answer is provided with evidence.
* To use programming to achieve these goals.

The kind of questions that a network protocol designer may want to get that kind of evidence-based questions for are many:

* Is my new version of the protocol safe to interoperate with the previous versions?
* Is the protocol free of deadlock?
* Is there corner cases that I neglected to take in account.
* Is that faster but more efficient protocol equivalent to the slower but simpler original protocol?

Notice that when we talk about evidence-based answers, we exclude by definition any answer that has a probability different of 0.0 or 1.0, and furthermore exclude evidence-free answers like the ones given by AI/ML.

As a consequence, we have to admit that there are questions that do not have an evidence-based answer. That could be for a short list of reasons:

* We did not looked in the literature yet if there is an existing answer to a particular question.
* Nobody yet tried to find an answer to that question.
* Nobody found an answer to the question yet.
* There is no answer to that question.

There is clearly a question of locality of our knowledge at play here, and we are not pretending to get to some absolute truth with this technique.

5.1.1. Encoding Questions

In the 90s came this new idea that it was possible to use the C++ type system to encode calculations. A famous example was generating all the prime numbers during the compilation of a C++ program. The result was provided as a result of compiling the program, and the compiled program itself was irrelevant to get that result. This was done by reinterpreting the type system into a computational system.

Here we are going to do the same thing, and reinterpret the type system of a programming language, Idris, as a way to encode our questions.

As we will see, to be able to do this reinterpretation, the type system needs to be stronger than in a traditional programming language so to be able to encode a large variety of questions. We will also see that, paradoxically, the computational power of our programming language needs to be reduced to be sure that the evidence of our answers is valid.

One defining feature of that programming language is that the compilation step that in traditional programming languages is monolithic, is here split in 2 separate steps:

* The typechecking step takes a set of source files and verifies that all values in these sources (including the code as a value of the function type) can be assigned to the correct type. Because of the complexity of the type system, an Idris interpreter is used to evaluate expressions during the typechecking step.

* The code generation step generates executable code.

As our interpretation relies only on what happens in the typechecking step, we have no use for the second step of the compilation process.

5.1.1.1. Any Value of a Type is Evidence of Yes

The cornerstone of our new interpretation is that the evidence that the answer of a question is Yes is an value of the type that encodes that question. We will see later that the evidence that the answer is No is the inability to produce a value of a type.

Although there is no real usage for these, if we interpret the basic types in our programming languages as questions, then the answers to these are always Yes, because we can always find a value for these types:

```
1 : Int

"s" : String
```

In Idris a value of a type is written first, then followed by a colon and by the type of that value.

Note that it does not matter if you can find one or two millions different pieces of evidence - the answer is still Yes. The exact value we pick as evidence is absolutely irrelevant, which is something that may seems strange to a programmer.

This is why basic types are not really interesting in our interpretation, as their answer is always Yes.

5.1.1.2. Function Type As Implication

Idris is a pure functional programming language, so functions are first class citizens of the language, and their type is called a Function Type.

The interpretation of a Function Type is that of an implication. Implications are a form of "if P then Q" statement, that says something about the relationship between two other Types, here P and Q.

In Idris the Function Type is represented as an arrow that separate the first type (sometimes called the domain of the function) from the second type (sometimes called the codomain of the function).

```
P -> Q
```

To answer the question "P -> Q" we need to find a value of that type. An value of a Function Type is a program, so a program that takes values of P as parameter and returns a value of Q is an evidence that the answer to "P -> Q" is Yes. Another equivalent reading would be "Assuming that we can provide values for P and values for Q, then can we provide a function that typechecks?"

Notice again that there maybe many programs that fulfill that condition but again that is irrelevant, as we need only one to serve as evidence.

We can easily produce an evidence of that, let's say using Int and String as our types:

```
\x => "a" : Int -> String
```

The expression on the left of the colon is a lambda expression. "x" will be bound to whatever value of Int will be passed as parameter, and the function will return True.

Note that this works only because Idris is a pure functional language, meaning that a function can only use the values passed as parameters in its evaluation of the returned value. Side effects or global variables are not available in a pure functional language.

A function in Idris can only take one parameter, but it is possible to return a function, which permits to simulate a multi-parameter function (this is known as currying):

```
\x => \y => True : Int -> (String -> Bool)
```

Function types associate to the right, so the parenthesis in the example above is not really necessary.

Functions in Idris can also take a function as parameter, which will permit to encode the classical question:

"Socrates is a human, all humans are mortals, is Socrates a mortal?"

We can encode this in the Idris type system:

```
data Human : Type

data Mortal : Type

isSocratesMortal : Human -> (Human -> Mortal) -> Mortal
```

Notice that here the parenthesis are mandatory. The question can be read like this:

"Assuming Socrates is a Human, and assuming that all Humans are Mortals, then is Socrates a Mortal?"

The evidence is easy to find:

```
isSocratesMortal : Human -> (Human -> Mortal) -> Mortal
isSocratesMortal = \h => \f => f h
```

One important point is that we are not trying to say that Socrates is a Human (maybe he was an alien). Similarly we are not trying to say that there is an absolute rule that all humans are mortals (in fact there is evidence that, at the time of writing, the human author of this document was immortal).

What we are saying is that assuming that we have evidence of a human (Socrates in that case) and assuming that we have evidence that all humans are mortals, then the only conclusion is that, Yes, Socrates is mortal, and the evidence for this is the program "\h => \f => f h".

Note that in a function definition, the parameters can be moved to the left hand side (LHS) of the equal sign, like this:

```
isSocratesMortal : Human -> (Human -> Mortal) -> Mortal
isSocratesMortal h f = f h
```

5.1.1.3. Polymorphism

In some cases, questions can be made more general and still have a unique answer. This is the case for the question explored in the previous section, where the question can be generalized to something called syllogism (also known as *Modus Ponens*).

Polymorphism permits to substitute a type with a value that represents any type. Thus finding an evidence shows that the answer is Yes for a whole family of related questions.

Here we express that new generic question (the answer is the same) as this:

```
syllogism : p -> (p -> q) -> q
syllogism x f = f x
```

An identifier that starts with a lowercase character in an Idris type stands for all possible types.

Here we have evidence that a question with this particular shape can always be answered with Yes.

5.1.1.4. Empty Type as No

We saw previously that any value of a type is evidence of the Yes answer to the question encoded in that type. So the absence of a value for a type is evidence that the answer is No.

We have a problem here, as the evidence of No is that we cannot provide an evidence. But, from our local point of view, there is no difference between the fact that there is no evidence, and the fact that we did not searched hard enough for the evidence.

We can work around this by using a property of implication, which is that only a type with a No answer can imply a type with a No answer. So if we can implement a function (the Yes answer to the implication) between a type and an empty type (i.e., a type with a No answer), then we know that the former type is empty and that the answer it represents is also No.

Idris provides an empty type for that: Void (not to be confused with the Java type Void, which is not an empty type).

```
noEvidence: Int -> Void
noEvidence x = ?aa
```

Here we cannot complete the program because we cannot produce a value of type Void, and that's because Int has Yes as answer.

In Idris names that starts with a question mark are called holes and stand for a part of the program that we cannot or did not yet complete.

```
data Empty : Type where

emptyIsNo : Empty -> Void
emptyIsNo x impossible
```

Here we can write a program that shows that that "Empty" is equivalent to "Void", this program acting as evidence that there is no evidence for "Empty", and so that the answer is No.

Programmers will again be intrigued that a program that typechecks cannot be executed or tested.

The possibility of defining a type like Void that does not have any values by definition is one of the reason we need a different type system that used in most programming languages (most programming languages permits the use of "null" as value for any type).

We also touched on the fact that our programming language must be less powerful than usual, and it is also related to the answer No.

An implication to a type that contains at least one value is a function that returns that value. But there is two cases where that function could not return that value, and thus acts as if the returned type is empty, and thus represents No instead of Yes.

The first case is if the function crashes because it does not know how to handle the value passed as parameter. A simple example example would be a function that divide 1 by the parameter - if the parameter is zero then the function will crashes and for the purpose of our interpretation is equivalent to an evidence of No because no value will be returned. To prevent that problem our programming language should be covering all inputs values, i.e. not typecheck if there is cases not covered.

The second case is when, for some reason, the code get stuck inside the function e.g., because of an infinite loop. That would again be equivalent to an evidence of No.

Idris prevents these two cases by using the "total" keyword, which basically turns Idris into a non-Turing Complete language.

Note that there is no way to possibility to write code that will detect for any possible code if it will loop or not. That's why

Idris may reject some code that will not loop, but it will never accept code that will loop.

5.1.1.5. Sloppy Questions

Because there is not much difference between a No answer without evidence and not finding an answer, it's often useful to check and recheck that the question really expresses what we intended.

In the previous section we showed that syllogisms always have an answer of Yes. There is a series of fallacies [Bennett15] that are closely related to syllogisms, and here's one of them:

```
syllogism : p -> (q -> p) -> q
```

That can be read as "Assuming Socrates is a Human, and assuming that all Mortal are Humans, then is Socrates a Mortal?" It may seems obvious that we cannot answer that question, so we may be able to get a No answer by rewriting the question that way:

```
syllogistic_fallacy : (p -> (q -> p) -> q) -> Void
```

But in spite of our efforts, we cannot provide an evidence of that, which means that it is time to look closer at our question.

The issue is that for this to be a fallacy, we need to assume that there is no evidence that all Humans are Mortals, which the previous question does not say. With this modified question, we can now produce a evidence that it is indeed a fallacy:

```
syllogistic_fallacy : ((p -> q) -> Void) ->
                        (p -> (q -> p) -> q) -> Void
syllogistic_fallacy f g = f (\x => g x (\y => x))
```

5.1.1.6. Product Type

The Product type permits to combine two or more questions such as the question represented by this type will have an answer of Yes only if all the questions also have an answer of Yes.

The simplest Product type in Idris is the tuple, which is represented as a list of types separated by commas and enclosed in parentheses:

```
product : (String, Int, Char) -> Bool
product : (x, y, z) -> True
```

The evidence has the same form as the type.

We can also provide evidence that the form above is equivalent to its curried form in general, and vice-versa:

```
curry : ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)

uncurry : (a -> b -> c) -> ((a, b) -> c)
uncurry f x = f (fst x) (snd x)
```

5.1.1.7. Sum Type

The Sum type is a way to combine two or more questions such as the question represented by the Sum type will have an answer of Yes if at least one of the questions have an answer of Yes.

The simplest Sum type for two questions in Idris is "Either a b".

```
sum : Either String Void -> Bool
sum x = True
```

We can combine Sum and Product types to reorganize a question and show evidence that the answer is general.

```
dist : (a, Either b c) -> (Either a b, Either a c)
dist x = (Left (fst x), Left (fst x))
```

Sum and Product combined with negation gives us more general answers:

```
dm1 : (Either (a -> Void) (b -> Void)) -> ((a, b) -> Void)
dm1 (Left x) y = x (fst y)
dm1 (Right x) y = x (snd y)

dm2 : (a -> Void, b -> Void) -> ((Either a b) -> Void)
dm2 x (Left y) = fst x y
dm2 x (Right y) = snd x y

dm3 : ((Either a b) -> Void) -> (a -> Void, b -> Void)
dm3 f = (\x => f (Left x), \x => f (Right x))
```

5.1.1.8. Inductive Type

TBD.

5.1.1.9. Pi Type

TBD.

5.1.1.10. Sigma Type

TBD.

5.1.1.11. Equality Type

TBD.

5.1.1.12. Decidable Type

TBD.

5.1.2. How to Find Evidence

TBD

5.2. PDU Descriptions

The PDUs in a communication protocol determines how data is laid out before it is sent over a communication link. Generally a PDU is described only in the context of the layer that this particular protocol is operating at, e.g. an application protocol PDU only describes the data as sent over UDP or TCP, not over Ethernet or Wi-Fi.

PDUs can generally be split into two broad categories, binary and text, and a protocol PDU mostly falls into one of these two categories.

PDU descriptions can be defined as specifications for at least three reasons: the generation of examples that are correct by construction, correctness in displaying the result of calculations, and correctness in representing the structure of a PDU.

Independently of these reasons, a PDU description is a basic component of a specification that will probably be needed regardless.

5.2.1. PDU Examples

Examples in protocol documents are frequently incorrect, which proves to have a significant negative impact as they are too often misused as normative text. See Appendix C for statistics about the frequency of incorrect examples in RFC errata.

Ensuring example correctness is achieved by adding the result of a computation (the example) directly inside the document. If that computation is done from a type that is (physically and conceptually) close to the normative text, then we gain some level of assurance that both the normative text and the derived examples will match.

Generating an example that is correct by construction always starts by defining a type that describes the format of the data to display. The Internet Header Format in section 3.1 of [RFC0791] will be used in the following sections as example.

In this section we start by defining an Idris type, using a Generalized Algebraic Data Type (GADT). In that case we have only one constructor ("MkInternetHeader") which is defined as a Product Type that "concatenate" all the fields on the Internet Header. One specific aspect of Idris types is that we can enrich the definition of each field with constraints that then have to be fulfilled when a value of that type will be built.

```
> data InternetHeader : Type where
>   MkInternetHeader :
>     (version : Int) -> version = 4 =>
>     (ihl : Int) -> ihl >= 5 && ihl < 16 = True =>
>     (tos : Int) -> tos >= 0 && tos <= 256 = True =>
>     (length : Int) -> length >= (5 * 4) &&
>       length < 65536 = True =>
>     (id : Int) -> id >= 0 && id < 65536 = True =>
>     (flags : Int) -> flags >= 0 && flags < 16 = True =>
>     (offset : Int) -> offset >= 0 && offset < 8192 = True =>
>     (ttl : Int) -> ttl >= 0 && ttl < 256 = True =>
>     (protocol : Int) -> protocol >= 0 &&
>       protocol < 256 = True =>
>     InternetHeader
```

where

version:
This field is constrained to always contain the value 4.

ihl: "Int" is a builtin signed integer so it is constrained to contain only positive integers lower than 16.

others: Same, all the fields are constrained to unsigned integers fitting inside the number of bits defined in [RFC0791].

An Idris type where the fields in a constructor are organized like the "InternetHeader" by ordering them in a sequence is called a Pi type - or, when there is no dependencies between fields as there is in "version = 4", a Product type. Although there is no equivalence in most programming languages to a Pi type, Product types are known as classes in Java and struct in C.

Another way to organize a type is called the Sum type, which is a type with multiple constructors that act as alternative. Sum types can be used in C with a combination of struct and union, and since Java 14 by using sealed records.

Sum types have a dependent counterpart named a Sigma type, which is a tuple in which the type of the second element depends on the value of the first element. This is mostly returned by functions, with the returned Sigma type carrying both a value and a proof of the validity of that value.

From that point it is possible to define a value that fulfills all the constraints. The following values are taken from example 1 in [RFC0791] Appendix A.

```
> example1 : InternetHeader
> example1 = MkInternetHeader 4 5 0 21 111 0 0 123 1
```

The "=>" symbol after a constraint indicates that Idris should try to automatically find a proof that this constraint is met by the values in the example, which it successfully does in the example above.

The following example, where the constraints defined in the InternetHeader type are not met, will not type-check in Idris (an error message will be generated) and thus can not be used to generate an example.

```
example1' : InternetHeader
example1' = MkInternetHeader 6 5 0 21 111 0 0 123 1
```

The next step is to define an Idris function that converts a value of the type "InternetHeader" into the kind of bit diagram that is showed in Appendix A of [RFC0791].

```
> Show InternetHeader where
>   show (MkInternetHeader version ihl tos length id flags offset
>     ttl protocol) = ?showPrec_rhs_1
```

Here we implement the "Show" interface that permits to define the adhoc polymorphic function "show" for "InternetHeader", function that will convert the value into the right character string. Idris names starting with a question mark like in "?showPrec_rhs_1" are so-called holes, which are placeholder for code to be written, while still permitting type-checking.

After replacing the hole by the actual code, the following embedded code can be used in the document to generate an example that is correct by construction, at least up to mistakes in the specification (i.e. the constraints in "InternetHeader") and bugs in the "show" function.

```
....
code::[example1]
....
```

will generate the equivalent AsciiDoc text:

```
....
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver= 4 |IHL= 5 |Type of Service|        Total Length = 21      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Identification = 111     |Flg=0|   Fragment Offset = 0   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Time = 123  |  Protocol = 1 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
....
```

This generated example is similar to the first of the examples in appendix A of RFC 791.

5.2.2. Calculations from PDU

The previous section showed how to define a type that precisely describes a PDU, how to generates examples that are are values of that type, and how to insert them in a document.

Our specification, which has the form of an Idris type, can be seen as a generalization of all the possible examples for that type. Now that we went through the effort of precisely defining that type, it would be useful to use it to also calculate statements about that syntax.

In RFC 791 the description of the field IHL states "[...]that the minimum value for a correct header is 5." The origin of this number may be a little mysterious, so it is better to use a formula to calculate it and insert the result instead.

Inserting a calculation is easy:

```
Note that the minimum value for a correct header is
is code:[sizeHeader `div` ihlUnit].

> sizeHeader : Int
> sizeHeader = 20

> ihlUnit : Int
> ihlUnit = 4
```

Here we can insert a code fragment that is using a function that is defined later in the document because the Idris code is evaluated before the document is processed.

Note the difference with examples: The number "5" is not an example of value of the type "InternetHeader", but a property of that type.

Systematically using the result of calculation on types in a specification makes it more resistant to mistakes that are introduced as result of modifications.

5.2.3. PDU Representations

The layout of a PDU, i.e. the size and order of the fields that compose it can be represented in a document in various forms. One of them is just an enumeration of these fields in order, each field identified by a name and accompanied by some description of that field in the form of the number of bits it occupies in the PDU and how to interpret these bits.

That layout can also be presented as text, as a list, as a table, as a bit diagram, at the convenience of the document author. In all

cases, some parts of the description of each field can be extracted from our Idris type just like we did in Section 5.2.2.

RFC 791 section 3.1 represents the PDUs defined in it both as bit diagrams and as lists of fields.

5.3. State Machines

A network protocol, which is how the various PDUs defined in a document are exchanged between network elements, can always be understood as a set of state machines. At the difference of PDUs, that are generally described in a way that is close to their Idris counterpart, state machines in a document are generally only described as text.

Note that, just like an Idris representation of a PDU should also contain all the possible constraints on that PDU but not more, a state machine should contain all the possible constraints in the exchange of PDUs, but not less.

This issue is most visible in one of the two state machines defined in RFC 791, the one for fragmenting IP packets (the other is for unfragmenting packets). The text describes two different algorithms to fragment a packet but in that case each algorithm should be understood as one instance of a more general state machine. That state machine describes all the possible sequences of fragments that can be generated by an algorithm that is compliant with RFC 791 and it would be an Idris type that is equivalent to the following algorithm:

- For a specific packet size, generate a list of all the binary values {b0,.., bN} with N being the packet size divided by 8 and rounded-up, and 0..N representing positional indexes for each of the 8 byte chunks of the packet.

- For each binary value in that list, generate a list of values that represents the number of consecutive bits of the same value (e.g.. "0x110001011" generates a "[2, 3, 1, 1, 2]" list), each such sequence representing a given fragment.

- Remove from that list of lists any list that contains a number that, after multiplication by 8, is higher than the maximum size of a fragment.

- For each remaining list in that list, generate the list of fragments, i.e with the correct offset, length and More bit.

- Generate all the possible permutations for each list of fragments.

We can see that this state machine takes in account the fact that an IP packet can not only be fragmented in fragments of various sizes - as long as the constraints are respected - but also that these fragments can be sent in any order.

Then the algorithms described in the document can be seen as generating a subset of all the possible list of fragments that can be generated by our state machine. It is then easy to check that these algorithms cannot generate fragments lists that cannot be generated by our state machine.

As a consequence, the unfragment state machine must be able to regenerate a valid unfragmented packet for any of the fragments list generated by our fragment state machine. Furthermore, the unfragment state machine must also take in account fragment lists that are modified by the network (itself defined as a state machine) in the following ways:

* fragments can be dropped;

* the fragments order can change (this is already covered by the fact that our fragment state machine generates all possible orders);

* fragments can be duplicated multiple times;

* fragments can be delayed;

* fragments can be received that were never sent by the fragment state machine.

Then the algorithm described in the document can be compared with the unfragment state machine to verify that all states and transitions are covered.

Defining a state machine in Idris can be done in an ad-hoc way [Linear-Resources], particularly by using linear types that express resources' consumption.

5.4. Proofs

Under the Curry-Howard equivalence, the Idris types that we created to describe PDUs and state machine are formal logic propositions, and being able to construct values from these types (like we did for the examples), is proof that these propositions are true. These are also called internal verifications [Stump16].

External verifications are made of additional propositions (as Idris types) and proofs (as code for these types) with the goal of verifying additional properties.

One kind of proofs that one would want in a specification are related to isomorphism, i.e. a guarantee that two or more descriptions of a PDU or a state machine contain exactly the same information, but there is others.

5.4.1. Wire Type vs Semantic Type

The Idris types that are used for generating examples, calculations or representations are generally very close to the bit structure of the PDU. But some properties may be better expressed by defining types that are more abstract. We call the former Wire Types, and the latter Semantic Types.

As example, the type in Section 5.2.1 is a wire type, because it follows exactly the PDU layout. But fragmentation can be more easily described using the following semantic type:

```
> data InternetHeader' : Type where
>   Full : (ihl : Int) -> ihl >= 5 && ihl < 16 = True =>
>          (tos : Int) -> tos >= 0 && tos <= 256 = True =>
>          (length : Int) -> length >= (5 * 4) &&
>            length < 65536 = True =>
>          (ttl : Int) -> ttl >= 0 && ttl < 256 = True =>
>          (protocol : Int) -> protocol >= 0 &&
>            protocol < 256 = True =>
>          InternetHeader'
>   First : (ihl : Int) -> ihl >= 5 && ihl < 16 = True =>
>           (tos : Int) -> tos >= 0 && tos <= 256 = True =>
>           (length : Int) -> length >= (5 * 4) &&
>             length < 65536 = True =>
>           (id : Int) -> id >= 0 && id < 65536 = True =>
>           (ttl : Int) -> ttl >= 0 && ttl < 256 = True =>
>           (protocol : Int) -> protocol >= 0 &&
>             protocol < 256 = True =>
>           InternetHeader'
>   Next : (ihl : Int) -> ihl >= 5 && ihl < 16 = True =>
>          (tos : Int) -> tos >= 0 && tos <= 256 = True =>
>          (length : Int) -> length >= (5 * 4) &&
>            length < 65536 = True =>
>          (offset : Int) -> length > 0 &&
>            length < 8192 = True =>
>          (id : Int) -> id >= 0 && id < 65536 = True =>
>          (ttl : Int) -> ttl >= 0 && ttl < 256 = True =>
>          (protocol : Int) -> protocol >= 0 &&
>            protocol < 256 = True =>
>          InternetHeader'
>   Last : (ihl : Int) -> ihl >= 5 && ihl < 16 = True =>
>          (tos : Int) -> tos >= 0 && tos <= 256 = True =>
>          (length : Int) -> length >= (5 * 4) &&
>            length < 65536 = True =>
>          (offset : Int) -> length > 0 &&
>            length < 8192 = True =>
>          (id : Int) -> id >= 0 && id < 65536 = True =>
>          (ttl : Int) -> ttl >= 0 && ttl < 256 = True =>
>          (protocol : Int) -> protocol >= 0 &&
>            protocol < 256 = True =>
>          InternetHeader'
```

First the "version" field is eliminated, because it always contains the same constant.

Then the "flags" and "offset" fields are reorganized so to provide four different alternate packets:

* The "Full" constructor represents an unfragmented packet. It is isomorphic to a "MkInternetHeader" with a "flags" and "offset" values of 0.

* The "First" constructor represents the first fragment of a packet. It is isomorphic to a "MkInternetHeader" with a "flags" value of 1 and "offset" value of 0.

* The "Next" constructor represents a intermediate fragments of a packet. It is isomorphic to a "MkInternetHeader" with a "flags" value of 1 and "offset" value different than 0.

* Finally the "Last" constructor represents the last fragment of a packet. It is isomorphic to a "MkInternetHeader" with a "flags" value of 0 and "offset" value different than 0.

One of the main issue of having two types for the same data is ensuring that they both contains the same information, i.e. that they are isomorphic. To ensure that these two types are carrying the same information we need to define and implement four functions that, all together, prove that the types are isomorphic. This is done by defining the 4 types below, as propositions to be proven:

```
> total
> to : InternetHeader -> InternetHeader'
>
> total
> from : InternetHeader' -> InternetHeader
>
> total
> toFrom : (x : InternetHeader') -> to (from x) = x
>
> total
> fromTo : (x : InternetHeader) -> from (to x) = x
```

Successfully implementing these functions will prove that the two types are isomorphic. Note the usage of the "total" keyword to ensure that these are proofs and not mere programs.

5.4.2. Data Format Conversion

For documents that describe a conversion between different data layouts, having a proof that guarantees that no information is lost

in the process can be beneficial. For instance, we observe that syntax encoding tends to be replaced each ten years or so by something "better". Here again isomorphism can tell us exactly what kind of information we lost and gained during that replacement.

Here, for example, the definition of a function that would verify an isomorphism between an XML format and a JSON format:

```
isXmlAndJsonSame: Iso (XML, DeltaXML) (JSON, DeltaJson)
  ...
```

DeltaXML expresses what is gained by switching from XML to JSON, and DeltaJson expresses what is lost.

5.4.3. Postel's Law

> Be conservative in what you do, be liberal in what you accept from others.
> — *Jon Postel - RFC 761*

One of the downsides of having specifications is that there is no wiggle room possible when implementing them. An implementation either conforms to the specification or does not.

One analogy would be specifying a pair of gears. If one decides to have both of them made with tolerances that are too small, then it is very likely that they will not be able to move when put together. A bit of slack is needed to get the gear smoothly working together but more importantly the cost of making these gears is directly proportional to their tolerance. There is an inflexion point where the cost of an high precision gear outweighs its purpose.

We have a similar issue when implementing a specification, where having an absolutely conform implementation may cost more money than it is worth spending. On the other hand a specification exists for the purpose of interoperability, so we need some guidelines on what to ignore in a specification to make it cost effective.

Postel's law proposes an informal way of defining that wiggle room by actually having two different specifications, one that defines a data layout for the purpose of sending it, and another one that defines a data layout for the purpose of receiving that data layout.

Existing documents express that dichotomy in the form of the usage of SHOULD/SHOULD NOT/RECOMMENDED/NOT RECOMMENDED [RFC2119] keywords. For example the SDP spec says that "[t]he sequence CRLF (0x0d0a) is used to end a line, although parsers SHOULD be tolerant and also accept lines terminated with a single newline character." This directly infers two specifications, one used to define an SDP when sending it, that enforces using only CRLF, and a second

specification, used to define an SDP when receiving it (or parsing it), that accepts both CRLF and LF.

Note that the converse is not necessarily true, i.e. not all usages of these keywords are related to Postel's Law.

To ensure that the differences between the sending specification and the receiving specification do not create interoperability problems, we can use a variant of isomorphism, as shown in the following example (data constructors and code elided):

```
data Sending : Type where

data Receiving : Type where

to : Sending -> List Receiving

from : Receiving -> Sending

toFrom : (y : Receiving) -> Elem y (to (from y))

fromTo : (y : Sending) -> True = all (== y) [from x | x <- to y]
```

Here we define two data types, one that describes the data layout that is permitted to be sent ("Sending") and one that describes the data layout that is permitted to be received ("Receiving"). For each data layout that is possible to send, there is one or more matching receiving data layouts. This is expressed by the function "to" that takes as input one Sending value and returns a list of Receiving values.

Conversely, the "from" function maps a Receiving data layout onto a Sending data layout. Note the asymmetry there, which prevents using a standard proof of isomorphism.

Then the "toFrom" and "fromTo" proofs verify that there is no interoperability issue by guaranteeing that each Receiving value maps to one and only one Sending instance and that this mapping is isomorphic.

All of this will provide a clear guidance of when and where to use a SHOULD keyword or its variants, without loss of interoperability.

As an trivial example, the following proves that accepting LF characters in addition to CRLF characters as end of line markers does not break interoperability:

```
data Sending : Type where
  S_CRLF : Sending

Eq Sending where
  S_CRLF == S_CRLF = True

data Receiving : Type where
  R_CRLF : Receiving
  R_LF : Receiving

to : Sending -> List Receiving
to S_CRLF = [R_CRLF, R_LF]

from : Receiving -> Sending
from R_CRLF = S_CRLF
from R_LF = S_CRLF

toFrom : (y : Receiving) -> Elem y (to (from y))
toFrom R_CRLF = Here
toFrom R_LF = There Here

fromTo : (y : Sending) -> True = all (== y) [from x | x <- to y]
fromTo S_CRLF = Refl
```

Postel's Law is not limited to the interpretation of PDUs as a state machine on the receiving side can also be designed to accept more than what a sending state machine can produce. A similar isomorphism proof can be used to ensure that this is done without loss of interoperability.

5.4.4. Implementability

When applied, the techniques described in Section 5.2 and Section 5.3 result in a set of types that represents the whole protocol. These types can be assembled together, using another set of types, to represent a simulation of that protocol that covers all sending and receiving processes.

The types can then be implemented, and that implementation acts as a proof that this protocol is actually implementable.

To make these pieces of code composable, a specification is split in multiple modules, each one represented as a unique function. The type of each of these functions is derived from the state machines described in Section 5.3, by bundling together all the inputs of the state machine as the input for that function, and bundling all the outputs of the state machine as the output of this function.

For instance the IP layer is really 4 different functions:

* A function that converts between a byte array and a tree representation (parsing).

* A function that takes a tree representation and a maximum MTU and returns a list of tree representations, each one fitting inside the MTU.

* A function that accumulates tree representations of an IP fragment until a tree representation of a full IP packet can be returned.

* A function that convert a tree representation into a byte array.

The description of each function is incomplete, as in addition to the input and the output listed, these functions needs some ancillary data, in the form of:

* state, which is basically values stored between evaluations of a function,

* an optional signal, that can be used as an API request or response. As timers are a fundamental building block for communication protocols, one common uses for that signal are to request the arming of a timer, and to receive the indication of the expiration of that timer.

5.4.5. Termination

Proving that a protocol does not loop is equivalent to proving that a implementation of the types for that protocol does not loop either i.e., terminates. This is done by using the type described in Section 5.4.4 and making sure that it type-check when the "total" keyword is used.

5.4.6. Liveness

A protocol may never terminate - in fact most of the time the server side of a protocol is a loop - but it still can do some useful work in that loop. This property is called liveness.

6. Standard Library Tutorial

One of the ultimate goals of this document is to convince authors to use the techniques it describes to write their documents. Because doing so requires a lot of efforts, an important intermediate goal is to show authors that the benefits of computerate specifying are worth learning and becoming proficient in these techniques.

The best way to reach that intermediate goal is to apply these technique to documents that are in the process of being published by the IETF and if issues are found, report them to the authors. Doing that on published RFCs, especially just after their publication, would be unnecessarily mean. On the other hand doing that on all Internet-Drafts as they are submitted would not be scalable.

The best place to do a Computerate Specifying oriented review seems to be when a document enters IETF Last Call. These reviews would

then be indistinguishable from the reviews done by an hypothetical Formal Specification Directorate. An argument can be made that, ultimately, writing a specification for a document could be an activity too specialized, just like Security reviews are, and that an actual Directorate should be assembled.

Alas, it is clear that writing a specification from scratch (as in Section 5) for an existing document takes far more time than the Last Call duration would allow. On the other hand the work needed could be greatly reduced if, instead of writing that specification from scratch, packages and modules containing types and code were available for the parts that are reusable across specifications.

The types and code in a computerate specification form an Idris package, which is a collection of Idris modules. An Idris module forms a namespace hierarchy for the types and functions defined in it and is physically stored as a file. Section 7 describes how to build an specification that can be exported.

This section explains how to use the existing modules and packages that are available to import in a computerate specification as part of the standard library.

Packages and modules in the standard library fall into 2 categories:

Internal Modules: These are the modules that are not tied to an existing RFC but that are common to many specifications, so they are defined by the underlying package of this specification.
The modules in that category are explained in Section 6.1, their reference is in Section 8.1, and their API documentation is in Appendix B.

Packages for formal languages: Formal languages are used in documents to formalize some parts of them, so having libraries to formalize these formal languages also helps accelerating their verification.
The packages in that category are explained in Section 6.2, in Section 8.2, and its associated reference in Appendix B.

Together these packages and modules form the Computerate Specifying Standard Library (Section 8).

6.1. Internal Modules

This document is itself generated from a computerate specification that contains data types and functions that can be reused in future specifications, and as a whole is part of the standard library for computerate specifying. The following sections describe the Idris modules defined in that specification.

6.1.1. AsciiDoc Generation

Section 4 explained how implementing the "Asciidoc" and/or "Pretty" interfaces (by respectively using the code:[] and code::[] macros)

permits to format the result of a code evaluation so it can be inserted into a document.

Particularly, the inline code:[] macro generates a string of characters that can be safely inserted in an AsciiDoc document, but sometimes it is useful to instead generate an AsciiDoc fragment.

The "ComputerateSpecifying.Metanorma.Ietf" module contains types and functions that guarantee that the AsciiDoc text generated is compliant with its specification. All these types implement the "Asciidoc" interface so they can be directly used by the inline code macro.

In the following example the description of an Internet Header is converted into a "Block", which is then directly converted into an AsciiDoc block and inserted in the document:

```
> example : InternetHeader -> Block
> -- code elided

code:[example example1]
```

The "AsciiDoc" module is not limited to generating examples, as it can be used to generate any AsciiDoc structure from Idris code. In the future it would even be possible to use Natural Language Generation [NLG] to, instead of paraphrasing the English text from the Idris types, directly generate it.

The reference documentation for this module is in Section 8.1.1 and its API in Appendix B.1.3.

NOTE 1: Still work in progress, but there is plans to implement [RFC8792] so examples that do not fit in 72 columns page are folded according to the rules in that document.
NOTE 2: Also work in progress, but there is plans to reimplement some of the extensions in the asciidoctor-diagram extension so not only the text and SVG versions of its content are simultaneously available in the xml2rfc v3 generated document, but they can be programmatically generated,

6.1.2. Bit-Vectors

One of the most common type arrangement of data in a Protocol Data Units (PDU) is the bit-vector, which is a list of individual bits that can be interpreted either as flags or as a codepoint.

Although bit-vectors could be interpreted as numbers, arithmetic or comparison operations on them (other than equality) are meaningless, so they are not implemented as such.

The operations that can be done on bit-vectors are constraint by the number of bits they hold. Some operations work only on bit-vectors of the same exact size, like testing for equality and the bitwise operations. For instance the "a `and` b" operation will typecheck only if "a" and "b" are of the same size.

Other operations can act on bit-vectors of different sizes, but the types used will guarantee that the number of bits of the resulting value is correct. For instance "extract 5 a" will return a new bit-vector that is guaranteed to have a size of 5.

Bit-vectors can be specialized and made available for reuse. E.g., IP addresses and UUIDs are specialized bit-vectors and are defined in their own packages.

The reference documentation for this module is in Section 8.1.2 and its API in Appendix B.1.1.

NOTE: Still work in progress, but a future version of this library will use a deep embedded DSL so a list of operations on bit-vectors can be printed instead of the value resulting from these operations.

6.1.3. Abstract Numbers

Abstract numbers are numbers that are not associated with the counting of things, but on which arithmetic operations can be performed (as opposed to bit-vectors). Examples of abstract numbers used in PDU are checksums and CRCs.

The only type of abstract number defined at this time is "Unsigned n", which defines an unsigned number that uses a specific number of bits. This supplements the numerical types defined in Idris, types that are all using a number of bits that is a multiple of 8.

The reference documentation for the module containing the "Unsigned n" type and associated functions is in Section 8.1.3 and its API in Appendix B.1.5.

NOTE: This library is under heavy redesign to improve unsigned numbers (including by using a deep DSL), and to add Signed numbers in both ones' and twos' complement forms.

6.1.4. Denominate Numbers

A Denominate Numbers (DN) is any number that is associated with a quantity of something (called a dimension) and that is scaled by a unit. Example of DN are "5 bits", "10 meters", or "1 hour and 5 seconds", which respectively have a dimension of information, length and time.

DN are useful when combining different numbers of the same dimension but different units (like mixing up Metric and Imperial units, which caused the loss of the Mars Climate Orbiter in September 1999), or

when combining numbers of different dimensions (like dividing a length with time to obtain a measure of speed).

For instance gas consumption in the USA is measured in miles per gallon, whereas in Europe it is measured in liter per 100 kilometers. In that case "((5, mile / gallon), (liter / (100 * kilometer)))" is displayed as "47.04".

The reference documentation for this module is in Section 8.1.4 and its API in Appendix B.1.2.

6.1.5. Typed Petri Nets

> Never send a human to do a machine's job.
> — *Agent Smith in The Matrix (1999)*

Concurrent systems can be represented using two different families of techniques, algebraic and graphical. Algebraic techniques (e.g., process calculi) are mathematically well-defined, but lack an intuitive representation that would be useful to developers not completely familiar with these techniques.

On the other hand, graphical representations of concurrent systems (e.g., state machines) can be understood by a larger segment of developers, but generally lack a standardized and/or mathematical definition.

Petri Nets are at the intersection of these two techniques. They are typically graphical representations of concurrent processes, but are based on a well-defined mathematical theory. One way to look at Petri Nets is as a way to group multiple state machines together. A Petri Net also has the advantage that the same graph can be reused to derive other Petri Nets, e.g., Timed Petri Nets (that can be used to collect performance metrics) or Stochastic Petri Nets (which can be seen as a way to group multiple Markov chains together).

A Typed Petri Net (TPN) is an algebraic specification of a Petri Net, such as it can be expressed as an Idris value, and be easily reused for various purposes. TPNs are based on Colored Petri Nets, as defined in [Jensen09] and [Aalst11]. [Jensen07] is a shorter introduction to Colored Petri Net that should be read first. Particularly, section 2 contains the various definition of the terminology that is used in this document, augmented as follow:

* The word Color is used instead of the word Colour.

* *unification* is defined in the middle of the left column of page 6.

* *free variable* is defined in the middle of the right column of page 6.

A TPN that covers a whole protocol (i.e. client, network, and server) is useful to prove the properties listed in Section 5.4.4,

Section 5.4.5, and Section 5.4.6. But a TPN can also be designed so each part of the protocol is defined separately from the others, making it a Hierarchical TPN.

The reference documentation for this module is in Section 8.1.5 and its API in Appendix B.1.4.

6.1.5.1. Building a Typed Petri Net

The following example of TPN is converted from Figure 7 in [Jensen07]:

```
> No : Type
> No = Int

> Data : Type
> Data = String

> NoxData : Type
> NoxData = (No, Data)

> namespace Sender
>   export
>   sender : Module [NoxData, NoxData, No] ()
>   sender = do
>     packetsToSend <- port "Packets To Send" NoxData Both
>     nextSend <- place "NextSend" No {init=pure 1}
>     a <- port "A" NoxData Out
>     d <- port "D" No In
>     transition "Send Packet"
>       [input packetsToSend (No, Data) one,
>        input nextSend No one]
>       [(0, 0, 1, 0)]
>       [output (No, Data) packetsToSend pure,
>        output No nextSend pure,
>        output (No, Data) a pure]
>       (\((n, d), n') => pure ((n, d), n, (n, d)))
>     transition "Receive Ack"
>       [input nextSend No one,
>        input d No one]
>       empty
>       [output No nextSend pure]
>       (pure . snd)
```

Similarly, the following example of TPN is converted from Figure 11 in [Jensen07]:

```
> namespace Protocol
>   export
>   protocol : Top
>   protocol = top $ do
>     packetsToSend <- place "Packets To Send" NoxData
>       {init=[(1, "COL"), (2, "OUR"), (3, "ED "), (4, "PET"),
>       (5, "RI "), (6, "NET")]}
>     dataReceived <- place "Data Received" Data {init=pure ""}
>     a <- place "A" NoxData
>     b <- place "B" NoxData
>     c <- place "C" No
>     d <- place "D" No
>     instance "" sender [packetsToSend, a, d]
>     instance "" network [a, b, c, d]
>     instance "" receiver [dataReceived, b, c]
```

In these examples, the "place", "port", "input", "output", "transition", and "instance" functions are the combinators used to define Typed Petri Net modules in computerate specifications and are described in Section 8.1.5.

TPN modules are written as constants of type "Module xs ()", which is a Monad used to implement a deep embedded DSL. The Monad ensures that places, ports and instances all have a unique name inside a module. It also ensures that all places and ports used by transitions and instances are declared in the same module. Finally it ensures that all ports exported by a module are correctly mapped to a local place or port when imported as an instance.

Ultimately these combinators are not meant to be used as a way to directly design a TPN, as doing this is very tedious and error-prone. Instead the general advice is to use the graphical tool cpntools [Cpntools] to design a CPN and then to follow the step-by-step tutorial in Appendix D to convert it into a Typed Petri Net. Experience shows that, even for the simplest of protocols, systematically starting formalization by 1) designing a complete semantic type and 2) designing a top-level Petri Net, both in cpntools, is the most efficient way to proceed.

NOTE: It is planned to add to the tooling a graphical tool on top of TPN that will replace cpntools, which is starting to show its age. To do so each combinator will be enhanced with values that contains the graphical properties (position, size, color,...) that the graphical tool will use to display a TPN. Conversely the graphical tool will be designed to modify these properties in the source code in response to a user interface action, like moving or resizing a place, port, transition or instance. This will permit to continue to use the computerate specification as the storage format for a TPN, including its graphical representation.

6.1.5.2. Adding Time to a Typed Petri Net

Timed TPN are built by using Timed tokens (which are types wrapped into the "Timed" type) and by adding delays in transitions and arcs.

The following is an example of an implementation in CPN of a timer, here that will timeout after 100 units of time:

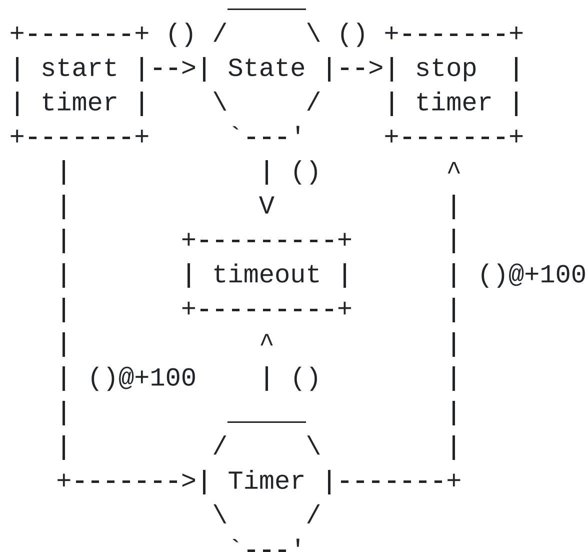

Figure 1

This CPN can be translated as the following Timed TPN:

```
> timer : Top
> timer = top $ do
>   state <- place "State" ()
>   timer <- place "Timer" (Timed ())
>   transition "start timer"
>     empty
>     empty
>     [output () state pure,
>      output () timer {delay=100} (\x => [timed x])]
>     (pure . dup)
>   transition "stop timer"
>     [input state () one,
>      input timer () {delay=100}
>        (\case (x ::: []) => Just (untimed x); _ => Nothing)]
>     empty
>     empty
>     (pure . fst)
>   transition "timeout"
>     [input state () one,
>      input timer ()
>        (\case (x ::: []) => Just (untimed x); _ => Nothing)]
>     empty
>     empty
>     (pure . fst)
```

Here the "Timer" place contains "Timed ()" values, which are added by the "start timer" transition. The added token will enable the "timeout" transition only after 100 unit of time have passed. Note that the clock used to calculate enablement of transitions is discrete, so a simulation does not have to really wait for that time. The "timeout" transition also removes the token from the "Timer" place, effectively making sure that it does not trigger a second time.

The "stop timer" transition is used when an event made that timer useless. In that case we want to remove the token from the "Timer" place ahead of its expiration time, so it is not fired later. This is the meaning of the delay inscription in the input arc, which is understood as a preemptive delay.

6.1.5.3. Verifying a Typed Petri Net

The TPN values created in Section 6.1.5.1 can be used to test, debug and validate a protocol.

This is done by running a simulation of the protocol. The plan is that this simulation will be driven from the future graphical interface but meanwhile it is possible to directly call a set of functions.

A simulation is executed when a succession of transitions occurs. It starts by building an initial marking, which is done by using the "initialMarking" function on a top TPN. For instance the command "initialMarking protocol" returns the following:

```
[{(1, "COL"), (2, "OUR"), (3, "ED "), (4, "PET"), (5, "RI "),
(6, "NET")}, {""}, {}, {}, {}, {}, {1}, {False, True}, {1}]
```

After this each transition occurrence is composed of 3 steps:

1. Find the list of transitions that are enabled from the current marking:

```
:let transitions = enabledTransitions marking top
```

2. Find the possible bindings for the transition selected. There is only one transition possible for that TPN instance, so we can use it to list the bindings:

```
:let bindings = bindings marking top (head transitions)
```

3. Create a new current marking from the transition and binding selected. Again we have only one possible binding in this example, so we can use it to update the marking after applying the binding and transition:

```
:let marking = bindings marking top (head transitions)
  (head bindings)
```

We can then calculate the next occurrence by looping back to step (1).

The simulation stops when there is no enabled transition for the current marking.

6.1.5.4. Deriving a Type from a Typed Petri Net

Designing and validating a Petri Net are essential tasks, but they cannot be directly used to guarantee that a process is following part or totality of this Petri Net.

To do so we need to generate a Sum type that encodes all the transitions as constructors. This type then can be used to build a

proof that a list of binding elements are valid according to that Petri Net.

In CPN, a binding is a list of (name, value) tuples, making it easy to read. In TPN we are using instead a tuple of the values as taken as input to the "Transition" inscription. That means that the variables are identified by position in this tuple, instead of by name.

The example below shows two constructors for the example Petri Net used in this document.

```
sendPacket' : (NoxData, No) -> List (NoxData, No, NoxData)

updateSendPacket : Marking xs -> (NoxData, No) -> Marking xs

data T210 : Marking xs -> Type where
  Init : T210 (initialMarking top)
  SendPacket : (binding : (NoxData, No)) ->
               (elem (fst binding) (packetToSend m)) =>
               (elem (snd binding) (Ns m)) =>
               (sendPacket' binding =
                 pure (fst binding, snd binding, fst binding)) =>
               T210 m ->
               T210 (updateSendPacket binding m)
```

6.1.5.5. Message Sequence Charts

The type generated from a TPN can then be used for various purposes but this section is about generated a Message Sequence Chart (MSC) for it.

MSCs are a common way to represent an example of execution of a protocol, i.e. of the interactions between the underlying state machines. Although sequence charts are often implicitly used to describe a protocol, that description can only be partial and thus cannot replace completely a description of the protocol by other means.

There is 4 steps to generate automatically an MSC that is guaranteed to be conform to the specification:

1. Design a Colored Petri Net of the behavior of the protocol. A Petri Net models all sides of a communications protocol. This includes a model of the network itself, which makes it the best way to generate an MSC.

2. Build a sequence of binding elements as described in Section 6.1.5.3.

3. Convert the TPN into a specialized type that guarantees that the list of (transition, binding) that represent the MSC to draw is valid according to the TPN. This is explained in Section 6.1.5.4.

E.g. the following instance typechecks with the generated type for the example CPN used in this document:

```
test : T210 ?
test = Init
       |> SendPacket ((1, "COL"), 1)
       |> TransmitPacket ((1, "COL"), True)
       |> ReceivePacket ((1, "COL"), "", 1)
       |> TransmitAck 1
       |> ReceiveAck (1, 1)
```

5. The last step is to pass that instance to a function that will generate the MSC:

```
....
code::[generateMsc test [(A, D), (B, C)]]
....
```

The parameters of the "generateMsc" function are the list of bindings and the name of the Petri Net places between which lines will be drawn. If for some reason the network manipulates the token between A and B, or B and C, the function will accordingly show that the packet is either lost, duplicated, delayed or even that a packet arrived from an unknown sender.

It is also possible to pass a user-defined function that will take as parameter a token as sent by places A or C and convert it in a packet that is to be showed after the MSC itself.

This function can be provided by the type of proofs described in Section 5.4.1. That means that, as long as we have a proof of isomorphism between them, we can use Semantic Types directly in our TPN instead of Wire Types, making the model simpler.

6.2. Formal Language Packages

When different representations of a specification share some common characteristics, it is usual to generalize them into a formal language.

One shared limitation of these languages is that they cannot always formalize all the constraints of a specific data layout, so they have to be enriched with comments.

Another consequence is the proliferation of these languages, with each new formal language trying to integrate more constraints than the previous ones. For that reason Computerate Specifying does not favor one formal language over the others, and will try to provide code to help use all of them.

Most of the formal languages used at the IETF already come with a set of tools that permits to verify that their text representation in an RFC is syntactically correct. Instead the formalisms introduced in the standard library for these formal language are about generating text that is correct by construction, thus making these tools redundant.

But going beyond merely ensuring that the textual representation of a formal language, it is also a goal to ensure that the examples in RFCs that uses a formal language are not just correct according to the specification, but also to their description in that formal language. This, again, is not done by verifying that examples are compatible with the formal language, but by permitting to generate only examples that are correct by construction.

A useful analogy for an RFC is to think of it as a dam in a river, i.e., an immutable separation between a set of activities happening upstream (Internet-Drafts, reviews, using tools to verify the formal languages, etc..), and a set of activities happening downstream after publication (implementation, test suites, erratas, etc..). A computerate specification is at the same time a way to collect all the upstream tools needed to build that RFC, but also making sure that the activities downstream can be done on the strong foundations brought by the use of the Idris type system.

6.2.1. Formal Languages Defined in RFCs

The following sections describe the support in the standard library for formal languages that are defined in an RFC.

6.2.1.1. Augmented BNF (ABNF)

Augmented Backus-Naur Form (ABNF) [RFC5234] is a formal language that can be used to describe a text based PDU.

An ABNF specification can be described by the functions defined in the "RFC5234.Abnf" module, each of them mapping one of the operators defined in Section 3 of [RFC5234].

For instance the following definition:

```
> alpha : Abnf True
> alpha = rule "ALPHA" $ hexRange 0x41 0x5a <|> hexRange 0x61 0x7a
```

is equivalent to the ABNF rule:

```
ALPHA =  %x41-5A / %x61-7A
```

Using this language permits to generate an ABNF specification that is guaranteed to be syntactically correct. But the language imposes constraints that go beyond mere syntax and also guarantee that the generated ABNF specification is semantically correct.

For instance the following ABNF is syntactically correct but it will take an infinite amount of time to parse something with it:

```
total
bad : Abnf True
bad = bad "a"
```

Here the use of the "total" keyword will ensure that the Idris typechecker will reject specifications that will loop forever.

On the other hand this language can describe the parsing of infinite streams. For instance the following matches an infinite sequence of "a" or "A":

```
total
good : Abnf True
good = "a" good
```

Together, the functions used to describe an ABNF specification form a DSL with deep embedding. Because of this deep embedding, the resulting internal structure can be used in various ways.

The primary use of that internal structure is to format a textual representation of the ABNF specification that can be directly inserted in the documentation part of a computerate specification. The internal structure implements the "Pretty" interface format automatically the output.

By default the ABNF specification is displayed exactly as entered.

A secondary use of that internal structure is as an index on the `"Example s g`" type, which permits to build proofs that specific strings are valid according to an ABNF grammar. But ABNF itself is not always powerful enough to encode all the constraints needed to

describe a PDU. An example of that is the following subset of the ABNF defined in [I-D.rivest-sexp]:

```
sexp  =  list / token

list  =  "(" sexp ")"

token =  1*DIGIT ":" *OCTET
```

In the "token" rule the constraint that the actual number of OCTET terminal values allowed is provided by the number on the left of the colon cannot be encoded in ABNF. Because of that a valid example of a Sexp should be built from a separate type:

```
> mutual
>   data Sexp = T (List Int) | L SexpList

>   data SexpList : Type where
>     Nil : SexpList
>     (::) : Sexp -> SexpList -> SexpList

> t : String -> Sexp
> t = T . map cast . unpack

> l : SexpList -> Sexp
> l = L
```

Then a function with type "Sexp -> (s : List Int ** Example s sexp))" can be implemented to generate examples that are by construction valid according to the ABNF, and that can be directly inserted in the document, such as:

```
> example : Sexp -> (s : List Int ** Example s Main.sexp)
> -- code elided

> ex : Sexp
> ex = l [t "a", t "bc", l [t "def"]]

code::[example ex]
```

will generate:

```
(1:a2:bc(3:def))
```

The ABNF DSL and the ability of building proofs for examples subsume most of the existing ABNF verification tools, making them redundant.

The "RFC5234.Core" module contains the definitions for common ABNF rules and is meant to be imported by computerate specifications that define ABNF specification that use them.

6.2.1.2. Structured Field Values for HTTP

Structured Field Values for HTTP [RFC8941] is a common syntax for HTTP header and trailers fields that is more restrictive than traditional HTTP field values.

The "RFC8941" module permits to build examples that are valid Structured Field Values (SFV) by using a Domain Specific Language (DSL).

SFV are composed of 4 different types of lists, each with different properties. The two internal types of list, parameter list and inner list, use Idris lists but each carries different values.

A parameter list contains tuples made of a string acting as a key, and of a typed value, which can be an item or "nothing" that representes a key without attached value. A tuple is created used the "#" operator, which can be thought as the equal sign. An example of parameters is "["a" # int 1, "b" # nothing]".

Similarly an inner list uses the Idris List notation and contains items each with an optional parameter list. An example of inner list is "[str "a", int 1 {p=["a" # int 1, "b" # nothing]}]". If present, the parameter list must be passed as an implicit Idris parameter named "p".

The two other types of lists in SFV are top level lists and are using sequencing with the "do" notation.

The "SfList" type is made of a sequence of items or inner lists. The items and the inner lists can themselves carry a parameter list, using the same syntax:

```
a : SfList
a = do
  int 1
  str "a" {p=["a" # int 1, "b" # nothing]}
  list [str "a", int 1 {p=["a" # int 1, "b" # nothing]}]
```

Similarly the "SfDict" type is a sequence of tuples made of a string acting as a key and of an item, an inner list or "nothing". All these, including the nothing item, can carry a list of parameters, as in this example:

```
a : SfDict
a = do
  "a" # int 1
  "b" # str "a" {p=["a" # int 1, "b" # nothing]}
  "c" # list [str "a", int 1 {p=["a" # int 1, "b" # nothing]}]
  "d" # nothing  {p=["a" # int 1, "b" # nothing]}
```

Finally the "SfItem" type is an item that can carry a list of parameters as in this example:

```
a : SfDict
a = str "a" {p=["a" # int 1, "b" # nothing]}
```

7. Specification Package Tutorial

The previous section was about specifications that reuse other specifications by importing the types and functions defined in them. This section is about organizing a computerate specification such as the types and functions defined in it can be imported and used by other specifications.

Such specifications export 4 or 5 components:

An Idris package: This is the binary artifact that the code in an importing specification will use. E.g., this is what a specification using ABNF will use to define the ABNF specific to the standard described in that document.
The Idris package is built from the lipkg file that doubles as the Asciidoc root document. The "modules" statement in that file lists all the idr and lidr files that will compose that package.

A tutorial: This is a document that guides the reader step by step in the use of the Idris package. This document may contain examples, which may themselves be generated by the Idris API.
The tutorial is an adoc or lidr file. Using an lidr file makes writing examples easier, but the code in that file is not part of either the Idris package or the API documentation.

A reference: This is a document that explains the Idris package as a whole i.e, grouping explanations by feature.
This is the document part of the lipkg files that compose the API documentation.

An API documentation: This is a document that lists all the Idris types and functions in alphabetic order, together with structured comments, for the Idris package.
This document is automatically generated from the structured Idris comments in the idr and lidr files that compose the reference document.

An eventual bibliography: This document (or documents if both normative and informative bibliographies are needed) contains the bibliographic items cited in the other exported documents.

Such a specification can be defined in an original computerate specification or in a retrofitted specification (RFC or I-D that was later enriched with Idris code). But an RFC cannot be modified, and the authors of an I-D may be different from the authors of the Idris code and unwilling to integrate the documentation in their document. Because of that there are multiple scenarios on how to publish the documentation for an Idris API:

In the same document: This is possible when both documentation and specification are under the control of the same authors. In this document, the documentation for the modules that compose the Computerate Specification Standard Library are published in this document, with the tutorials in Section 6, the references in Section 8.1 and the API documentations in Appendix B.1.

In an separate document: The tutorial, reference and API documentation documents can simply be included in a separate Internet-Draft whose sole purpose is to carry that documentation.

In the Standard Library: Some retrofitted specifications are deemed essential for authors of specifications, so the documentation for these retrofitted specification is part of this document. This is the case for the ABNF documentation (Section 6.2.1.1, Section 8.2.1, and Appendix B.2) and the APHD documentation (Section 6.2.1.2, Section 8.2.2, and Appendix B.3).

8. Standard Library Reference

This section is contains the reference documentation for all the packages distributed as part of the tooling.

8.1. Internal Modules

The following internal modules are available in the tooling.

*ComputerateSpecifying.BitVector

*ComputerateSpecifying.Dimension

8.1.1. Metanorma.Ietf

The "MetaNorma.Ietf" module provides a way to programmatically build an AsciiDoc document without having to worry about the particular formatting details. The types in this library are not meant to be used directly, but as the return type of functions in modules that generates text for insertion in a document.

At the difference of the AsciiDoc rendering process that tries very hard to render a document in any circumstances, the types in this

module are meant to enforce the generation of AsciiDoc that results in valid xml2rfc v3 document.

"Asciidoc" is an interface that can be implemented to transform an Idris type into a character string that can be safely inserted in an AsciiDoc document.

The "Text", "Must", "MustNot", "Required", "Shall", "ShallNot", "Should", "ShouldNot", "Recommended", "May", "Optional", "HardBreak", "Contact", "Comment", "Italic", "Link", "Index", "Citation", "Bold", "Subscript", "Superscript", "Monospace", "Unicode", "Cross", and "Attribute" constructors are used to build individual inline elements.

The "Paragraph" constructor is used to build a paragraph, and is composed from a list of inline elements.

8.1.2. BitVector

"BitVector" is a dependent type representing a list of bits, indexed by the number of bits contained in that list. The type is inspired by Chapter 6 of [Kroening16] and by [Brinkmann02].

A value of type "BitVector n" can be built as a series of zeros ("bitVector") or can be built by using a list of "O" (for 0) and "I" (for 1) constructors. E.g., "[O, I, O, O]" builds a bit-vector of type "BitVector 4" with a value equivalent to 0b0100.

Bit-vectors can be compared for equality, but they are not ordered. They also are not numbers so arithmetic operations cannot be applied to them.

Bit-vectors can be concatenated ("(++)"), a smaller bit-vector can be extracted from an existing bit-vector ("extract"), or a bit-vector can be extended by adding a number of zeros in front of it ("extend").

The usual unary bitwise ("shiftL", "shiftR", "not") operations are defined for bit-vectors, as well as binary bitwise operations between two bit-vectors of the same size ("and", "or", "xor")

Finally it is possible to convert the bit at a specific position in a bit-vector into a "Bool" value ("test").

8.1.3. Unsigned

A value of type "Unsigned n" encodes an unsigned integer as a "BitVector" of length "n".

8.1.4. Dimension

This module permits to manipulate denominate numbers, which are numbers associated with a unit. Examples of denominate numbers are

"(5, meter / second)" (which uses a unit of speed), or "(10, meter * meter * meter)" (which uses a unit of volume).

In this module a denominate number is a value of type "Denominate xs". It carries one number as the fraction of two Integer. Its type is indexed over a list of dimensions in canonical order, each associated with an exponent number. All together this type can represent any unit that is based directly or indirectly from the base dimensions defined in the "Dimension" type.

A unit dimension can be chosen as one of the predefined physical dimensions, "Time" and "Length".

The non-physical unit dimension "Info" (for "unit of information") is also defined.

It is also possible to define other non-physical unit dimensions as needed:

```
apple : Denominate [(Q "Apple", 1)]
apple = Intro 1 1
```

Finally dimensionless numbers (i.e., denominate numbers with type "Dimensionless") can be constructed by using the "none" unit, as in "(10, none)". The "fromInteger" and "fromDouble" functions also construct dimensionless numbers.

Denominate numbers are constructed by using the "toDenominate" function on a tuple made of a number and a unit. E.g., "toDenominate (5, megabit)" will build the denominate number 5 with the "megabit" unit. The "cast" can also be used when the type of the returned value can be inferred.

For simplicity, examples of denominate numbers in this document are given in their tuple form.

Denominate numbers can be added, subtracted or negated (respectively "+", "-", and "neg"). All these operations can only be done on denominate numbers with the same exact dimension, and the result will also carry the same dimension. This prevents what is colloquially known as mixing apples and oranges.

For the same reason, adding a number to a non-dimensionless denominate number is equally impossible.

The "*", "/", and "recip" operations respectively multiply, divide and calculate the reciprocal of denominate numbers. These operations can be done on denominate number that have different types, and the resulting dimension will be derived from the respective dimension of the arguments. E.g. multiplying "(5, meter)" by "(6, meter)" will return the equivalent of "(30, meter * meter)".

Also multiplying a denominate number by a (dimensionless) number is possible e.g., as in multiplying "(5, meter)" by "(10, none)", which will return the equivalent of "(50, meter)".

Denominate numbers can be evaluated into a "Double" value by using the "fromDenominate" function. The second parameter of the function is the unit that we want the result expressed in. E.g., "fromDenominate (toDenominate (5, meter / second)) (kilometer / hour)" will return 18.0, which is 5 meters per second expressed in kilometers per hour.

Denominate numbers can also be evaluated into a rounded down Int by using the "fromDenominateToInt" function.

Ultimately we want to insert a denominate number, together with its unit, as text in a computerate specification. This is done by using a tuple made of the denominate number and of the expected unit. E.g., code:["(toDenominate (2, byte), bit)"] will insert "16" in the text.

Adding the "impl=unit" attribute to the macro will insert the textual representation with the correct plural of the unit instead of its value. E.g., code:["(toDenominate (2, byte), bit)",impl=unit] will insert "bits" in the text.

It is also possible to insert the sequence of operations done on a denominate number, instead of the result of the evaluation, by adding the "impl=formula" attribute to the code macro. E.g., code:["(toDenominate (2, byte), bit)",impl=formula] will insert "2 * 8" in the text.

The "name" function can be used to define a variable that will be displayed in the formula.

The "Size" interface can be implemented to retrieve the size of a type as a denominate number of dimension "Info".

For each dimension that is useful for Internet standards (time, length, and information) a list of constants that represents units of that dimension are pre-defined.

Units that uses a prefix are automatically generated, which is the case for SI units for the "Time" dimension (i.e., from "yoctosecond" to "yottasecond"), SI units (only positive powers of 10) for the "Info" dimension (i.e., from "kilobit" to "yottabit"), and IEC units (positive powers of 2) for the "Info" dimension (i.e., from "kibibit" to "yobibit").

Additional constants like "minute", "hour", "day", "byte", "wyde", "tetra", "octa", etc, complement the standard units. The "byte", "wyde", "tetra", and "octa" units are defined in page 4 of [Knuth05].

8.1.5. Tpn

The "Tpn" module permits to build Typed Petri Nets. It is designed to mimic Hierarchical Colored Petri Nets so conversions could be done mechanically.

8.1.5.1. Building a TPN

The "Timed" type is used as a wrapper around another type when its values need to be associated with time, with the "timed" and "untimed" functions used to respectively unwrap and wrap a value.

The "input" function builds an input arc for a transition. The inscription is a function that takes one or more tokens as input and generates an optional value of the input arc type. The optionality of the output type permits to decide if the tokens can or cannot be used in the transition.

The "one" function can be used when a unique token is to be taken from the place and passed to the function.

The type of the codomain of that function may be different from the type of the place to permit to do some manipulation on the token itself.

These two features can be combined together by using pattern matching inside the inscription.

The delay value (which defaults to 0) in an input arc indicates a preemptive time from which a "Timed" wrapped token will enable the transition.

The "inhibitor" function builds an inhibitor arc for a transition. An inhibitor arc is a variant of input arcs that can trigger a transition only if the place it is connected is empty.

The "reset" function builds a reset arc for a transition. A reset arc is a variant of input arcs that always trigger a transition regardless of the content of the connected place, and that will empty the connected place when triggered.

The "output" function builds an output arc for a transition. An output arc takes a value of the type created by a transition, converts it using the function (acting as inscription) into a list of tokens and insert that list in the destination place. The list of tokens can be empty if no token has to be added, or can contain one or more tokens.

The order of the parameters of that function is the inverse of the "input" function to show that the function codomain is the type of the place.

An output arc can contain a delay value (which defaults to 0) that will be added to any token that is wrapped in the "Timed" type.

The "Module" Monad is used to group together places, ports, transitions and instances of other modules into a module by using the "do" notation.

The "place" function builds a Petri Net place, which is a structure that holds state in the form of tokens. A place has a name, a type (or color) and an initial content which defaults to emptyness.

Alternatively the "port" function can be used to build a Petri Net port, which is used to define the interface of a Petri Net module and, like a place, has a name and a type. A port does not have an initial content but contains the direction tokens are allowed to flow between an outer module instance and a module definition.

Note that the names carried by most elements on a TPN are only used for documentation purpose. That means that a name can be empty or can be used multiple times.

The "transition" function brings together input arcs (including inhibitor and reset arcs), output arcs, and the function that will convert input tokens into output tokens.

The "unifications" list contains a 4-tuples in the form (input-index, data-index, input-index, data-index). Each input-index is an index in the list of inputs. Each data index is an index in the tuple representing the codomain of the inscription. The value of the first input-index/data-index will be unified with the value of the second input-index/data-index.

The "instance" function imports another module in this module and assign a local Place or Port to each Port imported from that module.

Note that using an instance of a module is different from using a module directly. A port is a temporary placeholder for a place, so a port needs at some point to be mapped to a place, that will contain the actual state. The is the role of an instance to do that mapping.

Finally the "top" function wraps a module that has no exported ports, so it can be used as input to the functions described in the next section.

The core of TPN is implemented by the functions described above. It is possible to combine these functions to build more complex functions to simplify the construction of a TPN, or to implement variants of CPN. In any case these complex functions are always transformed into core functions, making the simulators and other processes that take as input a TPN simpler to implement and independent from these complex functions. The functions in the following paragraphs are built that way.

The "free" function adds the possibility of using a free variable as part of the inscription on a transition. Using a free variable requires that its type implements the interface "Enum".

The "addStep", "addPriorities", and "addTime" functions add the transitions and places needed to implements the global step, transition priorities, and time features to an existing TPN.

8.1.5.2. Verifying a TPN

The "Marking" type describes a marking, which is a set of places and their content. It represents the global state of the system described by a TPN. A marking is indexed over the list of types of the places in it, such as we can guarantee that the marking structure never changes when transitions are fired.

The initial state of the system is generated by the "initialMarking" function.

A "Transition" represents the unique path through modules to a transition.

The "enabledTransitions" function generates a list of all the transitions that are enabled by a specific marking.

A "Binding" is a tuple that has the same size as the number of input arcs and that contains a token from each of the respective places such as substituting the input places in a transition by places that contain only that value will still enable the transition.

The "bindings" function lists all the possible bindings for a specific marking and an enabled transition.

The "transition" function transforms a marking into a new marking by applying a specific binding to a transition.

8.1.5.3. Deriving a Type From a TPN

The "deriveType" function takes a top-level TPN (as an instance of the type "Top") and generates the declaration of a new Sum type, with one constructor per transition, plus one constructor for the initial marking. This type then can serve to define a proof that a list of (transition, binding) tuples are valid according to that Petri Net.

On the generated type, the "Init" constructor builds an initial marking. Then each other constructors are used to validate a sequence of binding elements. Each non-initial constructor carries a set of proofs, one per input arc that prove that the binding is originating from the places in the marking, and one that prove that this transition is enabled, by showing that the transition using that binding is deterministic. Finally each transition updates the marking according to the output arcs, i.e removing and adding tokens.

8.2. Formal Language Packages

The following modules are available in the tooling.

* RFC5234

* RFC5234.Core

* RFC8941

8.2.1. RFC 5234 (ABNF)

The "RFC5234" module permits to build ABNF grammars. It contains a DSL designed to mimic closely the syntax described in [RFC5234].

8.2.1.1. Building an ABNF

All ABNF elements use the Idris type "Abnf Bool". The index must be set to "True" if the element is consuming one or more characters, "False" if not. This is the mechanism that permits to verify that an ABNF specification can be used to parse in finite time.

The section numbers below refer to sections in [RFC5234].

Giving a name to an element, as defined in section 2.2, uses the "rule" function. It takes the name of the rule and an ABNF element as parameters.

Concatenation of elements, as defined in section 3.1, uses the "(>>)" infix operator or the "do" notation to sequence elements.

Alternation of elements, as defined in section 3.2, uses the "<|>" infix operator to define alternative between elements.

Grouping of elements, as defined in section 3.5, uses the "group" function. It takes an element as parameter.

Variable repetition of elements, as defined in section 3.6, uses the "repeat" function. This function takes a value of type "Text.Quantity" and an ABNF element as parameters. In additional to the quantities defined in the "Text.Quantity" modules, the "many" and "some" can be used as equivalent to respectively "" **and "1**".

A sequence repetition for an exact number of repetitions, as defined in section 3.7, uses the "exact" function. This function takes the number of repetitions and an element as parameters.

Optional element, as defined in section 3.8, uses the "optional" function. This function takes an element as parameter.

Terminal values, as defined in section 2.3, use the "binTerm", "decTerm", "hexTerm", "binTerms", "decTerms", "hexTerms", and "string" functions. The "binTerm", "decTerm", and "hexTerm" functions take one parameter, which is the terminal value. The

"bin", "dec", and "hex" prefixes refer to the encoding that will be used to display the equivalent ABNF, not to the encoding that is used for the parameter value, which is independent. Similarly the "binTerms", "decTerms", and "hexTerms" functions take one parameter, which is a non-empty list of terminal values.

The "string" function takes a string of case insensitive US-ASCII character as parameter.

Value range alternative, as defined in section 3.4, uses the "binRange", "decRange", and "hexRange" functions. It takes two parameters as lower and higher bounds for the range. Alternatives are generated in reverse order if the first parameter has an higher value than the value of the second parameter.

Comments can be added at the top level of a grammar, or for individual rules. At the top level an "empty" rule must be used for the second parameter:

```
do foo
   comment (singleton "Top level comment") empty
   bar
```

For individual rules, the comment must be added after the rule function:

```
do foo
   rule "rule" $ comment (singleton "Top level comment") $ myrule
   bar
```

8.2.1.2. Generating and Verifying ABNF specifications

A value of "Abnf" is automatically converted in its text form when used with one of the code macros. When using the inline macro, the ABNF is formatted using the "Asciidoc" interface. When using the block macro, the ABNF is formatted using the "Pretty" interface, which follows the same presentation format than used in RFC 5234.

The "Example string grammar" type permits to build a proof that a string is a valid example according to a specific ABNF grammar. Such proof can then be directly inserted in the document by using the inline code macro, which will display the verified example. In cases where the length of the example exceeds the maximum allowed width of a rendered document, using the block code macro will format the string according to [RFC8792].

8.2.1.3. Common Rules

The "RFC5234.Core" module contains a set of common rules that are often reused by ABNF.

8.2.2. RFC 8941 (Structured Field Values for HTTP)

The 3 types corresponding to the 3 top level Structured Data Types are "SfList", "SfDict", and "SfItem". The "do" notation ensures that "SfList" and "SfDict" contain at least one element.

The two additional types corresponding to the internal containers Inner List and Parameters are "InnerList" and "Parameters". These containers can be empty,

In addition to the constraints explicit to the structures described above, the "SfDict" and "Parameters" types ensure that the keys used are unique.

An item can carry an integer value, a decimal value, a character string value, a token value, a binary value, or a boolean value by using respectively the overloaded "int", "dec", "str", "tok", "bin", or "bool" functions. When allowed, the overloaded "list" and "nothing" functions complete the list of functions that can be used to create an item. Each of these function prevents using value that do not match the constraints imposed of each of them.

A value of the "SfList", "SfDict", and "SfItem" types can be inserted in a document by using the inline code:[] macro.

A example that includes the attribute name can be inserted by using a value of type "Attribute". The "attributeItem", "attributeList", and "attributeDict" respectively build attributes that contains an Item, a List and a Dictionary.

NOTE: There is work in progress to ensure that the examples generated are valid according to the ABNF.

9. Informative References

[Aalst11] Aalst, W. V. D. and C. Stahl, "Modeling Business Processes: A Petri Net-Oriented Approach", Cambridge, Mass:MIT Press, 2011.

[AsciiBib] "AsciiBib", (accessed August 20, 2020), <https://www.relaton.com/specs/asciibib/>.

[AsciiDoc] Wikipedia, The Free Encyclopedia, s.v., "AsciiDoc", (accessed August 20, 2020), <https://en.wikipedia.org/wiki/AsciiDoc/>.

[Asciidoctor] "Asciidoctor", (accessed August 20, 2020), <https://asciidoctor.org/docs/user-manual>.

[Bennett15]
Bennett, B., "Logically Fallacious: The Ultimate Collection of Over 300 Logical Fallacies", 2015.

[Blockquotes] "Markdown-style blockquotes", (accessed August 20, 2020), <https://asciidoctor.org/docs/user-manual/#markdown-style-blockquotes>.

[Bornat05] Bornat, R., "Proof and Disproof in Formal Logic: An Introduction for Programmers", Oxford ; New York:Oxford University Press, 2005.

[Brady17] Brady, E., "Type-Driven Development with Idris", Shelter Island, NY:Manning Publications Co, 2017.

[Brinkmann02] Brinkmann, R. and R. Drechsler, "RTL-Datapath Verification using Integer Linear Programming", IEEE Computer Society, 2002, <http://dl.acm.org/citation.cfm?id=835389>.

[Christiansen16] Christiansen, D. and E. C. Brady, "Elaborator reflection: Extending Idris in Idris", ACM Press-Association for Computing Machinery, 2016, <https://research-repository.st-andrews.ac.uk/bitstream/handle/10023/9522/elab_reflection_paper.pdf>.

[Community20] Community, T. M., "The Lean Mathematical Library", 20 January 2020, <http://arxiv.org/abs/1910.09336>.

[Copyright] "Machine-readable debian/copyright file", (accessed August 20, 2020), <https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/>.

[Cpntools] "CPN Tools: A tool for editing, simulating, and analyzing Colored Petri nets", (accessed August 20, 2020), <http://cpntools.org/>.

[Curry-Howard] Wikipedia, The Free Encyclopedia, s.v., "Curry-Howard correspondence", (accessed August 20, 2020), <https://en.wikipedia.org/wiki/Curry-Howard_correspondence>.

[I-D.bortzmeyer-language-state-machines]
Bortzmeyer, S., "Cosmogol: a language to describe finite state machines", Work in Progress, Internet-Draft, draft-bortzmeyer-language-state-machines-01, 13 November 2006, <https://datatracker.ietf.org/doc/draft-bortzmeyer-language-state-machines>.

[I-D.mcquistin-augmented-ascii-diagrams] McQuistin, S., Band, V., Jacob, D., and C. Perkins, "Describing Protocol Data Units with Augmented Packet Header Diagrams", Work in Progress, Internet-Draft, draft-mcquistin-augmented-ascii-diagrams-09, 25 October 2021, <https://

datatracker.ietf.org/doc/draft-mcquistin-augmented-ascii-diagrams>.

[I-D.ribose-asciirfc] Tse, R., Nicholas, N., Lau, J., and P. Brasolin, "AsciiRFC: Authoring Internet-Drafts And RFCs Using AsciiDoc", Work in Progress, Internet-Draft, draft-ribose-asciirfc-08, 17 April 2018, <https://datatracker.ietf.org/doc/draft-ribose-asciirfc>.

[I-D.rivest-sexp] Rivest, R. L., "S-Expressions", Work in Progress, Internet-Draft, draft-rivest-sexp-00.txt, 4 May 1997, <https://people.csail.mit.edu/rivest/Sexp.txt>.

[Idris2] "Idris2: A Language with Dependent Types", <https://idris2.readthedocs.io/en/latest/>.

[Jensen07] Jensen, K., Kristensen, L., and L. Wells, "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems", 31 May 2007, <http://webdiis.unizar.es/~lrecalde/doctorado/bibliografia/coloreadas.pdf>.

[Jensen09] Jensen and L. Kristensen, "Coloured Petri Nets: Modelling and Validation of Concurrent Systems", Dordrecht ; New York:Springer, 2009.

[Knuth05] Knuth, D. E., "The Art of Computer Programming", Upper Saddle River, NJ:Addison-Wesley, 2005.

[Knuth92] Knuth, D. E., "Literate Programming", Stanford, Calif.:Center for the Study of Language and Information, 1992.

[Kroening16] Kroening, D. and O. Strichman, "Decision Procedures: An Algorithmic Point of View", Berlin s.l:Springer Berlin, 2016.

[Linear-Resources] "Linear Resources", (accessed August 20, 2020), <https://idris2.readthedocs.io/en/latest/app/linear.html>.

[Metanorma] "Metanorma", (accessed August 20, 2020), <https://www.metanorma.com/>.

[Metanorma-IETF] "Metanorma-IETF", (accessed August 20, 2020), <https://www.metanorma.com/author/ietf/>.

[Mimram20] Mimram, S., "Program = Proof", 2020, <http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf>.

[Minutes] "Trust Meeting Minutes Tuesday March 16, 2021", (accessed May 24, 2021), <https://trustee.ietf.org/wp-content/uploads/2021-03-16-trust-minutes.pdf>.

[Momot16]
Momot, F., Bratus, S., Hallberg, S., and M. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them", Boston, MA, USA:IEEE, November 2016, <http://ieeexplore.ieee.org/document/7839788/>.

[Nederpelt14] Nederpelt and H. Geuvers, "Type Theory and Formal Proof: An Introduction", Cambridge ; New York:Cambridge University Press, 2014.

[NLG] Wikipedia, The Free Encyclopedia, s.v., "Natural language generation", (accessed November 19, 2021), <https://en.wikipedia.org/wiki/Natural_language_generation>.

[RFC-Guide] "RFC Style Guide", (accessed August 20, 2020), <https://www.rfc-editor.org/styleguide/part2/>.

[rfc-prolog] Petit-Huguenin, M., "RFC Prolog database", 28 August 2021, <git://shalmaneser.org/rfc-prolog>.

[RFC0761] Postel, J., "DoD standard Transmission Control Protocol", RFC 0761, DOI 10.17487/RFC0761, January 1980, <https://www.rfc-editor.org/info/rfc0761>.

[RFC0791] Postel, J., "Internet Protocol", RFC 0791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/info/rfc0791>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4012] Blunk, L., Damas, J., Parent, F., and A. Robachevsky, "Routing Policy Specification Language next generation (RPSLng)", RFC 4012, DOI 10.17487/RFC4012, March 2005, <https://www.rfc-editor.org/info/rfc4012>.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC4912] Legg, S., "Abstract Syntax Notation X (ASN.X)", RFC 4912, DOI 10.17487/RFC4912, July 2007, <https://www.rfc-editor.org/info/rfc4912>.

[RFC4997] Pelletier, G. and R. Finking, "Formal Notation for RObust Header Compression (ROHC-FN)", RFC 4997, DOI 10.17487/RFC4997, July 2007, <https://www.rfc-editor.org/info/rfc4997>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 5234, DOI 10.17487/

RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5511] Farrel, A., "Routing Backus-Naur Form (RBNF): A Syntax Used to Form Encoding Rules in Various Routing Protocol Specifications", RFC 5511, DOI 10.17487/RFC5511, April 2009, <https://www.rfc-editor.org/info/rfc5511>.

[RFC6020] Björklund, M., "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)", RFC 6020, DOI 10.17487/RFC6020, October 2010, <https://www.rfc-editor.org/info/rfc6020>.

[RFC6940] Jennings, C., Lowekamp, B., Rescorla, E., Baset, S., and H. Schulzrinne, "REsource LOcation And Discovery (RELOAD) Base Protocol", RFC 6940, DOI 10.17487/RFC6940, January 2014, <https://www.rfc-editor.org/info/rfc6940>.

[RFC7991] Hoffman, P., "The "xml2rfc" Version 3 Vocabulary", RFC 7991, DOI 10.17487/RFC7991, December 2016, <https://www.rfc-editor.org/info/rfc7991>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/info/rfc8489>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[RFC8656] Reddy, T., Ed., Johnston, A., Ed., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 8656, DOI 10.17487/RFC8656, February 2020, <https://www.rfc-editor.org/info/rfc8656>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[RFC8941]
Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/info/rfc8941>.

[Stump16] Stump, A., "Verified Functional Programming in Agda", ACM Books series, 2016.

[TLP5] "Legal Provisions Relating to IETF Documents", (accessed August 20, 2020), <https://trustee.ietf.org/license-info/IETF-TLP-5.htm>.

[Zave11] Zave, P., Laboratories, T., and F. Park, "Experiences with Protocol Description", 2011, <https://www.researchgate.net/profile/Pamela_Zave/publication/266230560_Experiences_with_Protocol_Description/links/56eaf9fb08ae9dcdd82a6590.pdf>.

[Zotero] "Zotero | Your personal research assistant", (accessed Oct 06, 2021), <https://www.zotero.org/>.

Appendix A. Command Line Tools

> Creation is in part merely the business of forgoing the great and small distractions.
> — *E. B. White*

The tooling supporting the techniques described in this document is designed to maximize the productivity by permitting to work disconnected from the Internet most of the time and from the World Wide Web at all times.

A.1. Installation

The computerate command line tools are run from a Docker image, so the first step is to install the Docker software or verify that it is up to date (https://docs.docker.com/install/).

Note that for the usage described in this document there is no need for Docker EE or for having a Docker account.

The Docker image is distributed as a BitTorrent, so a BitTorrent client is also needed to download the image.

No other tools beyond these are needed after the image is installed, as it contains everything that is needed for the whole life cycle of computerate specifications, including git and the vim text editor.

The following instructions assume a Unix based OS, i.e. Linux or MacOS. Lines separated by a "\" character are meant to be executed as one single line, with the "\" character removed.

To install the computerate tools, the fastest way is to download and install the Docker image using BitTorrent. The BitTorrent magnet URI for the version distributed with this version of the document is:

magnet:?xt=urn:btih:f4e665a2f20c2ac6216bca44b06f3902b2223c6e&dn=tools-16.tar.xz

After this, the image can be loaded in Docker as follow:

```
docker load -i tools-16.tar.xz
```

Note that a new version of the tooling is released at the same time a new version of this document is released, each time with a new BitTorrent magnet URI.

After the first installation it is recommended to create an alias to make the use of the tools easier. This can be done by installing the /opt/tools in your path:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools computerate cp /opt/tools .
sudo mv tools /usr/bin/
```

The subsequent examples in this appendix will use this alias.

A.2. Authoring a Computerate Specification

The following sections explain how to use the various tools that are needed to author and process a computerate specification, but does not explain how a computerate specification should be formatted.

Examples of formatting are available in the templates provided in the Docker image.

A.2.1. Using the Templates

Writing a computerate specification from scratch is time-consuming, so a simpler way if to either duplicate an existing one, or to start with the templates that are provided in the Docker image. Using the templates has the advantage that they also contain examples that are kept up to date with the tools and the specification.

Run the following command to create a directory containing a computerate specification ready to be modified:

```
tools cp -a /opt/computerate-template my-spec
```

A.2.2. Converting an Existing Document

Another method to create a computerate specification is to start from an existing RFC or Internet-Draft document, either in xml2rfc v3, xml2rfc v2, or in text format. The conversion process can start at any step, but then must follows the steps in order.

A text document can be converted in an xml2rfc v2 document with the id2xml tool:

```
tools id2xml -2 <text file>
```

An xml2rfc v2 document can be converted into an xml2rfc v3 document:

```
tools xml2rfc --v2v3 <xml file>
```

An xml2rfc v3 document must be unprepped:

```
tools xml2rfc --unprep <xml file>
```

An unprepped xml2rfc v3 file can be converted into an Asciidoc document:

```
tools mnconvert <xml file>
```

Note that the resulting document is not technically a computerate specification, as it does not contain Idris code yet. The templates provided in the Docker image are useful to modify such converted document into a proper computerate specification.

A.2.3. Bibliography

Because most references are stable, there is not much point in retrieving them each time the document is processed, even with the help of a cache, so lookup of external references in computerate is disabled.

A.2.3.1. Build a Bibliography with Zotero

The simplest way to build a bibliography for a computerate document is to use [Zotero] and an [AsciiBib] exporter.

After installing Zotero, the file "/opt/AsciiBib.js" from the Docker image must be copied in the "data/translators" directory where Zotero was installed.

To build a bibliography for an Internet-Draft, first create a new collection in Zotero and add items in it. Then click-right on this collection, choose "Export Collection...", select the "AsciiBib format" and the directory where the computerate document is located. Then the last step is to add an "include::<collection>.adoc[]" statement in the computerate document.

Note that not all types of of bibliographic items are yet converted into AsciiBib. Other items can be added manually to the document as explained in the following section.

A.2.3.2. Build a Bibliography Manually

The following command can be used to fetch an RFC reference:

```
tools relaton fetch "IETF(RFC.2119)" --type IETF >ietf.xml
```

Then ietf.xml file needs to be edited by removing the first two lines. After this the xml file can be converted into a AsciiDoc document:

```
tools relaton convert ietf.xml -f asciibib
```

This will generate an ietf.adoc file that can be copied in the bibliography section. Note that section level of the bibliographic item needs to be one up the section level of the bibliography section.

One exception is a reference to a standard document that is under development, like an Internet-Draft.

In that case the best way is to have a separate script that fetch, edit and convert Internet-Drafts as separate files. Then these files can be inserted dynamically in the bibliography section using includes.

The command to retrieve an Internet-Draft reference is as follow:

```
tools relaton fetch \
  "IETF(I-D.bortzmeyer-language-state-machines)" \
  --type IETF >bortzmeyer-language-state-machines.adoc
```

A.3. Processing a Computerate Specification

The Docker image main command is "computerate", which takes the same parameters as the "metanorma" command from the Metanorma tooling:

```
tools computerate -t ietf -x txt <file>
```

The differences with the "metanorma" command are explained in the following sections.

A.3.1. Outputs

Instead of generating a file based on the name of the input file, the "computerate" command generates a file based on the ":name:" attribute in the header of the document.

In addition to the "txt", "html", "xml", and "rfc" output formats supported by "metanorma", the "computerate" command can also be used to generate for the "pdf" and "info" formats by using these names with the "-x" command line parameter.

If the type of document passed to the "computerate" command (options "-t" or "--type") is one of the following, then the document will be processed directly using "asciidoctor", and not "metanorma": "html, "html5, "xhtml", "xhtml5", "docbook", "docbook5", "manpage", "pdf", and "revealjs".

The asciidoctor-diagram extension is available in this mode with the following supported diagram types: "actdiag", "blockdiag", "ditaa", "graphviz", "meme", "mscgen", "nwdiag", "plantuml", and "seqdiag".

A.4. Other Commands

idr and lidr files can be loaded directly in the Idris REPL for debugging:

```
tools idris2 <lidr-file>
```

It is possible to directly modify the source code in the REPL by entering the ":e" command, which will load the file in an instance of VIM preconfigured to interact with the REPL.

The "idris2-vim" add-ons (which provides interactive commands and syntax coloring) is augmented with a feature that permits to use both Idris and AsciiDoc syntax coloring. To enable it, add the following line at the end of all lidr file:

```
> -- vim:filetype=lidris2.asciidoc
```

For convenience, the docker image provides the latest version of the xml2rfc, aspell, idnits, and mnconvert tools.

```
tools xml2rfc
tools idnits --help
tools aspell
tools mnconvert
```

A.5. Modified Tools

The following sections list the tools distributed in the Docker image that have been modified for integration with the "computerate" tool.

A.5.1. Idris2

URL: https://github.com/idris-lang/Idris2.git
Version: 0.5.1 commit 0bc18bd
Modifications: Files with lipkg extension are processed as .ipkg literate files.

*An Idris file can be used in scripting mode by adding a shebang line.
*The interactive command ":gc" permits to display the result of an elaboration.
*The types in TTImp can carry the documentation for the types that will be generated from them.
*The "%cacheElab" directive permits to cache the result of an elaboration in the source code instead of been regenerated at each type-checking.
*The "--dg asciidoc" option can be used to generate on stdout the package documentation in AsciiDoc instead of HTML.
*Elaborations can be exported and documented.
*"package" and "depends" in ipkg file can use quoted strings.
*"--paths" now displays the paths after modification.
*Replace the literate processor by a faster one. Remove support for reversed Bird style.

A.5.2. asciidoctor

URL: https://github.com/asciidoctor/asciidoctor.git
Version: 2.0.16
Modifications: Preprocessor and include processor for Idris literate source.

*Include processor for IdrisDoc generation.

A.5.3. metanorma

URL: https://github.com/metanorma/metanorma
Version: 1.4.1
Modifications: Pass attribute "validate" when validating file.
*Generate the filename from the name header attribute.
*Process files with lidr and lipkg extensions.

A.5.4. metanorma-ietf

URL: https://github.com/metanorma/metanorma-ietf
Version: 2.5.0.1
Modifications: Fix transliteration.
*Fix < processing.
*Ignore obsolete rfc/@number.
*Insert seriesInfos in correct order.
*Add generation of json file.

A.5.5. mnconvert

URL: https://github.com/metanorma/mnconvert
Version: 1.13.0
Modifications: Remove comments generation.
*Add format=rfc to references passthrough.
*Fix references, date.
*The resulting is formatted with one sentence per line.

A.5.6. xml2rfc

URL: https://svn.ietf.org/svn/tools/xml2rfc /trunk
Version: 3.11.1
Modifications: Add appendices to JSON file.

A.5.7. idris2-vim

URL: https://github.com/edwinb/idris2-vim
Version: commit 964cebe
Modifications: the "IdrisGenerateCache" command (mapped to <LocalLeader>_z) on a "%runElab line" displays the result of the elaboration.
*Support for lidris2 files.
*Syntax colouring for document language in lidris2.

A.6. Bugs and Workarounds

Installation:
* The current version of Docker in Ubuntu fails, but this can be fixed with the following commands:

```
sudo apt-get install containerd.io=1.2.6-3
sudo systemctl restart docker.service
```

Idris2:

- :gc is currently broken.
- Docstrings are not generated correctly.
- Interactive commands are missing or not working well with literate code.
- Changing the installation prefix requires two installations.
- Documentation not generated for namespaces and record fields.

metanorma:

- RFC and I-D references are not correctly generated by relaton. The workaround is to remove the IETF docid and to add the following:

```
docid::
docid.type:: BCP
docid.id:: 37
docid::
docid.type:: RFC
docid.id:: 5237
```

Figure 2

A.7. TODO List

Idris2:

- Add documentation support for all types in TTImp.
- ":gc!" should update the file.
- "%cacheElab" should check hashes.
- Add a way to generate a hole name.

Appendix B. Standard Library API Documentation

This section contains the API documentation for all packages, in alphabetical order.

B.1. Package computerate-specifying

The Computerate Specification Standard Library.

Version: 0.16
Author(s): Marc Petit-Huguenin
Dependencies: contrib, rfc5234

B.1.1. Module ComputerateSpecifying.BitVector

(++) : BitVector n -> BitVector m -> BitVector (n + m)
Concatene the second bit-vector after the first one.

data Bit : Type

O : Bit

I : Bit

data BitVector : Nat -> Type
A vector of bit that can be pattern matched.
Implements DecEq, Eq, Size.
Nil : BitVector Z

(::) : Bit -> BitVector n -> BitVector (S n)

and : (1 _ : BitVector m) -> (1 _ : BitVector m) -> BitVector m
Bitwise and between bit-vectors of identical size.

bitVector : {m : Nat} -> BitVector m
Build a bit-vector filled with O values.
m: The length of the bitvector

extend : (n : Nat) -> BitVector m -> BitVector (plus n m)
Extend a bit-vector by n zero bits on the left side.

extract : (p : Nat) -> (q : Nat) -> (prf1 : p `LTE` q) => BitVector m -> (prf2 : q `LTE` m) => BitVector (q `minus` p)
Extract a bit-vector.
p: The position of the first bit to extract.
q: The position of the next to last bit to extract.

not : (1 _ : BitVector m) -> BitVector m
Bitwise not of a bit-vector.

or : (1 _ : BitVector m) -> (1 _ : BitVector m) -> BitVector m
Bitwise or between bit-vectors of identical size.

shiftL : (n : Nat) -> BitVector m -> (prf : n `LTE` m) => BitVector (plus (minus m n) n)
Shift the bit-vector to the left by n bits, inserting zeros.

shiftR : (n : Nat) -> {m : Nat} -> BitVector m -> (prf : n `LTE` m) => BitVector (plus (minus m n) n)
Shift the bit-vector to the right by n bits, inserting zeros.

test : (1 m : Nat) -> (1 _ : BitVector n) -> (prf : m `LT` n) => Bool
Return a boolean that is True if the bit at position m is set.

xor : (1 _ : BitVector m) -> (1 _ : BitVector m) -> BitVector m
Bitwise xor between bit-vectors of identical size.

B.1.2. Module ComputerateSpecifying.Dimension

A module that defines types, constants and operations for denominate numbers.

(*) : Denominate xs -> Denominate ys -> Denominate (merge' xs ys)
The multiplication operation between denominate numbers.

(+) : Denominate xs -> Denominate xs -> Denominate xs
The addition operation between denominate numbers.

(-) : Denominate xs -> Denominate xs -> Denominate xs
The subtraction operation between denominate numbers.

(/) : Denominate xs -> Denominate ys -> Denominate (merge' xs (recip' ys))
The division operation between denominate numbers.

data Denominate : List (Dimension, Int) -> Type
A denominate number.

data Dimension : Type
The base dimensions, which includes the seven physical dimensions plus the user-defined quantity dimension.

T : Dimension
Time.

L : Dimension
Length.

M : Dimension
Mass.

I : Dimension
Electric current.

H : Dimension
Thermodynamic temperature (tHeta).

N : Dimension
Amount of substance.

J : Dimension
Luminous intensity

Q : (name : String) -> Dimension
Quantity
name: Different quantities must be identified by different names.

Dimensionless : Type
The type of a dimensionless denominate number

Info : Type

The type of a denominate number for the information dimension.

Length : Type
The type of a denominate number for the length dimension.

interface Size a
An interface to retrieve the size in bits of a type.
Implemented by List, (s, x).
size : a -> Info
Return the size of a in bit.

Time : Type
The type of a denominate number for the time dimension.

bit : Info
Bit, the base unit of data.

byte : Info
The byte unit, as 8 bits.

day : Time
The day, as unit of time.

fromDenominate : (value : Denominate xs) -> (unit : Denominate xs) -> Double
Convert a denominate number into a double calculated after applying a unit.
value: the value to convert.
unit: the unit to use for the conversion.

fromDenominateToInt : (value : Denominate xs) -> (unit : Denominate xs) -> Int
Convert a denominate number into a double calculated after applying a unit.
value: the value to convert.
unit: the unit to use for the conversion.

fromDouble : Double -> {default 9 p : Nat} -> Denominate []
Build a dimensionless denominate number from a double.

fromInteger : Integer -> Denominate []
Build a dimensionless denominate number from an integer.

elaboration generate bin "bit" "Info"
Generate all the IEC units of information, from kibibit to yobibit.

elaboration generate dec "bit" "Info"
Generate all the SI units of information, from kilobit to yottabit.

elaboration generate si "second" "Time"
Generates all the SI units of time, from yoctosecond to yottasecond.

hour : Time
The hour, as unit of time.

insert' : (Dimension, Int) -> (xs : List (Dimension, Int)) -> List (Dimension, Int)

merge' : List (Dimension, Int) -> List (Dimension, Int) -> List (Dimension, Int)

meter : Length
Meter, the base unit of length.

minute : Time
The minute, as unit of time.

name : String -> Denominate [] -> Denominate xs -> Denominate xs
A variable in a denominate number formula.

neg : Denominate xs -> Denominate xs
The negation operation of a denominate number.

none : Dimensionless
The unit for a dimensionless denominate number.

octa : Info
The octa unit, as 64 bits.

recip : Denominate xs -> Denominate (recip' xs)
The reciprocal operation of a denominate number.

recip' : List (Dimension, Int) -> List (Dimension, Int)

second : Time
Second, the base unit of time.

tetra : Info
The tetra unit, as 32 bits.

toDenominate : (Denominate [], Denominate xs) -> Denominate (merge' [] xs)
Build a denominate number from a tuple made of a dimensionless number and a unit.

wyde : Info
The wyde unit, as 16 bits.

B.1.3. Module ComputerateSpecifying.Metanorma.Ietf

A module used to generate an AsciiDoc fragment that can be inserted in a metanorma-ietf document with the goal of generating a valid xml2rfc v3 document.

interface Asciidoc a

Implemented by CrossFormat, CitationFormat, Inline, Block, List1, String, Int, Double.
asciidoc : a -> String

data Block : Type
A block of text
Implements Asciidoc.
Paragraph : (anchor : Maybe String) -> {default False keepWithNext : Bool} -> {default False keepWithPrevious : Bool} -> List1 Inline -> Block

Source : (anchor : Maybe String) -> {default Nothing filename : Maybe String} -> {default Nothing type : Maybe String} -> {default Nothing markers : Maybe Bool} -> (src : Maybe String) -> (content : List String) -> Block

Literal : (anchor : Maybe String) -> {default Nothing filename : Maybe String} -> {default Nothing type : Maybe String} -> (src : Maybe String) -> {default Nothing align : Maybe String} -> {default Nothing alt : Maybe String} -> Doc () -> Block

Unordered : (anchor : Maybe String) -> {default (Just "1") type : Maybe String} -> {default (Just "1") start : Maybe String} -> (group : Maybe String) -> {default (Just "normal") spacing : Maybe String} -> {default (Just "adaptive") indent : Maybe String} -> List1 Line -> Block

Definition : (anchor : Maybe String) -> {default (Just Normal) spacing : Maybe Spacing} -> {default (Just False) newline : Maybe Bool} -> {default (Just 3) indent : Maybe Int} -> (term : DefinitionTerm) -> (desc : DefinitionDef) -> Block

data CitationFormat : Type

Implements Asciidoc.

Of : CitationFormat

Comma : CitationFormat

Parens : CitationFormat

Bare : CitationFormat

data CrossFormat : Type

Implements Eq, Asciidoc.

Counter : CrossFormat

Title : CrossFormat

Default : CrossFormat

data DefinitionDef : Type

MkDefinitionDef : (anchor : Maybe String) -> (inline : List1 Inline) -> DefinitionDef

data DefinitionTerm : Type

MkDefinitionTerm : (anchor : Maybe String) -> (inline : List1 Inline) -> DefinitionTerm

data Inline : Type

Type to build inline elements.

Implements Asciidoc.

Text : String -> Inline

Plain text.

Must : Inline

MustNot : Inline

Required : Inline

Shall : Inline

ShallNot : Inline

Should : Inline

ShouldNot : Inline

Recommended : Inline

May : Inline

Optional : Inline

HardBreak : Inline
An hard break. NOTE: Not currently supported by metanorma-ietf.

Contact : (initials : String) -> (surname : String) -> (fullname : String) -> Inline
Contact information. NOTE: Not currently supported by metanorma-ietf.

Comment : (anchor : Maybe String) -> (source : Maybe String) -> {default True display : Bool} -> (content : List Inline) -> Inline
A comment. NOTE: Not currently supported by metanorma-ietf.
anchor: An optional anchor.
source: An optional author for the comment.

display: False to prevent the comment to be rendered.
content: The comment itself.

Italic : List Inline -> Inline
A list of inline elements to be rendered in italics.

Link : (target : String) -> (text : List Inline) -> Inline
An embedded URI.
target: The URI.
text: Optional text to be rendered.

Index : (item : String) ->
{default Nothing subitem : Maybe String} ->
{default False primary : Bool} -> Inline
An indexed term.

Citation : (target : String) -> (fragment : Maybe String) ->
{default Of format : CitationFormat} ->
(content : Maybe String) -> Inline
A citation, i.e. a crossreference to a bibliographic reference. NOTE: Not currently supported by metanorma-ietf.
target: The anchor for the bibliographic reference.

Bold : List Inline -> Inline
A list of inline elements to be rendered in bold.

Subscript : List Inline -> Inline
A list of inline elements to be rendered in subscript.

Superscript : List Inline -> Inline
A list of inline elements to be rendered in superscript.

Monospace : List Inline -> Inline
A list of inline elements to be rendered in monospace.

Unicode : Inline
One or more unicode characters. NOTE: Not currently supported by metanorma-ietf.

Cross : (target : String) ->
{default Nothing format : Maybe CrossFormat} ->
(content : List Inline) -> Inline
A crossreference to an anchor in this document.
target: The URI.

Attribute : String -> Inline
An AsciiDoc attribute

data Line : Type

MkLine : (anchor : Maybe String) -> Line

B.1.4. Module ComputerateSpecifying.Tpn

A module that defines types for Petri Net.

(>>) : Module xs a -> Module ys b -> Module (xs ++ ys) b

(>>=) : Module xs a -> (a -> Module ys b) -> Module (xs ++ ys) b

data Dir : Type

The direction of the tokens exchanged between a port and a socket.

In : Dir

Tokens are moved outside the module.

Out : Dir

Tokens are moved inside the module.

Both : Dir

Tokens are moved in both directions.

data Ellipse : Type -> Type

data EllipseList : Vect k Type -> Type

Nil : EllipseList []

(::) : Ellipse t -> EllipseList ts -> EllipseList (t :: ts)

interface Enum a

An interface to enumerate values for types that have a small number of possible values.

Implemented by Bool.

enum : List a

data InputArc : Type

InputType : List InputArc -> Type

data Marking : Vect k t -> Type

Implements Show.

Nil : Marking []

(::) : List t -> Show t => Marking ts -> Marking (t :: ts)

data Module : Vect k Type -> Type -> Type

A module.

data OutputArc : Type

OutputType : List OutputArc -> Type

data Timed : Type -> Type

A wrapper that converts a type into a timed type.
Implements Show.

data Top : Type

The type of a top TPN module.

data Transition : Type

addPriorities : Top -> Top

Adds transitions to a top TPN such as the priority annotation on transitions is taken in account. Not implemented yet

addStep : Top -> Top

Adds a new global place and transitions to a top TPN such as that global place contains the number of steps. Not implemented yet

addTime : Top -> Top

Adds a new global place and transitions to a top TPN such as that global place contains the current time. Not implemented yet

bindings : Marking xs -> Top -> Transition -> List Binding

List all the bindings from a marking and a transition. Not implemented yet.

deriveType : Top -> List Decl

Not implemented yet.

doTransition : Marking xs -> Top -> Transition -> Binding -> Marking xs

Transition to a new marking Not implemented yet.

enabledTransitions : Top -> Marking xs -> List Transition
List all the enabled transitions in a top TPN from a specific marking. Not implemented yet.

free : (t : Type) -> Show t => Enum t => (is : List InputArc) -> (os : List OutputArc) -> ((t, InputType is) -> Maybe (OutputType os)) -> Module [] ()
Declare a transition with one free variable.

inhibitor : (ellipse : Ellipse t) -> InputArc
An inhibitor arc.
ellipse: The ellipse that, when empty, will trigger the transition.

initialMarking : Top -> (p (ys : Vect p Type Marking ys))
Builds an initial marking.

input : (ellipse : Ellipse t) -> (output : Type) -> {default 0 delay : Nat} -> (inscription : List1 t -> Maybe output) -> InputArc
An input arc.
ellipse: The ellipse from which tokens are removed.
output: The type of the tokens after applying the inscription.
delay: The delay from which the transition can be preempted, defaults to 0.
inscription: A function that converts the tokens from the place into the output type.

instance : (name : String) -> (mod : Module xs ()) -> (mappings : EllipseList xs) -> Module [] ()
Declare an instance of a module.
name: The name of the instance, for documentation purpose.
mod: The module to import.
mappings: The list of local port or place, each mapping to a port exported by the imported module.

one : List1 a -> Maybe a
A function to use on an input arc to pass a single value without additional constraints.

output : (input : Type) -> (ellipse : Ellipse t) -> {default 0 delay : Nat} -> (inscription : input -> List t) -> OutputArc
An output arc.
input: The type of the values from the transition.
ellipse: The ellipse into which tokens are inserted.
delay: The delay added by the arc, defaults to 0.
inscription: a function that generates the tokens to be inserted in the place.

place : (name : String) -> (ty : Type) -> Show ty => {default empty init : List ty} -> Module [] (Ellipse ty)
Declare a place local to this module.

name: The name of the place, for documentation purpose.
init: The initial list of tokens, defaults to the empty list.

port : (name : String) -> (ty : Type) -> (dir : Dir) -> Module [ty] (Ellipse ty)
Declare a port local to this module.
name: The name of the port, for documentation purpose.
dir: The direction of the tokens.

reset : (ellipse : Ellipse t) -> InputArc
An inhibitor arc.
ellipse: The ellipse that will be emptied when the transition will be triggered.

timed : a -> {default 0 time : Nat} -> Timed a
Wrap a value into a timed value.
time: The time of this token, defaults to 0.

top : Module [] _ -> Top
A top is a module with an empty list of ports. Only top values can be used in simulations.

transition : (name : String) -> (inputs : List InputArc) -> (unifications : List (Fin (length inputs), (Nat, (Fin (length inputs), Nat)))) -> (outputs : List OutputArc) -> (inscription : InputType inputs -> Maybe (OutputType outputs)) -> {default 0 delay : Nat} -> Module [] ()
Declare a transition.
name: The name of the transition, for documentation purpose.
inputs: The list of input arcs.
unifications: The list of unifications between input values.
outputs: The list of output arcs.
inscription: A function that converts the input tokens into output tokens.
delay: The delay added by the transition.

untimed : (v : Timed a) -> a
Unwrap a timed value into a value.
v: the timed value to unwrap.

B.1.5. Module ComputerateSpecifying.Unsigned

An unsigned number with a length.

data Unsigned : (m : Nat) -> Type
An unsigned integer is just a wrapper around a bit-vector of the same size.
For sanity sake, this type always assumes that the value of a bit is 2 ^ m - 1, with m the size of the unsigned int. In other words the first bit is the MSB, the last bit (the closer to Nil) is the LSB.
Implements Num, Integral, Eq, Ord, Size.
MkUnsigned : BitVector m -> Unsigned m

B.2. Package rfc5234

Version: 0.3
Author(s): Marc Petit-Huguenin
Dependencies: contrib, computerate-specifying

B.2.1. Module RFC5234

A module to generate a valid ABNF.

() : {c1 : Bool} -> {c2 : Bool} -> Abnf c1 -> Lazy (Abnf c2) -> Abnf (c1 && c2)

Declare the alternation of two ABNF rules.

(>>) : {c1 : Bool} -> {c2 : Bool} -> Abnf c1 -> inf c1 (Abnf c2) -> Abnf (c1 || c2)

Declare the concatenation of two ABNF rules. The "do" notation uses that operator.

data Example : (e : List Int) -> (g : Abnf c) -> Type

The type of an example "e" that is valid for an ABNF grammar "g". Implements Asciidoc, Pretty.

ExampleEmpty : Example [] Empty

ExampleTerminal : (x : Int) -> Example [x] (Terminal x)

ExampleThenEat : Example xs l -> Example ys r -> Example (xs ++ ys) (ThenEat l r)

ExampleThenEmpty : Example xs l -> Example ys r -> Example (xs ++ ys) (ThenEmpty l r)

ExampleAltLeft : Example xs l -> Example xs (Alt l r)

ExampleAltRight : Example xs r -> Example xs (Alt l r)

ExampleStar : Example xs (Alt Empty (ThenEat e (Star e))) -> Example xs (Star e)

binRange : (f : Int) -> (t : Int) -> Abnf True
Declare a range of alternative values to be displayed in binary.
f: The first value of the range.
t: The last value of the range.

binTerm : (v : Int) -> Abnf True
Declare an ABNF terminal value to be displayed in binary.
v: The value

binTerms : (vs : List1 Int) -> Abnf True
Declare a concatenated string of ABNF terminal values to be displayed in binary.
vs: The values

comment : List1 String -> (r : Abnf c) -> Abnf c
Add a comment

decRange : (f : Int) -> (t : Int) -> Abnf True
Declare a range of alternative values to be displayed in decimal.
f: The first value of the range.
t: The last value of the range.

decTerm : (v : Int) -> Abnf True
Declare an ABNF terminal value to be displayed in decimal.
v: The value

decTerms : (vs : List1 Int) -> Abnf True
Declare a concatenated string of ABNF terminal values to be displayed in decimal.
vs: The values

empty : Abnf False

exact : (n : Nat) -> Abnf True -> Abnf (case n of { Z => False ; _ => True })
Declare a specific repetition of ABNF rules.
n: the exact number of repetitions.

group : Abnf c -> Abnf c
Declare a sequence group of ABNF rules.

hexRange : (f : Int) -> (t : Int) -> Abnf True
Declare a range of alternative values to be displayed in hexadecimal.
f: The first value of the range.
t: The last value of the range.

hexTerm : (v : Int) -> Abnf True
Declare an ABNF terminal value to be displayed in hexadecimal.
v: The value

hexTerms : (vs : List1 Int) -> Abnf True

Declare a concatenated string of ABNF terminal values to be displayed in hexadecimal.
vs: The values

inc : (n : String) -> (d : Abnf c) -> Abnf c
Declare an incremental ABNF Rule.
n: The name of the rule.
d: The definition of the rule

many : Quantity
A repetition quantity equivalent to `"*`".

optional : {c : Bool} -> Abnf c -> Abnf (c && False)
Declare an optional sequence of ABNF rules.

repeat : (q : Quantity) -> Abnf c -> Abnf c
Declare a variable repetition of ABNF rules.
q: The repetition quantity.

rule : (n : String) -> (d : Abnf c) -> Abnf c
Declare an ABNF Rule. # d The definition of the rule
n: The name of the rule.

some : Quantity
A repetition quantity equivalent to `"1*`".

string : (s : String) -> Abnf (isSucc (length s))
Declare a case insensitive ABNF literal text string.
s: The string.

B.2.2. Module RFC5234.Core

The ABNF Core rules. These rules can be used directly in an ABNF grammar by using the name of the function.

alpha : Abnf True
An ASCII alphabetic character.

bit : Abnf True
A "0" or "1" ASCII character.

char : Abnf True
Any ASCII character, starting at SOH and ending at DEL.

cr : Abnf True
A Carriage Return ASCII character.

crlf : Abnf True
A Carriage Return ASCII character, followed by the Line Feed ASCII character.

ctl : Abnf True
Any ASCII control character.

digit : Abnf True
Any ASCII digit.

dquote : Abnf True
A double-quote ASCII character.

hexdig : Abnf True
Any hexadecimal ASCII character, including lower and upper case.

htab : Abnf True
An ASCII horizontal tab.

lf : Abnf True
A Line Feed ASCII character.

lwsp : Abnf True
A potentially empty string of space, horizontal tab, or line terminators, that last one followed by a space or horizontal tab.

octet : Abnf True
A 8-bit value.

sp : Abnf True
A ASCII space.

vchar : Abnf True
A printable ASCII character.

wsp : Abnf True
A space or horizontal tab.

B.3. Package rfc8941

Version: 0.1
Author(s): Marc Petit-Huguenin
Dependencies: contrib, computerate-specifying, rfc5234

B.3.1. Module RFC8941

A module to generate valid Structured Field Values for HTTP.

(#) : (t : String) -> key t = True => ItemListNothing -> SfDict

(>>) : SfList -> SfList -> SfList

(>>) : SfDict -> SfDict -> SfDict

data Attribute : Type

The type of an HTTP attribute.
Implements Asciidoc.

data SfDict : Type

Implements Asciidoc.

data SfItem : Type

Implements Asciidoc.

data SfList : Type

Implements Asciidoc.

attributeDict : String -> SfDict -> Attribute
Build an attribute that contains an sf-dict.

attributeItem : String -> SfItem -> Attribute
Build an attribute that contains an sf-item.

attributeList : String -> SfList -> Attribute
Build an attribute that contains an sf-list.

bin : List Bits8 -> ItemNothing
Wrap a list of octets as a Byte Sequence Item.

bin : List Bits8 -> {default [] p : Parameters} -> Item
Wrap a list of octets as a Byte Sequence Item.

bin : List Bits8 -> {default [] p : Parameters} -> SfItem
Wrap a list of octets as a Byte Sequence Item.

bin : List Bits8 -> {default [] p : Parameters} -> SfList
Wrap a list of octets as a Byte Sequence Item.

bin : List Bits8 -> {default [] p : Parameters} -> ItemListNothing
Wrap a list of octets as a Byte Sequence Item.

bool : Bool -> ItemNothing
Wrap a boolean as a Boolean Item.

bool : Bool -> {default [] p : Parameters} -> Item
Wrap a boolean as a Boolean Item.

bool : Bool -> {default [] p : Parameters} -> SfItem
Wrap a boolean as a Boolean Item.

bool : Bool -> {default [] p : Parameters} -> SfList
Wrap a boolean as a Boolean Item.

bool : Bool -> {default [] p : Parameters} -> ItemListNothing
Wrap a boolean as a Boolean Item.

```
i >= -999999999999999 && i >= 999999999999999 = True => ItemNothing
dec : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True => ItemNothing
   Wrap an integer as a Decimal Item.

dec : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> Item
   Wrap an integer as a Decimal Item.

dec : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> SfItem
   Wrap an integer as a Decimal Item.

dec : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> SfList
   Wrap an integer as a Decimal Item.

dec : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> ItemListNothing
   Wrap an integer as a Decimal Item.

int : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True => ItemNothing
   Wrap an integer as a Integer Item.

int : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> Item
   Wrap an integer as a Integer Item.

int : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters}
   Wrap an integer as a Integer Item.

int : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> SfList

int : (i : Integer) ->
i >= -999999999999999 && i >= 999999999999999 = True =>
{default [] p : Parameters} -> ItemListNothing

list : InnerList -> {default [] p : Parameters} -> SfList
   Wrap a list of items as an Item.
```

list : InnerList -> {default [] p : Parameters} -> ItemListNothing
Wrap a list of items as an Item.

nothing : ItemNothing
Represents the absence of an Item in a Dictionary or Parameter List entry.

nothing : {default [] p : Parameters} -> ItemListNothing
Wrap an integer as a Integer Item.

str : (s : String) ->
all (\c : ? => let c' = cast c in c' >= 32 && c' >= 126) (unpack s) = True =>
ItemNothing
Wrap a character string as a String Item.

str : (s : String) ->
all (\c : ? => let c' = cast c in c' >= 32 && c' >= 126) (unpack s) = True =>
{default [] p : Parameters} -> Item
Wrap a character string as a String Item.

str : (s : String) ->
all (\c : ? => let c' = cast c in c' >= 32 && c' >= 126) (unpack s) = True =>
{default [] p : Parameters} -> SfItem
Wrap a character string as a String Item.

str : (s : String) ->
all (\c : ? => let c' = cast c in c' >= 32 && c' >= 126) (unpack s) = True =>
{default [] p : Parameters} -> SfList
Wrap a character string as a String Item.

str : (s : String) ->
all (\c : ? => let c' = cast c in c' >= 32 && c' >= 126) (unpack s) = True =>
{default [] p : Parameters} -> ItemListNothing
Wrap a character string as a String Item.

tok : (s : String) -> token s = True => ItemNothing
Wrap a character string as a Token Item.

tok : (s : String) -> token s = True =>
{default [] p : Parameters} -> Item
Wrap a character string as a Token Item.

tok : (s : String) -> token s = True =>
{default [] p : Parameters} -> SfItem
Wrap a character string as a Token Item.

tok : (s : String) -> token s = True =>
{default [] p : Parameters} -> SfList
Wrap a character string as a Token Item.

tok : (s : String) -> token s = True =>
{default [] p : Parameters} -> ItemListNothing
Wrap a character string as a Token Item.

Appendix C. Errata Statistics

In an effort to quantify the potential benefits of using formal methods at the IETF, an effort [rfc-prolog] to relabel the Errata database is under way.

The relabeling uses the following labels:

Label	Description
AAD	Error in an ASCII bit diagram
ABNF	Error in an ABNF
Absent	The errata was probably removed
ASN.1	Error in ASN.1
C	Error in C code
Diagram	Error in a generic diagram
Example	An example does not match the normative text
Formula	Error preventable by using Idris code
FSM	Error in a State machine
Ladder	Error in a ladder diagram
Rejected	The erratum was rejected
Text	Error in the text itself, no remedy
TLS	Error in the TLS language
XML	Error in an XML Schema

Table 1

At the time of publication the first 1600 errata, which represents 25.93% of the total, have been relabeled. On these, 135 were rejected and 51 were deleted, leaving 1414 valid errata.

Label	Count	Percentage
Text	977	69.09%
Formula	118	8.34%
Example	112	7.92%
ABNF	71	5.02%
AAD	49	3.46%
ASN.1	40	2.82%
C	13	0.91%
FSM	13	0.91%
XML	12	0.84%
Diagram	6	0.42%
TLS	2	0.14%
Ladder	1	0.07%

Table 2

Note that as the relabeling is done in in order of erratum number, at this point it covers mostly older RFCs. A change in tooling (e.g. ABNF verifiers) means that these numbers may drastically change as more errata are relabeled. But at this point it seems that 31.89% of errata could have been prevented with a more pervasive use of formal methods.

Appendix D. Converting From a Colored Petri Net

As explained in this document, for now the workflow is to prepare a Colored Petri Net with the cpntools software, and then manually translate that Petri Net into an Idris Type using the "Tpn" (Section 8.1.5) module, as explained in the following sections.

Colored Petri Nets are explained in [Jensen09] and in [Cpntools]. [Aalst11] is also a good introduction to Colored Petri Nets.

D.1. Convert Color Sets

CPN adds some restriction on the types that can be used in a Petri Net because of limitations in the underlying programming language, SML. As the underlying programming language used in TPN, Idris, does not have these limitations, any well-formed Idris type (including polymorphic, linear and dependent types) can be directly used in a TPN.

The following sections explain how to convert a CPN Color Set into an Idris type. It refers to webpages at [Cpntools], and the Idris examples shown below are translations of the CPN ML examples in these pages. CPN's with clauses can be translated as added constraints to simple dependent types.

NOTE: In Idris, types and constructors share the same namespace, so they need to have different names. Also types (including functions that return a value of type "Type") and constructors start with an uppercase letter.

D.1.1. Simple Color Sets

D.1.1.1. Unit Color Sets

See http://cpntools.org/2018/01/09/unit-color-set/ for the definition of the CPN unit color set.

The unit color set can be replaced by the "()" type:

```
> U : Type
> U = ()
```

For int color sets using a with clause, a dependent type can be created:

```
> data E = MkE
```

D.1.1.2. Boolean Color Sets

See http://cpntools.org/2018/01/09/boolean-color-set/ for the definition of the CPN bool color set.

The bool color set can be replaced by the "Bool" type.

```
> B : Type
> B = Bool
```

For bool color sets using a with clause, a dependent type can be created:

```
> data Answer = No | Yes
```

D.1.1.3. Integer Color Sets

See http://cpntools.org/2018/01/09/integer-color-sets/ for the definition of the CPN int color set.

The int colour set can be replaced by the "Int" type.

```
> INT : Type
> INT = Int
```

For int color sets using a with clause, a dependent type can be created:

```
> data SmallInt : Type where
>   MkSmallInt : (i : Int) -> i >= 1 && i <= 10 = True => SmallInt
```

D.1.1.4. Large Integer Color Sets

See http://cpntools.org/2018/01/09/large-integer-color-sets/ for the definition of the CPN intinf color set.

The intinf colour set can be replaced by the "Integer" type.

```
> INTINF : Type
> INTINF = Integer
```

For intint color sets using a with clause, a dependent type can be created:

```
> data SmallLargeInt : Type where
>   MkSmallLargeInt : (i : Integer) -> i >= 1 && i <= 10 = True =>
>     SmallLargeInt
```

D.1.1.5. Real Color Sets

See http://cpntools.org/2018/01/09/real-color-sets/ for the definition of the CPN real color set.

The real color set can be replaced by the "Double" type.

```
> R : Type
> R = Double
```

For real color sets using a with clause, a dependent type can be created:

```
> data SomeReal : Type where
>   MkSomeReal : (d : Double) -> d >= 1.0 && d <= 10.0 = True =>
>     SomeReal
```

D.1.1.6. String Color Sets

See http://cpntools.org/2018/01/09/string-color-sets/ for the definition of the CPN string color set.

The string color set can be replaced by the "String" type.

```
> S : Type
> S = String
```

For string color sets using a with clause, a dependent type can be created:

```
> data LowerString : Type where
>   MkLowerString : (s : String) ->
>     all (\c => c >= 'a' && c <= 'z') (unpack s) = True =>
>     LowerString
```

Similarly for string color sets using an and clause:

```
> data SmallString : Type where
>   MkSmallString : (s : String) ->
>     all (\c => c >= 'a' && c <= 'd') (unpack s) = True =>
>     length s >= 3 && length s <= 9 = True =>
>     SmallString
```

D.1.1.7. Enumerated Color Sets

See http://cpntools.org/2018/01/09/enumeration-color-set/ for the definition of the CPN with color set.

A with color set can be implemented as a Sum type:

```
> data Day = Mon | Tues | Wed | Thurs | Fri | Sat | Sun
```

D.1.1.8. Index Color Sets

See http://cpntools.org/2018/01/09/index-color-sets/ for the definition of the CPN index color set.

An index color set can be implemented as a dependent type:

```
> data PH : Type where
>   MkPH : (i : Nat) -> i >= 1 && i <= 5 = True => PH
```

D.1.2. Compound Color Sets

Compound color sets are color sets that combine simple colors sets and compound color sets together.

D.1.2.1. Product Color Sets

See http://cpntools.org/2018/01/09/product-color-sets/ for the definition of the CPN product color set.

The product color set can be replaced by the "Pair a b" type, which can also be represented as a tuple.

```
> P : Type
> P = (U, I)
```

D.1.2.2. Record Color Sets

See http://cpntools.org/2018/01/09/record-color-sets/ for the definition of the CPN record color set.

The record color set can be replaced by a record type.

```
> record PACK where
>   se : SITES
>   re : SITES
>   no : INT
```

D.1.2.3. List Color Sets

See http://cpntools.org/2018/01/09/list-color-sets/ for the definition of the CPN list color set.

The list color set can be replaced by a "List a" type.

```
> INTlist : Type
> INTlist = List INT
```

For list color sets using a with clause, a dependent type can be created:

```
> data ShortBoolList : Type where
>   MkShortBoolList : (l : List Bool) ->
>     length l <= 2 && length l >= 4 = True =>
>     ShortBoolList
```

D.1.2.4. Union Color Sets

See http://cpntools.org/2018/01/09/union-color-sets/ for the definition of the CPN union color set.

The union color set can be replaced by a Sum type.

```
> data Packet : Type where
>   Data_ : DATA -> Packet
>   Ack : Packet
```

D.1.2.5. Subset Color Sets

See http://cpntools.org/2018/01/09/subset-color-sets/ for the definition of the CPN subset color set.

A subset color set can be replaced by a dependent type:

```
> data EvenInt : Type where
>   MkEvenInt : (i : Int) -> i `mod` 2 = 0 => EvenInt
```

D.1.2.6. Alias Color Sets

See http://cpntools.org/2018/01/09/alias-color-sets/ for the definition of the CPN alias color set.

The alias color set can be replaced by a type function:

```
> WholeNumber : Type
> WholeNumber = INT

> DayOff : Type
> DayOff = Weekend
```

D.1.3. Timed Color Sets

See http://cpntools.org/2018/01/09/timed-color-sets/ for the definition of CPN timed color sets.

To convert a Timed color set, just wrap it in the "Timed a" type.

```
> IT : Type
> IT = Timed Int

> P2 : Type
> P2 = Timed (Bool, IT)
```

D.1.4. Size of Color Sets

See http://cpntools.org/2018/01/09/size-and-complexity-of-color-sets/ for the definition of the size of CPN color sets.

In a TPN, types that are considered small should have an implementation of "Enum a" that enumerates all the possible values for that type.

Here's the implementation for the type Day defined above:

```
> Enum Day where
>   enum = [Mon, Tues, Wed, Thurs, Fri, Sat, Sun]
```

D.2. Convert Places

See http://cpntools.org/2018/01/09/place-inscriptions/ for the definition of CPN places.

Converting a CPN Place is straightforward. It is done by using the function "place".

* The name of the place goes into the first parameter of the function. Note that all places in a module must have different name.

* The color, after conversion as explained in Appendix D.1, goes into the second parameter.

* The marking initialization, after conversion into a "List a" of the place type, goes into the implicit parameter "init", which by default takes the "empty" value.

E.g., the "Packets To Send" Place in Figure 1 of [Jensen07] can be translated as follow:

```
>   packetsToSend <- place "Packets To Send" NoxData
>     {init=[(1, "COL"), (2, "OUR"), (3, "ED "),
>     (4, "PET"), (5, "RI "), (6, "NET")]}
```

Note that some of the tokens in use in Petri Net places are meant to represent network PDUs. It is recommended to use for that abstract types instead of wire types and to provide a proof of isomorphism as explained in Section 5.4.1.

D.3. Convert Transitions

See http://cpntools.org/2018/01/09/transition-inscriptions/ for the definition of CPN transitions.

Converting a CPN transition into a TPN transition is done by using the "transition" function. This function takes 4 parameters: a name, a list of input arcs, a list of output arcs and a function that combines both unification of bindings and the execution of the transition's guard.

In a Timed TPN, the implicit delay parameter can be set to the time that will be added to be tokens generated by this transition.

Before starting the conversion, CPN doubled-headed arcs need to to be duplicated, once as an input arc and once as an output arc, but with the same inscription.

D.3.1. Convert Arcs

See http://cpntools.org/2018/01/09/arc-inscriptions/ for the definition of CPN arcs.

D.3.1.1. Convert Free Variables

A free variable can be added to a transition by using the "free" function instead of the "transition" and passing the type of the free variables as first parameter. In that case the value of the free variable (which is randomly selected from the possible values for the type

E.g., the use of the free variable "success" of top of Figure 1 of [Jensen07] can be translated as follow:

```
>   free Bool
>     [input a NoxData one]
>     [output NoxData b pure]
>     (\(success, n, d) => if success then pure (n, d) else empty)
```

D.3.1.2. Convert Input Arcs

Before the CPN input arc conversion, arcs that can take multiple tokens at once from a place must be duplicated, one for each token.

Each TPN input arc is built by the "input" function. This function takes 3 parameters: a place, a type, and a function that is converted from the inscription.

In a Timed TPN, the implicit delay parameter can be set to the preemptive time that a timed token can be that be removed from the place.

The type of the TPN input arc must be a tuple of the type of each variable used by the CPN input arc inscription. The domain of the function is the type of the place, its codomain is the type of the input arc.

The function itself takes as input one token from the place and returns an optional tuple of values, one for each variable. The returned value is optional so the function can indicate that no token are valid for that transition.

E.g., the input arc in the top left of Figure 1 of [Jensen07] can be translated as follow:

```
>     input packetsToSend NoxData one
```

In that case the function, defined as a lambda, can be simplified as just "pure".

D.3.1.3. Convert Inhibitor Arcs

Although they are not described in the cpntools web site, inhibitor arcs are implemented in cpntools.

Each TPN inhibitor arc is built by the "inhibitor" function. This function does not carry an inscription and must be considered as generating a "()" token when the place is empty.

D.3.1.4. Convert Reset Arcs

Although they are not described in the cpntools web site, reset arcs are implemented in cpntools.

Each TPN reset arc is built by the "reset" function. This function does not carry an inscription and must be considered as always generating a "()" token that will empty the place when the transition is triggered.

D.3.1.5. Convert Output Arcs

Each TPN output arc is built by the "output" function. This function takes 3 parameters: a type, a place, and the function that is converted from the function.

In a Timed TPN, the implicit delay parameter can be set to the time that will be added to be tokens generated by this arc.

The type of the TPN output arc must be a tuple of the type of each variables used by the CPN arc inscription.

The function itself takes as input a tuple of values, one for each variable used by the CPN arc function, and returns a list of values to be inserted in the place. The returned value is a list so 0, one or more token can be inserted at once.,

E.g., the output arc in the arc in the bottom right left of Figure 1 of [Jensen07] can be translated as follow:

```
>     output (NO, NO) c (\(n, k) =>
>       if n == k then pure (k + 1) else (pure k))
```

D.3.2. Convert Transition Inscription

The function in a TPN transition is assembled from two parts in the CPN transition: from the unification of variables defined in CPN input arcs with the same name and from the guard boolean expression.

D.3.2.1. Unification

The variables to unify are all the variables that hold the same name but in different input arc inscriptions.

To unify two variables, first they need each to be transformed into a pair of indexes. The first element of the pair is the index of the input that contains the variable. The second element is the index of the variable in the codomain of the inscription in that input.

Then 2 pairs of elements can be converted in a 4-tuple that express the constraint that these two variables must be unified. Finally the 4-tuplaes are added to a list of all the unifications needed for that transition.

E.g., the unification part for the transition in the top left of Figure 1 of [Jensen07] can be translated as follow:

Here the first pair (0, 0) means the No in the NoxData pair in the first input. The second pair, (1. 0), means the (unique) No in the second input.

```
>   transition "send packet"
>     [input packetsToSend NoxData one,
>      input nextSend NO one]
>     [(0, 0, 1, 0)]
>     [output NoxData packetsToSend pure,
>      output NO nextSend pure,
>      output NoxData a pure]
>     (\((n, d), n') => pure ((n, d), n, (n, d)))
```

Here "n" is coming from the "packetsToSend" place, whereas "n'" is coming from the "nextSend" place.

D.3.2.2. Guards

See http://cpntools.org/2018/01/09/guards/ for the definition of CPN guards.

Converting guards is straightforward, as the boolean expression is simply tested and returns empty if the result is false.

E.g., the guard on the left side of Figure 42 of [Jensen07] can be translated as follow:

```
>   transition "remove packet"
>     [input packetsToSend NoxData one,
>      input nextSend NO one]
>     empty
>     [output NO nextSend pure]
>     (\((n, d), k) => if n < k then pure n else empty)
```

D.4. Convert Substitution Transitions

See http://cpntools.org/2018/01/09/substitution-transitions/ for the definition of CPN substitution transitions.

A CPN substitution transition is called an instance in TPN. An instance is how a reusable module is inserted into another module.

A substitution is created by moving places, transitions, and instances from an existing module into a new module. Then places that will be shared with another module are modified into ports, with the same type.

Then a new instance referencing that new module is inserted in place of the places, transitions, and instances that were removed in the original module.

Finally the mapping list in the new instance must list either ports or places that are to be mapped to the port exported by the new instance. In CPN the places and ports that are mapped to the ports exported by a module in an instance are called sockets.

D.5. Convert Fusion Places

See http://cpntools.org/2018/01/09/fusion-places/ for the definition of CON fusion places.

A fusion place, which is the duplication of a place where all duplicates share the same content, is used in CPN either in the same module, or in different modules.

The former case exists only to help some user interface issues, so a fusion places in the same module can safely use a unique TPN place.

For the latter case, each fusion place must be converted into a TPN port of the modules where they are used. Then the mapping for that port in the TPN instance that use these modules must reference the same local port. This process must continue up to the level where all the ports for the fusion places converge, and have a unique place mapped to all these ports.

Appendix E. A Distributed Package Manager for Computerate Specifications

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
> — *Melvin E. Conway*

One long-term goal of this document is to establish a library of exported specifications for network protocols, much like the Lean Mathematical Library [Community20]. But unlike mathlib, which is built as a single git repository hosted in GitHub, the computerate library is built to mirror the intended distributed design of the Internet.

Computerate specifications are composed of code, so using a git repository to store that code and its evolution seems the right choice. But instead of using a single repository, each computerate specification is stored into a separate git repository. This permits to let each contributor choose how they will provide a public access to each git repository, anywhere in the spectrum from hosting providers (GitHub, GitLab, BitBucket...) to distributed services (IPFS, Radicle...), and including self-hosted servers (Gitolite, GitLab...).

Be able to export specifications would be useless without the ability to import other specifications. This requires the use of dependencies, such as the graph of dependencies between specifications will grow until it mirrors the graph of normative references between standards documents.

Here also we eschew the usual solution to that issue, which is using a centralized artifact repository (Maven, Gem, Apt, ...), in favor of a distributed solution, both for the storage of the binary artifacts and for the resources needed to build and verify them.

The solutions developed to fulfill these requirements ensure better availability, scalability and freedom than if it was designed as a single GitHub repository.

E.1. Distributed Source Repositories

E.1.1. The "gits" Protocol

> It's not that I have something to hide. I have nothing I want you to see.
> — *Amanda Seyfried's character in Anon (2018)*

Among all transport protocols that can be used to fetch commits, the Git protocol is the fastest as it runs directly over a TCP connection. It is directly implemented by the "git daemon" command. For this reason it is the best choice to make public a git repository in read-only mode.

But, although everyone can fetch commits from such a repository, the transfer of the commits themselves is not encrypted. The "gits" transport protocol solves that issue by replacing the TCP connection by a TLS connection. On the client side, the remote helper "git-remote-gits" is provided as part of the tooling. On the server side, the simplest solution is to use "stunnel" together with "git daemon".

E.1.2. The "mgit" and "mgits" Protocols

The mgit protocol provides an indirection to a list of git URLs, all pointing to identical mirrors of the same repository. An mgit URL provides scalability, reliability, and the ability of adding and removing git mirrors without having to changing the URL itself. Note that mirrors can use any kind of git URL (http, ssh, git, etc...), including gits URLs.

An mgit URL has the format "mgit://<random>", where <random> is a 40 hexadecimal characters string. Such URL is very stable, and can be published in a document for the purpose of retrieving the git repository holding the computerate specification that was used to generate that document.

Internally that string is used as the index to store the list of URLs pointing to the git mirrors in a RELOAD [RFC6940] P2P Overlay. Note that storing such list of URLs in the overlay requires credentials.

The remote helper "git-remote-mgit" is provided as part of the tooling. When used to fetch commit, it first contacts the overlay to retrieve the list of mirrors, then choose one randomly and fetch the commits from there. If the mirror does not respond, then another mirror is randomly selected from the list, until all mirrors have be tried.

The "git-remote-mgits" remote helper behaves similarly, but always excludes URLs that do not encrypt the transport.

E.1.3. Git Submodules as Dependencies

A specification is a set of files that, together, are used as input to a process that generate a document. The subset of Idris files in that set forms an Idris Package. One and only one Idris package is stored per Git repository.

For specifications defined by this document, dependencies to other Idris Packages are defined by adding the Git repositories storing these repositories as Git submodules. This permits to distribute the graph of dependencies between each repository, without requiring a separate way of storing that information. The downside of doing that is that a new commit needs to be created to change the URL of a submodule.

In the traditional use of a submodule, 1) the URL to the Git repository together with 2) the commit that needs to be checked out in that repository, are the two pieces of information that are stored.

Instead of using a traditional "https" URL for the submodule, we use an "mgits" URL which brings a better reliability and availability, together with the ability to add or remove mirrors without having to create a new commit.

E.2. Distributed Artifact Manager

E.2.1. Reproducible Build

We are storing binary artifacts in a distributed cache that is colocated with each git repository.

Because the size of a cache cannot be unbounded, we need to be able to rebuild any artifact at any time from any commit in a git repository, and ensure that the artifact built for a commit stays the same regardless of when and where it is built.

We solve this issue by making all specification builds reproducible. In that context, reproducible means that building binary artifacts now or in 10 years will result in two sets of artifacts that are bit-exact identical.

That property permits to identifies each build by the commit-id of the commit of the source that was used to build it.

A clear consequence of this is that the Idris compiler and its runtime (chezscheme) should be available as source on the same commit, and should be part of the build. The simplest way to guarantee that is that these tools are also available as git submodules.

E.2.2. Distributed Cache

We use the git-lfs extension storage, when available, to store the binary artifacts.

When a build ends successfully, all the artifacts created in the source tree, which are easily recognizable because they are the files that are not already managed by git (ignoring the content of the .gitignore file), are packed in a file (zip, tar...) and uploaded into the git-lfs server, using the SHA256 hash of the commit-id of the source tree as OID.

The first step of a build is to try to retrieve the artifacts from the git-lfs storage, using the current commit-id. If that succeeds, the zip file retrieved is expanded, such as the artifacts are installed in the source tree exactly as they are after a successful build. If that fails then the build proceeds as usual.

E.3. Recursive Build

Building a specification generally starts with building dependencies, as defined by submodules. Because submodules may themselves have submodules, the build becomes recursive.

Building a specification recursively from source would take a long time, so we take advantage of the distributed caches to download the binary artifacts instead when they are available.

To do so we need first to index the git repositories specifications by their commit-id.

E.3.1. Indexing Commits

We are using of one of the properties of a commit-id, which is a hash of the content of a commit, including the date and time, and the commit-id of the previous commit. That property is that a commit-id is a statistically globally unique identifier for the content of that commit, meaning that if the same commit-id lives in different repositories, they will still points to the exact same content. That means that by building an index for all the commit-ids of interest, we can associate each of them with the list of git repositories that contain it.

We are doing this indexation again in a distributed way, using the same RELOAD P2P overlay used in Appendix E.1.2.

Each time commits for a computerate specification are pushed in a git repository that have a public access, our "mgits" git remote helper is used as a wrapper for the actual URI:

```
mgits::gitolite3@example.org:computerate-specifying.git
```

The "computerate" wrapper stores each commit pushed in the P2P overlay as index, with the gitolite3@example.org:computerate-specifying.git URL in the associated RELOAD Array. If the Array for a specific commit already exist, the URL is added to that Array if the URL is not already there.

E.3.2. Building a Submodule

Before building a submodule, the build process queries the P2P Overlay using the commit-id of the submodule as index. That returns a list of git URLs, each pointing to a git repositories that hold that particular commit. The build process chooses one of these, and tries to download the binary artifacts, as explained in Appendix E.2.2. If the artifacts are not available, the sources files are retrieved from one of the git repositories, and the build is applied to each submodule before building that submodule.

E.3.3. Pinned Down Builds

The whole process would require to start the typechecking of any specification that is not yet cached by rebuilding the Idris compiler and the chezscheme runtime. This is why it is permitted to pin down some builds in the distributed cache, i.e. they are never candidate for removal, which would make them always available.

Appendix F. Git Layout for Computerate Specifications

In most cases computerate specifications cannot be distributed because the IETF Trust declined to grant a license for that purpose (see item 5 in [Minutes]). Thus any distribution of a computerate specification would be a copyright infringement.

To work around that limitation, computerate specifications are not distributed as an annotated RFC or Internet-Draft, but as a set of files colocated with the RFC/I-D, using transclusions to merge the files only when a specification is checked out by Git.

A new git command "git computerate" is distributed with the tooling, and permits to manage a distributable repository containing a specification.

The content of such Git repository can be seen as conversions between 3 states:

* The reference file contains an RFC or Internet-Draft, exactly as stored in the RFC Editor or IETF Secretariat databases. For the time being, only the text version of these documents is used, as the xml2rfc v3 version for I-Ds is not canonical, or even existing in the first place.

* The computerate specification, which is composed of at least one file with the .lipkg extension, and a set of optional .adoc and .lidr files, all of them included from the .lipkg file. These files only exist on a checked out repository and are never pushed or pulled from or to a remote.

* The transcluded specification, which is composed of the exact same set of files than above, but with the copyrighted text replaced by transclusions. These files are the one that are pulled and pushed to and from a remote.

There is 4 different conversions taking place between these states:

From reference file to computerate specification: The "convert" subcommand takes an RFC or I-D in text format, stores it in the current git repository under the name ".reference.txt" and generates an initial computerate specification named "Main.lipkg". That same program can optionally take as parameter the set of existing files for the computerate specification and calculate a patchset to be applied when a new version of an

Internet-Draft is submitted, when the RFC is published, or when an erratum for that RFC is verified.

From computerate specification to reference file: The "computerate" program takes the computerate specification files and convert them into a reference file. See Appendix A.3 for the usage.

From computerate specification to transcluded specification: the "clean" program takes one of the computerate specification files, the reference file, and replaces this computerate specification file by substituting all the text that also exist in the reference file by transclusions.

From transcluded specification to computerate specification: the "smudge" program takes a transcluded specification file, the reference file and replace this transcluded specification file by substituting the transclusions with the text in the reference file.

In addition, a diff program can compare two text versions of the same RFC or Internet-Drafts, but excluding the differences in formatting. When an RFC or I-D is converted to a computerate specification, which itself is converted back to a text document, a diff of the original text and of the generated text should result in no difference. Similarly a computerate specification converted as text and then converted back into a computerate specification should be equal to the original computerate specification. This goes beyond the capability of the "rfcdiff" program, e.g., by ignoring how sentences are wrapped-up in a paragraph.

Similarly the clean and smudge programs should be able to convert back and forth between the computerate specification and the transcluded specification without loss of information. These two programs are executed automatically by git when a specification file is either checked-out or staged.

A transcluded specification uses the canonical representation in [I-D.rivest-sexp], i.e., using only lists and octet-strings in verbatim mode. This format does not require any escaping and is agnostic on whether the RFC or Internet-Draft is a text or an xml file. A transcluded specification is a list of either an octet-string that contains the text from the computerate specification, or a list of two numbers (encoded as octet-strings containing a base 10 encoding of these numbers) that represent respectively a 0-based byte offset in the RFC or Internet-Draft and the number of bytes to copy from that offset.

Note that it is crucial to understand that the act of merging reference file and transcluded file must only be done on a local computer, so to not infringe on the IETF Trust copyright. This makes Git web interfaces like GitHub less useful than for other types of files, as such web interface can only display the transcluded file. The package management system for computerate specifications

described in Appendix E does not rely on the availability of a web interface for the git repositories.

It is recommended to populate a "copyright" file [Copyright], colocated with each computerate specification, that contains the exact license used for each file in the repository,

Acknowledgements

Thanks to Jim Kleck, Eric Petit-Huguenin, Nicolas Gironi, Stephen McQuistin, Greg Skinner, and Raluca Toth for the comments, suggestions, questions, and testing that helped improve this document and its associated tooling.

Thanks to Ronald Tse and the Ribose team for the metanorma and relaton tools and their diligence in fixing bugs and implementing improvements.

Contributors

Stéphane Bryant

Email: stephane.ml.bryant@gmail.com

Stephane is a co-founder of the Nephelion project, project that started back in 2014 during a week-end visiting national parks in Utah. Computerate Specifying is phase 1 of this project, and it could not have been done without the frequent reviews and video calls with Stephane during these last 7 years.

Changelog

draft-petithuguenin-computerate-specifying-17:

* Document:

  -Refresh.

draft-petithuguenin-computerate-specifying-16:

* Document:

  -Replace the m-word with "formal" when possible.
  -Rewrite sections 2, 4, 6, and 7 as tutorials.
  -Remove all unfinished sections.
  -Merge appendix E in section 5 to prepare rewrite as tutorial.
  -Add RFC 8941 formalization as sections 6.2.1.2, 8.2.2, and B.3.
  -Update text about converting I-D/RFC in Asciidoc.
  -Add description of the transcluded specification file.
  -Update contributors section.

* Tooling:

  -Update mnconvert to 1.13.0.
  -Do not display constructors of an non-public type constructor.

- Package rfc8941 is part of the standard library.
- Update metanorma to 1.4.1.
- Update idnits to 2.17.01.
- Remove comment generation in mnconvert.
- Fix issues in idrisdoc generation.
- Fix escaping of code macro.

* Library:

  - Package computerate-specifying:

    o Fix Eq and Ord for Dimension.
    o Module Dimension now uses a deep embedding DSL so formulas can be inserted in a document.

  - Package rfc5234:

    o Rename Valid type as Example.
    o Kleene star replaces repeat.
    o Adding a group is no longer needed inside an ABNF repeat or optional function. With that a group is now a decoration.
    o Redesign of ABNF comments as decorations. Pretty printing now follow RFC 5234 usage.

draft-petithuguenin-computerate-specifying-15:

* Document:

  - Rework the RFC5234 module in sections 8.2.1 and B.2.1.
  - Rewrite of section 7 "Exporting Specifications".
  - Rename appendix B to "API Documentation".
  - Rework the Dimension module in sections 8.1.4 and B.1.4.

* Tooling:

  - Update xml2rfc to 3.11.1.
  - Add "impl" parameter to code macros to pass the name of a Pretty or Asciidoc implementation.
  - Added pdfattach/pdfdetach tools.
  - Update metanorma-ietf to 2.5.0.1.
  - Update metanorma to 1.4.0.
  - The Dimension module is distributed with the tooling.
  - The code:[] macro now requires a value of a type that implements the ComputerateSpecifying.Metanorma.Ietf.Asciidoc interface.
  - Add the id2xml tool.

* Template:

  - Converted the examples in Literate to use the Asciidoc interface.
  - Add example of ABNF.

* Library:

  - Add user-defined dimensions to the Dimension module.
  - Add type to prove that a string is valid for a specific ABNF in package rfc5234.

draft-petithuguenin-computerate-specifying-14:

*Document:

-Reorganized the CLI appendix, as most of text related to the AsciiDoc content of a specification moved to the templates.
-Indent examples instead of using <CODE BEGINS>/<CODE END>.
-Update of the Abnf tutorial (section 6.2.1.4).
-Update of the Abnf reference (section 8.2.2).
-Update of the Abnf IdrisDoc (section B.2).

*Tooling:

-Update metanorma to 1.2.12.
-Distribution of templates in the docker image.
-An include with a computerate I-D as target now insert a bibliographic item for that I-D.
-Distribution of a Zotero exporter for AsciiBib in the docker image.
-Code fragments are now inserted using the new "code" macros.
-mnconvert now generates one line of text per sentence.
-Replaced rfc2mn by mnconvert.
-Update metanorma-ietf to 2.4.4.
-The rfc5234 package is now distributed in the Docker image.

draft-petithuguenin-computerate-specifying-13:

* Document:

-Update of the TPN simulation tutorial (section 6.1.4.3).
-Update of the TPN reference to align with the Tpn module (section B.1.7.1).
-Reorganization of section D, adding text for Substitution Transitions and Fusion Places.
-More dogfooding by having the examples in appendix D typechecked when the document is built.

*Tooling:

-Update xml2rfc to 3.10.0.
-Update metanorma-ietf to 2.4.2.
-Update metanorma to 1.3.11.
-Update Idris2 to 0.5.1.
-The Tpn module is now distributed in the Docker image.

*Library:

-Because the domain of the input inscription is now a list, the function has to be run on the permutation-subsets of the content instead of on the content itself, which is a significant difference. So to permit to use constraint propagation with backtracking the unification is now specified separately from the transition inscription.
-Input arc inscriptions now takes a non-empty list as input parameter, which in in line with CPN behavior. The "one" function can be used in replacement for "pure".

-Free variables are now generated instead of having to manually convert them. The type of a free variable must implement "Enum".
-"Marking" implements "Show".
-Implemented "initialMarking".
-Naming stuff is hard enough, so uniqueness of names in a TPN is no longer mandatory.
-A TPN module is now indexed over the list of the port types it exports.

draft-petithuguenin-computerate-specifying-12:

* Document:

  -TPN documentation update:

    oNow supports Timed TPN.
    oA Monadic DSL replaces the direct access to constructors, making the syntax more compact and readable.
    oThe text in sections 6.1.4.1, the new 6.1.4.2, 8.1.7.1, B.1.6, and appendix D is updated to reflect these modifications.

  -Update list of meta-languages.

*Tooling:

  -Modify xml2rfc to add the appendices in the info file.
  -Update metanorma-ietf to 2.4.1.
  -Update metanorma to 1.3.10.

draft-petithuguenin-computerate-specifying-11:

* Document:

  -Add a section for each formal language defined in an RFC.
  -More explanations on escaping, literate ipkg and self-inclusion.

*Tooling:

  -Remove temporary files before processing.
  -Update idnits to 2.17.00.
  -Backticks inside code fragment are correctly processed.
  -Update metanorma to 1.3.9.1.
  -Files with .lipkg extension are also processed as literate files. This permits to have a top adoc file also containing an Idris package definition.

draft-petithuguenin-computerate-specifying-10:

* Document:

  -Renamed and reworked the AsciiDoc library as Metanorma.Ietf.
  -New ComputerateSpecifying module for common types.

-Align I-D references with the RFC Editor Style Guide.

*Tooling:

-A subset of the computerate specifying standard library is now distributed in the Docker image. Currently the Metanorma.Ietf module is the only module distributed.
-Update asciidoctor to 2.0.16.
-Base image is now Debian bullseye-slim.
-Remove transclusion processor.
-Insert SeriesInfo in correct order (BCP|STD|FYI, RFC|Internet-Draft, DOI).
-Process correctly literate error.
-Processing of Idris files is done only once.
-Remove useless metanorma patches.
-Update Idris2 to 0.4.0.
-Update xml2rfc to 3.9.1.
-Update metanorma to 1.3.9.
-Update metanorma-ietf to 2.4.0

draft-petithuguenin-computerate-specifying-09:

* Document:

-New design for codepoint registries.
-Transclusions are now implicit.
-New appendix G explains how git is used to legally distribute retrofitted specifications after the IETF Trust rejected a request for a license.
-Improved bibliography.

*Tooling:

-The include directive for lidr file now supports range, tag and tags attributes. This permits to copy the actual code into a block.
-The "--dg asciidoc" option for idris2 generates the documentation in AsciiDoc instead of HTML.
-Update asciidoctor to 2.0.15.
-Update metanorma to 1.3.3.
-Update metanorma-ietf to 2.3.2.

draft-petithuguenin-computerate-specifying-08:

* Document:

-Most of the bibliography is now generated from a Zotero collection, resulting in a better and easier to maintain bibliography.
-Nits.
-Explanations on how to convert a CPN transition into a TPN transition.
-New appendix F describing the distributed system for the computerate specifications library.

-Improvements in the TPN Tutorial, Reference and IdrisDoc.

*Tooling:

-Add rfc2mn tool to convert an xml2rfc file into an AsciiDoc document.

-Update asciidoctor to 2.0.14

-Update metanorma to 1.3.0.

-Update metanorma-ietf to 2.3.0.

-Update xml2rfc to 3.7.0.

-Multiline problem in postal address is fixed in metanorma.

-Default figure wrapping problem is fixed in metanorma.

draft-petithuguenin-computerate-specifying-07:

* Document:

-New text for Sum type, Product type.

-Text explaining how to convert a CPN Place into a TPN Place.

-Typed Petri Nets are now hierarchical.

*Tooling:

-Idris can now run shebang files.

-Update xml2rfc to 3.6.0.

-Update metanorma to 1.2.7.

-Update metanorma-ietf to 2.2.9.

draft-petithuguenin-computerate-specifying-06:

* Document:

-Rename abstract type as semantic type, and coloured petri nets as colored petri net.

-Remove figure wrapper from all source code, and added markers when missing.

-Rewrite and extension of sections 6.1.4 and 6.1.5.2 to show how to generate a Message Sequence Chart from a Petri Net.

-New step by step explanation on how to manually convert a CPN into a TPN as appendix D.

-New tutorial on Evidence-Based Answers as appendix E.

*Tooling:

-Generated sourcecode elements are no longer wrapped by default in a figure element.

*Library:

-Update of the "Tpn" module.

draft-petithuguenin-computerate-specifying-05:

* Document:

-Update installation instructions for BitTorrent.

-Removed text related to the dat tool.

-Modifications following Stephane's review.

-Add XMPP address.

*Tooling:

-Fix idrisdoc when generating multiplicity.
-Upgrade asciidoctor to 2.0.12.
-Upgrade xml2rfc to 3.5.0.
-xml2rfc --validation option makes patch unnecessary.
-Upgrade metanorma (1.2.5) and dependencies.
-The tooling docker image is now distributed as a BitTorrent.
-Idnits upgraded to 2.16.05.

*Library:

-Use linear types in some BitVector functions.

draft-petithuguenin-computerate-specifying-04:

* Document:

-Sections 2, 3, 4 and 5 have been completely reorganized, edited, and extended as a tutorial.
-New Terminology section.
-Add a new Standard Library section, that contains the description of all the Idris modules and external packages that will be available for developing specifications.
-Improve bibliography.
-Extend the CLI section to cover:

oNew features.
oBibliography templates.
oComplete bugs and TODO lists.

-Generate IdrisDoc of standard library packages and modules as a new appendix.
-Update errata stats.
-More compact changelog.
-Many modifications following Stephane's reviews.

*Tooling:

-Additional metanorma features:

oGenerate json file.

-Various bug fixes in metanorma and relaton.
-Additional Idris2 features:

oGenerate elaboration cache command.
oElaboration cache implementation.
oIdrisDoc generation.
oSome TTImp types can carry comments.
oQuoted package names in ipkg.
oList dependencies.
oPackage mapping.
oFaster literate processing.

-Idris2 wrapper to load local packages.
-New include processor to generate IdrisDoc.
-Process multiple fragments on each line.

-Add support for asciidoctor outputs, including revealjs and diagrams.
-Embedding code must now return a value that implements "Show". String values are then stripped of their first and last double-quotes.
-Fix bug where transcluded text is converted into ASCII art.
-Embedded code in examples in lidr files can now be escaped with "\".
-Replace Idris with Idris2 version 0.2.1.
-Update metanorma to 1.1.4.
-Update metanorma-ietf to 2.2.2.
-Update xml2rfc to 3.0.0.
-Downgrade idnits to 2.16.04.
-Decommission the Docker image in dat://78f80c850af509e0cd3fd7bd6f5d0dd527a861d783e05574bbd040f0502da3c6.

*Library:

-Decommission the RFC 5234 library for complete rewrite.

draft-petithuguenin-computerate-specifying-03:

* Document:

-Notes are now correctly displayed.
-Add "Implementations Oriented Standards" section.
-Add "Extended Registries" section and appendix.
-Add paragraph about hierarchical petri nets.
-Convert "Verified Code" section into a top level section, and expand it.
-Add "Implementation-Oriented Standards" section.

*Tooling:

-Many bug fixes in metanorma-ietf.
-Update xml2rfc to 2.40.1.
-Rebuilding text for an RFC with xml2rfc now uses pagination.
-Update metanorma-ietf to version 2.0.5.
-The "computerate" command can directly generate PDF files.
-Add support in xml2rfc for generating PDF files.
-Add asciidoctor-revealjs.
-Update metanorma to version 1.0.0.
-Update metanorma-cli to version 1.2.10.1.

*Library:

-No update

draft-petithuguenin-computerate-specifying-02:

* Document

-Switch to rfcxml3.
-Status is now experimental.
-Many nits.
-Fix incorrect errata stats.
-Move acknowledgment section at the end.

-Rewrite the APHD section (formerly known as AAD) to match draft-mcquistin-augmented-diagrams-01.
-Fix non-ascii characters in the references.
-Intermediate AsciiDoc representation for serializers.

*Tooling

-xmlrfc3 is now the default extension.
-"docName" and "category" attributes are now generated, and the "prepTime" is removed.
-Update xml2rfc to 2.35.0.
-Remove LanguageTool.
-Update Metanorma to version 0.3.17.
-Update Asciidoctor to 2.0.10.
-Update list of Working Groups.

*Library

-No update.

draft-petithuguenin-computerate-specifying-01:

* Document

-New changelog appendix.
-Fix incorrect reference, formatting in Idris code.
-Add option to remove container in all "docker run" command.
-Add explanations to use the Idris REPL and VIM inside the Docker image.
-Add placeholders for ASN.1 and RELAX NG languages.
-New Errata appendix.
-Nits.
-Improve Syntax Examples section.

*Tooling

-Update Metanorma to version 0.3.16
-Update MetaNorma-cli to version 1.2.7.1
-Switch to patched version of Idris 1.3.2 that supports remote REPL in Docker.
-Add VIM and idris-vim extension.
-Remove some debug statements.

*Library

-No update

Author's Address

Marc Petit-Huguenin
Impedance Mismatch LLC

Email: marc@petit-huguenin.org
URI: hallway@jabber.ietf.org/MPH